ASSE - ADA SIMULATION SUPPORT ENVIRONMENT

Heimo H. Adelsberger

University of Economics
Vienna, Austria

Abstract

An Ada Simulation Support Environment (ASSE) is presented. It covers all forms of combined continous and discrete modeling techniques, including a transaction flow part. The ASSE is applicable on two levels: on a lower level using different subprogram packages and on a higher level, using interactive model design and verification packages as well as interfaces to data base managers, graphics and statistical analysis systems. For the lower level packages the stress is layed on clear, simple, universal and nonrestricted concepts; for the higher level packages on the fact, that simulation is a human activity, where the "man in the middle" has to be supported.

INTRODUCTION

We present in this paper a design of a Simulation Support Environment. The computer language we have chosen for this project is Ada. The basical layout for this design is similar to the APSE (Ada Programming Support Environment (14)) concept. There, the programming language Ada lays in the center and the supporting tools are grouped around this center in different layers. Here, different packages are grouped in a hierarchical and parallel manner to support the simulation model design, the software development process, the data analysis and the model programmers.

Background

There are in principle three methods of proceeding, when doing simulation on a digital computer (which methods certainly cannot always be differentiated exactly).

(a) one can choose a general purpose language like FORTRAN, PL/I, Algol, APL ... and write ones own program. There are millions (or more?) of examples of FORTRAN programs (including some dozens of programs of the author himself);

(b) one can choose a general or specific simulation language like Simscript, GPSS, Simula ..., one has to learn this language and write ones program (much more easily than in (a));

(c) one can use a package system or a kind of preprocessor, written in a general purpose language, one has to learn to use this system and/or preprocessor and write ones program (much more easily than in (a)), and has not to learn a new language; examples are GASP and Simpas.

SLAM is somewhere in between, perhaps more close to (b) then to (c).

A well designed solution of the package or preprocessor principle is in our opinion by far superior to the other possibilities.

Arguments

If we consider the historical background, when e.g. Simula and Simscript have been designed, the reason hasn't been just only to make a simulation language, but to overcome the deficiencies of the then available programming languages (mainly FORTRAN). From the preface to the first edition of "The SIMSCRIPT II Programming Language":

"SIMSCRIPT II is a rich and versatile computer programming language well suited to general programming problems, though designed originally for discrete-event simulation applications." (06)

For GPSS we can observe, that from the very beginning there have been tendencies to break up the rigid limitations of such a special purpose simulation language, first in providing assembler interfaces, then through embedding GPSS in a more general simulation language, which itself can be embedded in a general high level programming language (PL/I) (10).

On the other hand neither the data nor the control structure of a simulation program differs essentially of programs in other fields. But what is typical for simulation programs? On the one hand well known tasks, used in different areas of data processing

Proceedings of the 1982
Winter Simulation Conference
Highland * Chao * Madrigal, Editors

82CH1844-0/82/0000-0089 $00.75 © 1982 IEEE

(like queue handling), on the other hand more specific problems (like to manage a next event logic in discrete simulation).

Therefore a programmer, who has to write simulation programs, is primarily interested in as good a programming language as possible, and in good and easily applicable tools, which support all his needs.

Now we will (briefly) discuss a hot problem: What means a good programming language and are there (m)any? In our opinion a good programming language reflects the state of the art in programming techniques and in software engineering (or the state of the art not too long ago!); has a not too complicated structure (or the approach to a very useful, standardized subset of the language is not to hard) and is widely used, i.e. compilers are available on practically every computer. Is there any? No! Is there any chance that there will be a language, which comes close to these reqirements? Yes, we hope, that Ada will be this language, because we believe, that if it isn't Ada, there will be no other such language for the next ten years.

What does Ada offer?

From the introduction to the "Reference Manual to th Ada Programming Language":

"... Thus the language is a modern algorithmic language with the usual control structures, and the ability to define types and subprograms. It also serves the need for modularity, whereby data, types, and subprograms can be packaged. It treats modularity in the physical sense as well, with a facility to support separate compilation.

In addition to these aspects, the language covers real time programming, with facilities to model parallel tasks and to handle exceptions. ...

Ada was designed with three overriding concerns: a recognition of the importance of program reliability and maintenance, a concern for programming as a human activity, and efficiency. ...

Concern for the human programmer was also stressed during the design. Above all, an attempt was made to keep the language as small as possible, given the ambitious nature of the application domain." (13)

What are the most important features of Ada? Data abstraction and information hiding (packages, generics, private types, overloading, redefination of operators), parallel processing and synchronization, separate compilation.

Many critics mean, that the Ada design team has failed in keeping the complexity of the language under control. But - and that is our personal view - a language with a complex structure, and therefore certainly not simple, simplifies the work, if complex and large software products are to be developed.

But to prevent seeming too enthusiastic: we have some criticism ourselves. We will shortly touch one problem, because it is strongly related to our project and concerns the important principle of data hiding or encapsulated data. Data hiding is designed to prevent the user from unintentionally erroneus access of data. Ada supports this idea in form of packages and (limited) private types. The basic principle in Ada is hierarchical: a (standard) package - like a queue manager - hides its information (e.g. about pointers) from the calling program. Access to the data is only possible via subprograms, provided by the package. Therefore a hierarchical model: The using program on top, the hiding package underneath! We think, that there are also symmetrical situations, where two packages share common resources and each package likes to hide its information from the other package. There is no possibility in Ada to implement such a situation in a natural way without producing an overhead. We shall come back to this point in discussing our entities - attributes - sets - manager.

A short but incomplete survey about Ada for users not familiar with Ada:

Ada is a language with a context-free syntax and with identifiers of in principle arbitrary length to provide a readeable code, and it is based on the principle of strongly typing.

(1) Scalar types: enumeration, character, boolean, integer, floating and fixed point reals; arrays and records as composite types with components of any types and with an optional dynamical structure; access (= pointer) types.

(2) Overloading of subprogram identifiers (procedure and function/operator symbols) and enumeration literals. Subprograms can be called recursively and are reentrant.

(4) Packages: information hiding and exact definition of the interface, which is given in the visible part of the package declaration; private and limited private types.

(5) Tasks: parallel processing with entry, accept, select, delay and abort statements and the possibility to set priorities.

(6) Seperate compilation.

(7) Exeption handling.

(8) Generic program units.

THE ADA SIMULATION SUPPORT ENVIRONMENT

The DoD Ada project not only consists of the language description, it specifies as well requirements for the Ada Programming Support Environment (APSE), known as "Stoneman".

The structure of a APSE is represented in form of layers or levels:

In the center lies the hardware and host software as appropriate. The Minimal Ada Program Support Environment (MAPSE), which provides a minimal set of tools to develop and support Ada programs, lies around a Kernel (KAPSE), which consists of communication and run-time support functions, and which enables the execution of an Ada program. In the outer layer, the APSEs, a fuller support of particular applications and methodologies are provided.

We have chosen the same concept for our project of a simulation support package system, which itself as a particular application will lie in one APSE. We call this package system Ada Simulation Support Environment (ASSE) in accordance to the similar KAPSE - MAPSE - APSE concept.

level 0:

"Hardware and host software as appropriate": the Ada programming language and the MAPSE.

level 1:

Kernel Ada Simulation Support Environment, which provides utilities to execute simulation programs:

queue_manager: a general queue management

random_variates: a package to generate random numbers and variates from specific distributions

simple_statistics: perform basical statistical computations (mean, variance, maximum, minimum, histograms ...) for time- and value weighted variables

level 2:

Minimal Ada Simulation Support Environment, consists of the actual simulation packages:

eas_manager: a general entities - attributes - sets manager

simulation_control: a powerfull package, which allows to control all types of combined continous, discrete event, activity scanning and process oriented models

transaction_flow_manager: a package to build models similar to GPSS, Q-GERT and the network part of SLAM

level 3:

Ada Simulation Support Environment, which consists of packages to support the whole task (software engineering process around a simulation project from system data analysis and preparing model input data over interactive model design and verification until validation together with model output data analysis), therefore supporting the "man in the middle". (12)

A list of the packages:

model_design
model_verification
model_documentation

screen_printer_IO

data_manager
graphics
statistical_analysis

Where standard packages can be used, we provide just interfaces to these standard packages, specifically tuned for simulation purposes. We expect, that soon packages for statistical analysis, graphics, data base managers and use input - output systems will be available in Ada. The later, a powerful, universal and uniform screen I/O and printer output package is developed by ouerselves, because we think, that such a package is a sine qua non for good software products. But the heart of the ASSE consists of interactive model design, documentation and verification packages, which, under the control of an Ada syntax and semantic check(!) renders possible a fast, simple and well structured model design and documentation, and includes as well a powerful facility to test and to update a model interactivily.

We can not give in this paper a full documentation of the ASSE (01). We pick up some important parts of our design and give some rationals of our form of implementation. We explain the eas_manager more in detail, to show, that the complexity of Ada simplifies the user's work and renders possible a clear and self documenting form to write programs. We give an example too, which shows how the Ada Simulation Support Environment is used. At last, we give a glimpse of how the transaction flow manager is designed and how it can be used.

But first, we are going to expose the main guidelines for our packages:

(1) Our packages shall be usable at two levels:

-- on a high level, which is very user-friendly and where the usage is decidely simple, but with a possible disadvantage of occasional limitations;

-- on a low level, where a deeper insight of all concepts is neccessary, but with the advantage, that the user can do practically everything he wants to do (e.g. to use the eas_manager in a non simulation context just to keep track of different entities and set/queues; or to exchange our queue_manager with an user written package to test for instance different queuing algorithms for the event calender).

We feel, that the effort to use our packages on the lower level is e.g. for discrete event models a little bit higher than in GASP/SLAM, Simscript or Simpas; for the transaction flow manager much higher than in GPSS and higher than in Q-GERT/SLAM; but with the advantage of using such a powerfull language like Ada and the additional advantage, that every exceptional situation in a simulation model design can be mastered.

On the higher level, using the interactive model design package, the effort to do simulation is much lower than in the above mentioned languages, and here the user comes to the total advantage of the power of our Ada Programming Support Environment. Just one example: There is no need anymore for transaction flow models, that the user has to act as his own interpreter to transform his network model in a statement form (like in GPSS or SLAM). In our system, the network will be designed graphically at the screen, using a light pen, if possible, and will be debugged in the same manner too.

(2) user orientation and simplicity

A software designer has many choices how to implement such a package system. We stated, that the user's interests, who should be able to use these packages in a way as naturally and simple as possible, has to be the principal aim.

(3) no (unnecessary) limitations and a clear, well structured, approach

We shall describe by two examples, what me mean:

(a) eas_manager: no limitations on numbers of queues and entities (apart from memory limitations); no differencies between permanent and temporary entities; one entity can be a member of one, more or no queue at all; queues can contain entities of different kinds.

(b) transaction_flow_manager: The network as a whole, a part of a network and one node are objects of the same type! This brings the big advantage that all the good concepts in programming techniques, like structured programming, modularity, generic units and information hiding are transferable to the design of transaction flow models.

A short survey of the project state:

At the moment we still have to fight against contrary working conditions: we have to use only a micro computer and can use only a subset of Ada (e.g. without tasks). But we hope to move soon with our project onto a VAX.

The packages depending on tasking are only designed; the eas_manager, the simulation_control and the transaction_flow_manager are inplemented in a non tasking version. The screen_printer I/O package, a combined and uniform screen I/O and printer output system, a very useful and powerful general tool not only for simulation projects, is allready finished. The interactive system_design package exists for the most part, but depends still on the Ada syntax and semantic check program, which will be finished soon. The graphic package will be developed as soon as we can get access to a good graphic terminal and graphic software. The interactive model trace program exists in a simple version and will be improved as soon as we can get access to a graphic package. Then a visual trace of a network model can be performed too! The random variates package is implemented in a preliminary version, using the standard algorithms. The data analysis package will be the last package we are going to work on, hoping that until then such standard statistical packages will be available in Ada, with - hopefully - well defined interfaces, so that these packages and our packages can be linked.

QUEUE_MANAGER:

A generic package, used by the eas_manger, to manage sets, FIFO, LIFO and ranked queues, with a special queue-type for event-calendars. This special type of a 'low value first out' queue type is intended to implement specific fast algorithms for event calendars. Currently all these queues are simply implemented as double linked lists. This package is normally not directly visible for the user, because he accesses queues on a higher level via the eas_manager.

EAS_MANAGER:

A generic package to manage entities, attributes and set/queues.

Principles of the design:

As already mentioned, our requirements have been: easy to use, no limitations! Before we will give an extract of the generic package declaration, we will show the easy use of the package by means of examples. This shall demonstrate as well, that Ada enables to write self-documenting code. Then we shall give some rationals for our form of implementation:

The different entity types are introduced by:

```
  type entity_name is (person, car);
```

and queues by

```
  type queue_name is (teller_one, special_teller,
                      traffic_light);
```

The attributes are introduced by

```
  type attributes (kind: entity_name) is
     record
        case kind is
           when person =>
              sex: gender;
              age: integer range 0 .. 130;
           when car    =>
              seats: integer range 1 .. 8;
        end case;
  end record;
```

Entities have to be declared before they can be used:

```
 alan, bernard, charles, henry, jean: entity;
                                      -- some persons
 next_person:   entity;          -- will be used to
                        -- denote an unspecific person
 a_car: entity;                  -- a car
```

Queues have to be initialized, before they can be used:

```
 queue_init (teller_one);
 queue_init (special_teller, HVFO, 10);
```

If the second parameter in 'queue_init' is missing, FIFO ranking is the default queue ranking order. The third parameter, if present, restricts the queue capacity. HVFO means high value first out.

The exception queue_size_error will be raised, if an attempt will be made to insert in a full queue or to remove from an empty queue. Two boolean functions, queue_full and queue_empty, can be used to check the state of the queue.

To 'create' a person, one has to say:

```
   entity_create (henry, person);
```

or with automatical insert in a queue:

```
   entity_create_and_insert (teller_one,
                             henry, person);
```

Assignments to attributes can be made by

```
   henry.age:= 18;
   henry.sex:= male;
```

To insert this person in queue special_teller too (why not!), one has to say

```
   queue_insert (special_teller, henry, 15.0);
```

where 15.0 denotes the priority value.

To remove the first member of a queue, one has to say

```
   queue_remove (teller_one, next_person);
```

or

```
   queue_remove (special_teller, next_person,
                 urgency);
```

for a ranked queue. If the next person would be 'henry', urgency would be 15.0 .

To remove a specific person, one has to say

```
   queue_remove (teller_one, henry, this_person);
```

To remove a person from a specific place (e.g. from the third place), one has to say

```
   queue_remove (teller_one, next_person,3);
```

To destroy an entity, a

```
   entity_destroy (henry);
```

is sufficient. It will be removed from all queues, where it is a member, and afterwards no access to its attributes via henry.age or henry.sex is possible anymore.

If ranked queues are used, a floating point real type has to be passed to the eas_manager at package instantation time, declaring the type of variables, on which ranked queues are ordered. Furthermore, two optional procedures can be passed as generic parameters:

```
   entity_init (e: in out entity);
```

which can be used to initialize the entity's attributes (which will be automatically called by the entity_create procedure) and

```
   do_something (e: in out entity;
                 v: out ranking);
```

which is automatically called for each entity when the 'for-all-entity-loop' procedure queue_loop(qe: queue_name) is used.

Realisation:

A useful characteristic of Ada is, that subprogram names can be overloaded and, if a subprogram declaration specifies a default value for an 'in' parameter, then the corresponding parameter may be ommitted from a call. Further on, actual parameters may be passed in positional order or by explicity naming the corresponding formal parameters. Positional parameters and named parameters may be used in the same call.

These features of Ada render possible a very natural formulation of the queue_insert and queue_remove procedures. There are in principle two different meanings of insert and remove: We can speak of an unknown entity on a specific place or we can speak of a specific entity on an unknown place in the queue. Therefore the meaning of the parameter 'rank' in these procedures shall be:

If rank in 1 .. integer'last, we mean this specific place in the queue, whereby a value greater than the current size of the queue denotes the end of the queue; if rank = 0, then the place is determined by the default ranking criterion of the queue. Therefore we stated the default parameter for 'remove' as :=1, because 'remove' has normally the first member in view; and :=0 for 'insert', because normally entities are ranked according the default queue ranking criterion. If value ranked queues are involved, there is a need for an additional parameter, the rank_value (the "priority"). Such a parameter is useless for FIFO and LIFO queues. Moreover, because we think it so important, that a code is easy to read, we overload all these procedures with procedures, where first_member, last_member and this_member can be used instead of 1, -1 and 0.

The sequence of parameters for ranked queues is

```
  queue_name - entity - rank - rank_value;
```

for FIFO and LIFO queues just the first three.

We think, that our examples above have been self explaining. We have to anote, that for remove calls, the entity parameter acts logically partly as an in and partly as an out parameter: We have chosen in our examples the variable name 'next_person', if the parameter acts as a parameter of mode 'out', which means, that an unknown queue member is referenced via a specific place (=rank). The contrary holds for variables like 'henry' a.s.o.

```
-----------------------------------------
-- package declaration eas_manager
-----------------------------------------

with queue_manager;
generic
   type entity_name is <>;
   type queue_name is <>;
   type attributes (kind: entity_name)
      is limited private;
   type entity is access attributes;
   type ranking is digits <>;
   with procedure entity_init (e: out entity) is <>;
   with procedure do_some_thing (e: in out entity;
                        v: out ranking);) is <>;
   with procedure connect_info (e: in out entity;
        get_hidden_info,
            assign_flag: in  boolean) is <>;
```

```
package eas_manager is

   type hidden_info is limited private;
   type queue_ranking is (FIFO,LIFO,HVFO,LVFO,EVCA);
   type queue_place is (first_member, this_member,
                        last_member);

   procedure set_link (e: in out entity;
                      pp: in out hidden_info;
         get_hidden_info,
            assign_flag: in boolean);

   procedure queue_init (
              qe: in queue_name;
               r: in queue_ranking := FIFO;
               m: in integer       := integer'last);
--
-- from the following subprogram declarations, as
-- queue_destroy, queue_info, queue_full,
-- queue_empty, queue_size, queue_insert,
-- queue_remove, entity_create, entity_destroy,
-- entity_create_and_insert, queue_loop;
-- most are overloaded; we give as an
-- example the procedures queue_insert:
--
-- rank=0 ... default queue ranking!!
--
   procedure queue_insert (
               qe: in queue_name;
                e: in out entity;
             rank: in integer := 0);
   procedure queue_insert (
               qe: in queue_name;
                e: in out entity;
             rank: in queue_place);
   procedure queue_insert (
               qe: in queue_name;
                e: in out entity;
             rank: in integer := 0;
       rank_value: in ranking);
   procedure queue_insert (
               qe: in queue_name;
                e: in out entity;
             rank: in queue_place;
       rank_value: in ranking);
private

   package entity_queue is new queue_manager (
      entity,
      ranking);
   use entity_queue;

   type hidden_record is record
      pointer: entity;
      queue_info: array (queue_name) of TPRLink;
   end record;

   type hidden_info is access hidden_record;

end eas_manager;

-----------------------------------------
-- end package declarat .on eas_manager
-----------------------------------------
```

We see, the type entity is an access type for the record type attributes. But because the eas manager has to have control over the record too, there has to be an additional component added to the record, called 'info', which is from type 'hidden_info'. This type is provided by the eas manager, and is limited private. This means, that the user has absolutely no access to this information, he can make no assignments to this component nor can he compare for instance

```
   henry.info /= king.info
```

(the only possibility for the user would be to declare variables of this type, but he can't use them).

The type 'hidden_info' is limited private, and is described in detail in the private part of the package declaration:

To have access to the private type tpr_link (in package queue_manager), which contains the link information of a specific queue, a generic package instantiation of the queue_manager is needed first. Then the type 'hidden_record' is declared, which contains the component 'pointer', the entity access variable itself, a principally redundant information, and the component queue_info, an array of link_pointer's for each user-defined queue. Then, the type 'hidden_info' can be declared as an access variable to the type hidden_record.

If we focus on the generic parameter part at the beginning of the package declaration, we see, that the entity and queue types are passed as enumeration or integer types (it is possible to declare the discriminant for the entities and queues as integer types). The attributes record type is passed as a limited private type, only the discriminant is known by the eas_manager (and this uncomplete information is - as we will see soon - the reason for the already mentioned troubles). The next three parameters are obvious: the type on which a queue can be ranked and the two above mentioned procedures. The last parameter, the procedure 'connect_info', is at first sight dubious. But this procedure is the tribute to the fact, that the eas_manager has no access to the structure of records from type 'attributes'. So this procedure, which is part of the user's main package, calls the procedure set_link, which is again in the package eas_manager. Procedure connect_info just makes possible the access to the so important information e.info. (To make it clear: only the eas_manager can process the information, stored in e.info; only the user's program, which defines the record type 'attributes', knows, where the information is, but is on the other hand unable to process it). Two flags are needed to convey, how the informations have to be interchanged.

We concede, that this procedure connect_info breaks the principle of complete date hiding, because the user could change this standard routine. But this would be wilfulness, and we think, that in this context, data hiding has to prevent from unintentional, not wilful errors.

A complete example, using the eas_manager, is given later.

SIMULATION_CONTROL

We see, similar to SLAM, the necessity to allow all forms of combined modelling techniques: continous, discrete event, activity scanning and process interaction approach. Furthermore, a complete interaction between all these model types and the transaction_flow manager shall be possible. This can be managed easily, because a transaction flow model can be seen as a specific form of a combined discrete event, activity scanning and process interaction model.

There are two forms, how this simulation control can be implemented: in a tasking or in a non tasking form.

Tasking:

(1) Process interaction approach can be implemented in a simple and natural way, using entry calls like

```
   hold(15.0);
```

to block a running process.

(2) If Ada is implemented on a computer with multiprocessor architecture, where different tasks execute on different processors, one can implement event procedures as tasks, with the advantage, that the simulation program runs faster than on a single processor computer, because different events can execute at the same (real!) time, but at possible different model times, provided, that the synchronization and the access to common data resources are managed correct.

Advantages for the user:

ad (1): the feasibility of a straight foreward formulation of a process interaction model

ad (2): faster execution; but possibly a deeper analysis of the model is necessary as mentioned above.

Advantages and disadvantages for the package designer:

It is obvious, that a package, consisting of task and task types is more complicated to design and to write than one without tasks.

On the other hand, with such a powerful language like Ada, a multiprocessor discrete event simulation is certainly much more easily to implement, than without such a help (04).

We have implemented for the present non tasking versions of the simulation control package, restricted to discrete event and combined discrete event, activity scanning and transaction flow approach. A simple sketch for a processoriented, discrete-system simulation control package in tasking form is given in Bryant (03).

We give here only a synopsis of the discrete event part of the simulation package. It works similar to SLAM: Events are implemented as user-written procedures, where the event-name acts as procedure name. An enumeration type 'event_name' is introduced too. The enumeration litterals have the same identifiers as the event procedures (and therefore are overloaded). At each event time, the control is passed to a user-written procedure 'user_event', where in form of a case statement the transfer to the appropriate event procedure is performed. An entity can be passed together with the event notice, when the scheduler is called, and this entity is to the event procedure's disposal, as soon as the 'user_event' procedure is called at the event time.

The package

The user passes the procedure 'user_event' and his types 'entity' and 'event_name' to the generic package. Then four more user written procedures are passed, which render possible to set initial conditions and to gain control at simulation end:

In 'system_init' global variables can be set like time to start and stop the simulation; 'system_end' is called after each simulation run and can be used to perform multiple runs. (These procedures correspond to the SLAM Control Statements.)

'user_init' is given control at simulation start; 'user_end' is called at the end of the simulation. (These procedures correspond to SLAM INTLC and OTPUT.)

The types event_time, a floating point real type to denote event times, and event_notice, a limited private type, are introduced through the visible part of the package. This later type is the particular entity type of the simulation control package, and these event notices are queued in the 'event_calendar'. User variables of the type event_notice are used to mark specific events in the event calendar (in Simscript ... "CALLED" ...; in Simpas the named clause), but can only be used as parameters for subprogram calls. The basic design for the scheduling purposes is similar to Simscript (and Simpas).

The scheduling of a new event notice is possible in a named and in an anonymous form. Additionally an enumeration type 'kind_of_scheduling' with litterals (at_time, now, delayed, before, after) is introduced to denote, how the event time is indicated. Events can be rescheduled in a similar form. (But no "cancel" is needed, if just the time of an event has to be changed; it can be done directly.) To eliminate an event_notice, the procedure 'destroy' is provided. To get all information for a certain event notice, procedure event_info can be used.

We provide overloaded procedures with default parameters, which render possible to write simple, self documenting scheduling calls.

Some examples:

```
type event_name is (customer_arrival,
       end_of_service, bank_robbery);
a_customer, gang: entity;
bank_closing: event_notice;
--
--
schedule (customer_arrival, now);
schedule (end_of_service, at_time, 150.0,
         a_customer);
reschedule (bank_closing, delayed, 15.0);
schedule (bank_robbery, after, bank_closing, gang);
```

There are subroutines to control the simulation run:

set_start_and_stop:
  to initiate simulation begin and end;
trace_start_and_stop: to set the begin and the
  end of a trace;
stop_simulation: to end a simulation run;

time_now and time_next: functions returning these
  informations. (Considering the principle of data
  hiding, the user has no direct access to the
  simulation clock!)

A simple call of the procedure start_simulation, performed in the user's main procedure, starts the simultion run.

```
-----------------------------------------------
-- package declaration simulation_control
-----------------------------------------------
--
with eas_manager;
generic
   type entity is private;
   type event_name is <>;
   with procedure user_event (p: in event_name;
                  e: in out entity) is <>;
   with procedure system_init       is <>;
   with procedure system_end        is <>;
   with procedure user_init         is <>;
   with procedure user_end          is <>;

package simulation_control is

   type kind_of_scheduling is
      (at_time, now, delayed, before, after);
   subtype event_time is float;
   type event_notice is limited private;

   function time_now  return event_time;
   function time_next return event_time;

   procedure start_simulation;
   procedure stop_simulation;
   procedure set_start_and_stop
                  (t1,t2: event_time);
   procedure trace_start_and_stop
                  (t1,t2: event_time);
   procedure event_info (event: in event_notice;
                      whenn: out event_time;
                      whatt: in out event_name;
                      withh: in out entity);
   procedure schedule
              (whatt: in event_name;
               how  : in kind_of_scheduling;
               whenn: in event_time := time_now();
               withh: in entity:= null);
  procedure schedule
              (whatt: in event_name;
               how  : in kind_of_scheduling;
               whenn: in event_notice;
               withh: in entity:= null);
  procedure schedule
              (whatt: in event_name;
               how  : in kind_of_scheduling;
               whenn: in event_time;
              called: out event_notice;
               withh: in entity:= null);

 procedure schedule
              (whatt: in event_name;
               how  : in kind_of_scheduling;
               whenn: in event_notice;
              called: out event_notice;
               withh: in entity:= null);

--
-- similar for reschedule, cancel and destroy
--

private

-- private declarations;
-- instantation of the eas_manager;
-- (event_notice is a particular entity type of
-- the simulation control manager)

end simulation_control;

-----------------------------------------------
-- end package declaration simulation_control
-----------------------------------------------
```

INTERACTIVE MODEL_DESIGN

We think, it is a big advantage, if a simulation model can be designed, updated and tested interactively. An absolute necessity for such a project is a very powerful screen and printer input output supporting package, which we have finished already. A further advantage can be noticed, if the design and update process is supported by a syntax and semantic check of the target language, what we intend to do as well.

We can only demonstrate here few aspects of these packages and we think, the best way is to do it by means of an example (given by Pritsker (10)).

Example: Drive-in bank with jockeying

A drive-in bank has two windows; customer arrivals are exponentially distributed (mean:=0.5); service time is normally distributed (mean:=1.0; std:=0.3); customers prefer the shortest lane or lane one, if neither teller is busy or if the waiting lanes are equal. A customer can change lanes, if he is the last one in his lane and if there is a difference of two customers between the two lanes.

Limitations:

A maximum of eigth cars in the system; if the system is full, an arriving customer balks and is lost to the system.

Initial conditions:

Both tellers are busy; two customer in each queue; first customer arrives at 0.1 time units.

Goal:

1. Teller utilization
2. Time-integrated average number of customers
3. Time between departures from the windows
4. Average time a customer is in the system
5. Average number of customer in each queue

6. Percent of arriving customers who balk
7. Number of times cars jockey

The system is to be simulated for 1000 time units, a trace is to be obtained for the first 10 time units, from 500.0 to 510.0 .

End of example description.

It is not easy to reproduce an interactive dialogue in printed form. We show the sequence of screen forms, first only the empty form without the user input, underneath the word "input:" and the lines, where an input is made. If a form provides only a choice, the form is given directly with the marked fields. The dialogue looks quite lengthy, but this is misleading: Beside the two event procedures and the entity_init procedure the input can be performed in less than five minutes.

```
-----------------------------------------------------
-        Ada Simulation Support Environment         -
-                 *** model_design ***              -
  vers.:  1                                         -
  rel.:   3                                         -
                                                    -
  date:  8/ 8/ 1982                                 -
                                                    -
  model name (short form): ............             -
-                                                   -
-----------------------------------------------------
input:
  model name (short form): jockeying
```

```
-----------------------------------------------------
-        Ada Simulation Support Environment         -
-                 *** model_design ***              -
  vers.:  1                                         -
  rel.:   3                                         -
                                                    -
  date:  8/ 8/ 1982                                 -
                                                    -
  new model:         last update: date: 00/00/00    -
                                  time: 00.00.00    -
  model vers.:   1                                  -
  by          : .................................   -
  model name  : .................................   -
-                                                   -
-----------------------------------------------------
input:
  by          : Heimo H. Adelsberger
  model name  : Drive-in bank with jockeying
```

```
-----------------------------------------------------
-        Ada Simulation Support Environment         -
-                 *** model_design ***              -
                                                    -
Give your entity names:                             -
                                                    -
   type entity_name is                              -
                                                    -
-                                                   -
-----------------------------------------------------
input:
   type entity_name is
      (customer);
```

```
-----------------------------------------------------
-        Ada Simulation Support Environment         -
-                 *** model_design ***              -
                                                    -
Give your queue names:                              -
                                                    -
   type queue_name is                               -
                                                    -
-                                                   -
-----------------------------------------------------
input:
   type queue_name is
      (teller_one, teller_two);
```

```
-----------------------------------------------------
-        Ada Simulation Support Environment         -
-                 *** model_design ***              -
                                                    -
Give your event names:                              -
                                                    -
   type event_name is                               -
                                                    -
                                                    -
-                                                   -
-----------------------------------------------------
input:
   type event_name is
      (customer_arrival, end_of_service);
```

```
-----------------------------------------------------
-        Ada Simulation Support Environment         -
-                 *** model_design ***              -
                                                    -
Give your attributes:                               -
                                                    -
Entity: customer                                    -
                                                    -
                                                    -
-                                                   -
-----------------------------------------------------
input
   arrival_time: event_time;
   teller:       queue_name;
```

```
-----------------------------------------------------
-        Ada Simulation Support Environment         -
-                 *** model_design ***              -
                                                    -
You can change the print form:                      -
                                                    -
//mask 101                                          -
   arrival_time: <          >                       -
   teller:       <          >                       -
                                                    -
-                                                   -
-----------------------------------------------------
input: no input
```

```
-----------------------------------------------------
-        Ada Simulation Support Environment         -
-                 *** model_design ***              -
                                                    -
Give your global declarations:                      -
                                                    -
-                                                   -
-----------------------------------------------------
input:
number_of_customers,
number_of_balks,
number_of_jockeys: integer := 0;
percent_of_balks:  float;
last_departure    : event_time;
```

```
----------------------------------------------------
-       Ada Simulation Support Environment         -
-                *** model_design ***              -
                                                   -
You can change the print form:                     -
                                                   -
//mask 102                                         -
number_of_customers        <        >              -
number_of_balks            <        >              -
number_of_jockeys          <        >              -
percent_of_balks           <        >              -
last_departure             <        >              -
                                                   -
-                                                  -
----------------------------------------------------
input: no input
----------------------------------------------------
-       Ada Simulation Support Environment         -
-                *** model_design ***              -
                                                   -
Give your variables for automatic                  -
collection of statistics:                          -
                                                   -
Value weighted:                                    -
                                                   -
called                                             -
                                                   -
-                                                  -
----------------------------------------------------
input:
   time_in_system: event_time;
called
   "time in system"

input:
   time_between_departures: event_time;
called
   "time between departure"

----------------------------------------------------
-       Ada Simulation Support Environment         -
-                *** model_design ***              -
                                                   -
Give your variables for automatic                  -
collection of statistics:                          -
                                                   -
Time weighted:                                     -
                                                   -
called                                             -
                                                   -
-                                                  -
----------------------------------------------------
input:
   busy: array (queue_name) of boolean;
called
   "teller utilization"

----------------------------------------------------
-       Ada Simulation Support Environment         -
-                *** model_design ***              -
                                                   -
Give your procedures:                              -
                                                   -
procedure entity_init:                             -
-                                                  -
----------------------------------------------------
```

```
input:
procedure entity_init (a_customer: out entity) is
begin

   a_customer.arrival_time:= time_now();

   if busy(teller_one) and busy(teller_two) then
--
-- both tellers are busy;
-- place customer in shortest lane
--
      if queue_size(teller_one) <=
            queue_size(teller_two)  then
         a_customer.teller:= teller_one;
         queue_insert (teller_one, a_customer);
      else
         a_customer.teller:= teller_two;
         queue_insert (teller_two, a_customer);
      end if;

   elsif not busy(teller_one) then
--
-- teller one is free
--
      busy(teller_one):= true;
      a_customer.teller:= teller_one;
      schedule (end_of_service, delayed,
            normal(1.0,0.3,2), a_customer);
   else
--
-- teller two is free
--
      busy(teller_two):= true;
      a_customer.teller:= teller_two;
      schedule (end_of_service, delayed,
            normal(1.0,0.3,2), a_customer);
   end if;

end entity_init;

----------------------------------------------------
-       Ada Simulation Support Environment         -
-                *** model_design ***              -
                                                   -
Give your procedures                               -
                                                   -
procedure user_init:                               -
                                                   -
                                                   -
-                                                  -
----------------------------------------------------
input

procedure user_init is
   a_customer: entity;
begin
--
-- creates two customers for each teller; they
-- will be inserted automatically in the queues
-- via procedure entity_init.
-- schedules a customer arrival for 0.1
--
   for i in queue_name'first .. queue_name'last loop
      busy(i):= false;
      entity_create (a_customer, customer);
      entity_create (a_customer, customer);
   end loop;
   schedule (customer_arrival,at_time,0.1);
end user_init;
```

```
---------------------------------------------------
-        Ada Simulation Support Environment       -
-               *** model_design ***              -
                                                  -
Give your procedures.                             -
                                                  -
procedure user_event:                             -
                                                  -
-                                                 -
---------------------------------------------------
input:

procedure user_event
   (p: event_name; e: in out entity) is
begin
   case p is
      when customer_arrival => customer_arrival(e);
      when end_of_service   => end_of_service(e);
   end case;
end user_event;

---------------------------------------------------
-        Ada Simulation Support Environment       -
-               *** model_design ***              -
                                                  -
Give your event procedures                        -
                                                  -
Event procedure: custome_arrival                  -
                                                  -
                                                  -
                                                  -
-                                                 -
---------------------------------------------------
input

procedure customer_arrival
          (a_customer: in out entity)  is
begin
--
-- cause next arrival and increments
-- number_of_customers
--
   schedule (customer_arrival, delayed,
                 exponential(0.5,1));

   number_of_customers:= number_of_customers+1;

   if queue_size(teller_one)+queue_size(teller_two)
      >= 6 then
--
-- if system full, then balk
--
      number_of_balks:= number_of_balks+1;
   else
--
-- look entity_init for initialization of attributes
-- and choice of lane or teller
--
      entity_create (a_customer, customer);
   end if;
   percent_of_balks:= 100.0*float(number_of_balks)/
                      float(number_of_customers);
end customer_arrival;
---------------------------------------------------
-        Ada Simulation Support Environment       -
-               *** model_design ***              -
                                                  -
Give your event procedures                        -
                                                  -
Event procedure: end_of_service                   -
                                                  -
-                                                 -
---------------------------------------------------
```

```
input

procedure end_of_service
          (a_customer: in out entity) is
   servicing_teller, other_teller: queue_name;
begin
   time_in_system:=
      time_now() - a_customer.arrival_time;
   time_between_departures:=
      time_now() - last_departure;
   last_departure:= time_now();
--
-- set 'servicing_teller' to teller just ending
-- service, 'other_teller' to other teller
--
   servicing_teller:= a_customer.teller;

   if servicing_teller = teller_one  then
      other_teller:= teller_two;
   else
      other_teller:=teller_one;
   end if;
--
-- test number of waiting customers
--
   if queue_size(servicing_teller) >= 0 then
--
-- lane is occupied, therefore process the first
-- customer in this lane
--
        queue_remove(servicing_teller, a_customer);
        a_customer.teller:= servicing_teller;
        schedule (end_of_service, delayed,
                   normal(1.0,0.3,2), a_customer);

        if queue_size(other_teller) >=
          queue_size(servicing_teller) +2 then
--
-- if the number in the other lane exceeds the
-- number in this lane by two, then jockey
-- last customer from other lane
--
          queue_remove (other_teller, a_customer,
                        last_member);
          queue_insert (servicing_teller,a_customer);
        end if;

   elsif queue_size(other_teller) >= 0 then
--
-- servicing lane is empty and other lane is
-- occupied, therefore jockey from other lane
--
      queue_remove (other_teller, a_customer,
                    last_member);
      a_customer.teller:= servicing_teller;
      schedule (end_of_service, delayed,
                 normal(1.0,0.3,2), a_customer);
      number_of_jockeys:= number_of_jockeys+1;

   else
--
-- both lanes are empty, therefore set
-- servicing teller idle
--
      busy(servicing_teller):= false;

   end if;

end end_of_service;
```

INTERACTIVE MODEL_VERIFICATION

The simulation model is under control of the model_verification package during the whole simulation run. At each time the trace feature can be switched on and off. When being in the trace mode, all variables can be displayed and changed, all reports on queues and entities can be called, queue characteristics can be changed, entities can be created, destroyed, inserted and removed from queues. We just give a glimpse on some few screen forms:

```
-------------------------------------------------------
-        Ada Simulation Support Environment           -
-            *** model_verification ***               -
                                                      -
System initialization                                 -
                                                      -
Simulation:  begin   ......  end   ......             -
                                                      -
Trace:     < > on       < > off                       -
             begin   ......  end   ......             -
                                                      -
-                                                     -
-------------------------------------------------------
input
Simulation:  begin   0.0     end   1000.0

Trace:     <x> on       < > off
             begin   500.0   end   510.0

-------------------------------------------------------
-        Ada Simulation Support Environment           -
-            *** model_verification ***               -
                                                      -
queue_name: teller_one:                               -
                                                      -
queue ranking ?                                       -
< > FIFO   < > LIFO   < > HVFO   < > LVFO             -
(default: FIFO)                                       -
                                                      -
queue capacity ......                                 -
(default: integer'last)                               -
                                                      -
-                                                     -
-------------------------------------------------------
input:
queue capacity 3

////// similar for teller two /////

-------------------------------------------------------
-        Ada Simulation Support Environment           -
-            *** model_verification ***               -
                                                      -
                  Model output       Trace output     -
                                                      -
Screen               < >                <x>           -
Printer              <x>                < >           -
File                 <       >          <             -
                                                      -
-                                                     -
-------------------------------------------------------
```

```
-------------------------------------------------------
-        Ada Simulation Support Environment           -
-            *** model_verification ***               -
                                                      -
At what action shall be traced:                       -
                                                      -
< > eas_manager action                                -
<x> simulation_control action                         -
                                                      -
system                                                -
   <x>         global variables                       -
   < >         watched variables: value weighted      -
   < >         watched variables: time  weighted      -
                                                      -
eas_manager                                           -
   < >         all actions on entities                -
   < >         all actions on queues                  -
   <x>         global informations entities           -
   <x>         global informations queues             -
   < >         involved entity                        -
                                                      -
simulation_control                                    -
   <x>         events                                 -
   <x>         involved entity                        -
-                                                     -
-------------------------------------------------------
```

```
-------------------------------------------------------
-        Ada Simulation Support Environment           -
-            *** model_verification ***               -
                                                      -
time now:      500.101      time next:      500.103   -
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
*** event notice:                                     -
this event:                 next  event:              -
system time:   500.101      at:             500.103   -
event: end_of_service       event: customer_arrival   -
entity: customer            entity: null              -
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
*** entity:  entity_name: customer                    -
arrival_time        499.671                           -
teller              teller_one                        -
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
*** global variables                                  -
number_of_customers              1048                 -
number_of_balks                    86                 -
number_of_jockeys                 135                 -
percent_of balks                8.206                 -
last_departure                499.728                 -
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
*** eas_manager                                       -
                                                      -
*queues:     curr. length    max      average         -
teller_one              3      3       1.3812         -
teller_two              2      3       1.1774         -
                                                      -
*entities:         curent    total                    -
customer                7    1048                     -
total                   7    1048                     -
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
-                                                     -
-------------------------------------------------------
```

At the end, similar to the trace mode, all reports can be called and the values of all variables can be displayed:

```
---------------------------------------------------
-         Ada Simulation Support Environment      -
-             *** model_verification ***          -
time now:     1000.000     time next:     ******** -
- - - - - - - - - - - - - - - - - - - - - - - - - -
******* end of simulation reached ********         -
                                                   -
Indicate your reports                              -
                                                   -
system                                             -
   <x>          global variables                   -
   <x>          watched variables: value weighted  -
   <x>          watched variables: time  weighted  -
                                                   -
eas_manager                                        -
   <x>          global informations entities       -
   <x>          global informations queues         -
                                                   -
simulation_control                                 -
   <x>          events                             -
-                                                  -
---------------------------------------------------
```

```
---------------------------------------------------
-         Ada Simulation Support Environment      -
-             *** model_verification ***          -
                                                   -
Reports:                                           -
                                                   -
- - - - - - - - - - - - - - - - - - - - - - - - - -
*** global variables                               -
number_of_customers            2016                -
number_of_balks                 179                -
number_of_jockeys               257                -
percent_of_balks              8.927                -
last_departure              999.204                -
- - - - - - - - - - - - - - - - - - - - - - - - - -
-                                                  -
---------------------------------------------------
```

TRANSACTION_FLOW MANAGER

We see transaction flow models as a specific form of a combined discrete event, activity scanning and process oriented approach. Furthermore, we distinguish exactly between what can be represented in the form of a network and what exists besides the network.

This eliminates unnaturally constructs like "supervisors", clock transactions, and immaterial entities, which e.g. rotate only to open and close gates. We think, that networks shall be more isomorph to the real system and not overloaded with artificial constructs.

Furthermore we model networks as an abstract data type, which is equal for networks as a whole, subnetworks and nodes. This concept renders possible a clear and simple design, because it enables to transfer useful programming techniques to the design of networks like:

modularity (subnetworks like subprograms)
top-down design
generics (generic subnetworks)

The number of "atomar" nodes is reduced in comparison to GPSS and SLAM, more complex node types are introduced as generic composite nodes, whereby the standard nodes of this type are provided by the network manager itself, but which nodes can be declared by the user too.

The network part of the ASSE is well supported by the interactive model design and verification packages. It renders possible an interactive, graphical design and there is no need anymore for the user to translate the graphic form of a network into a statement form.

The interactive model verification package supports the network part too. It is possible to introduce new nodes, to display and change all nodes and to follow a graphical trace of the network on the screen.

We know, we have given here a very incomplete description of the ASSE. The reader, who is interested in more details, is refered to (01), which shall soon appear in print.

BIBLIOGRAPY

/01/ Adelsberger, H.H., "Ada Simulation Support Environment - a Report", forthcoming.
/02/ Brayant, R.M., "SIMPAS -- A Simulation Language Based on Pascal", Proceedings of the 1980 Winte Simulation Conference, 25 - 40, 1980.
/03/ Brayant, R.M., "Discrete System Simulation with Ada", Computer Sciences Department Technical Report # 458, University of Wisconsin--Madison (November 1981).
/04/ Comfort, J.C., and Miller, A., "The Simulation of a Pipelined Event Set Processor", 1981 Winte Simulation Conference Proceedings, Atlanta 1981
/05/ Ledgard, H.F., and Singer, A., "Scaling Down Ada (Or Towards A Standard Ada Subset)", Comm. ACM, 25. 2. (Feb. 1982) 121-125.
/06/ Kiviat, P.J., Villanueva, R., and Markowitz, H.M., The SIMSCRIPT II Programming Language, CACI, Los Angeles, 1975.
/07/ Pritsker, A.A.B., The GASP IV Simulation Language, John Wiley & Sons, 1974.
/08/ Pritsker, A.A.B., Modeling and Analysis Using Q-GERT Networks, John Wiley & Sons, 1977.
/09/ Pritsker, A.A.B., Introduction to Simulation and SLAM, John Wiley & Sons, 1979.
/10/ Rubin, J., "Imbedding GPSS in a General Purpose Programming Language", 1981 Winter Simulation Conference Proceedings, Atlanta, 1981.
/11/ Schriber, T.J., Simulation using GPSS, John Wiley & Sons, 1974.
/12/ 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.
/13/ U.S. Department of Defense, "Reference Manual for the Ada Programming Language", Military Standard MIL-STD-1815, Naval Publications and Forms Center, Philadelphia, 1980.
/14/ U.S. Department of Defense, "Requirements for Ada Programming Support Environments - 'Stoneman'", 1980.