THE PROCESS VIEW OF SIMULATION IN ADA*

Greg Lomow and Brian Unger
University of Calgary
Calgary, Alberta, Canada

Abstract

Previously, the process view of simulation, which represents a model as a set of competing and cooperating entities, had been most successfully implemented in the general purpose language SIMULA. This paper describes such a system which is currently being implemented in ADA. ADA's suitability as the base language for such a package is first discussed followed by a description of the facilities offered in SAMOA† (Simulation and Modelling on Ada). SAMOA is a fully integrated, general purpose, discrete event simulation package whose features are illustrated through the use of examples.

The process view is a simple and natural way to define discrete event simulation models. This follows from the fact that a broad range of natural human-made systems can be easily described in terms of interacting concurrent processes [2], [7], [9], [20]. Models defined using this process approach are very similar to natural language descriptions of the actual systems being modelled. In fact, the process view has been used in the DELTA Language to structure system descriptions which are intended solely as aids for human communication about systems [8].

The SIMULA [5] programming language provided an early implementation of the process approach to simulation. This general purpose language was designed to enable the definition of abstract types in terms of both primitive types and other abstract types. These abstract types can include actions so that instances, or objects, of these types are pseudo concurrent autonomous processes. A process can invoke actions within other processes and thus they are able to interact. Unfortunately, this process model has not been widely adopted in the simulation community, particularly in North America. The impact of SIMULA has been much more pronounced in the computer science community. SIMULA provided one of the earliest models of communicating concurrent processes, an idea which has influenced the development of operating systems such as UNIX [10] and languages such as ADA [1], [12].

Many other languages and packages for discrete event simulation have been defined. The general purpose languages most widely used in North America have been GPSS [18], SIMSCRIPT [11], and GASP [15], [16]. GPSS is essentially a process based approach embedded in a Fortran like language. SIMSCRIPT and GASP are event based, although a process view has been added to SIMSCRIPT [17]. None of these approaches has the extensible expressive power of SIMULA. Franta and others [7], [9] have shown how SIMSCRIPT and GASP constructs can be implemented in SIMULA, while Birtwistle has recently implemented a GPSS like SIMULA extension called DEMOS [2]. Specifically, DEMOS offers the simplicity and report generation advantages of GPSS embedded in a powerful general purpose language. Furthermore, SIMULA and its process approach to simulation is applicable to a wide range of special purpose modelling problems, e.g. the OASIS simulation package [21], [22], [23] for the simulation of multicomputer systems and software. Despite the advantages of these SIMULA extensions GPSS, SIMSCRIPT, GASP, and related approaches still predominate. This situation was probably due to SIMULA's limited availability, especially on small machines. Recently, however, a portable SIMULA compiler system [13], [14] has been developed and as a result SIMULA will become available on a wider variety of machines, including microcomputers. The relatively large SIMULA run time system overhead of 15K to 30K memory words is also rapidly becoming negligible. Thus, the major disadvantages of SIMULA may soon disappear.

*Ada is a trade mark of the U.S. Dept. of Defense (Ada Joint Program Office).

†The name SAMOA is Graham Birtwistle's invention following his DEMOS (Discrete Event Modelling on Simula). The design of SAMOA is heavily based on DEMOS [2].

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

82CH1844-0/82/0000-0077 $00.75 © 1982 IEEE

ADA [12], the U.S. Department of Defense's recently developed programming language represents another alternative to SIMULA as a base language for the process view of discrete event modelling. This is because ADA offers some of the same features introduced by SIMULA 15 years ago (e.g. concurrent processes, definition of abstract data types, and language extensibility) while enjoying potentially wider availability. The SAMOA (Simulation And Modelling On Ada) package described herein is an initial attempt to apply ADA constructs to the development of a general purpose discrete event simulation package. This package is a fully integrated approach which employs the process view of simulation and augments it by providing automatic statistics collection, report generation, and tracing facilities. SAMOA's other goals include 1) limiting, to a PASCAL size subset, the amount of ADA the user is required to know, and 2) allowing the user access to the full power of ADA when it is necessary.

This paper will discuss ADA's suitability as the base language for a discrete event simulation language, how models are represented in SAMOA, and what modelling facilities are provided by SAMOA. Also, an example simulation will be defined using this package. This model will be compared with a corresponding GPSS model and the strengths and weaknesses of each implementation will be discussed.

Some of the fundamental advantages of implementing any package in ADA are derived from ADA's design goals as outlined in the Stoneman report [19]. The most important of these goals for SAMOA are:

1) Programming Support - the goal of an ADA environment is to support a basic set of common programming tools. These tools are to provide continuing support in the development and maintenance of ADA programs. This support ranges from the coding phase through the debugging and software maintenance phases. Tools to be provided include editors, debugging aids, and a set of project management and software configuration tools.

2) Programmer Portability - this goal ensures that a programmer who moves from one ADA environment to another will be able to work in the new environment with minimal retraining. The ADA language and the basic set of programming tools found in all environments will be consistent.

3) Program Portability - this goal is related to programmer portability and it ensures that programs which are moved from one ADA environment to another will compile and run without major modifications. This is important because it allows the SAMOA package to be portable and ensures that any special purpose, user defined simulation tools will also be portable.

Besides these design goals of ADA, SAMOA benefits from features of the ADA language itself. ADA's Pascal based syntax has language features which are intended to encourage modular program design and the separation of a program's specification from its implementation. This separation can be used by the ADA programmer, as it is in SAMOA, to provide and enforce a level of abstraction between a program's user and its implementation details. The most important of these language features are strong typing, data abstraction, generic units, libraries, subunits, and tasking.

Strong typing "... ensures that each object has a clearly defined set of values and prevents confusion between logically distinct concepts. As a consequence many errors are detected by the compiler which in other languages would have led to an executable but incorrect program" [1].

Data abstraction allows the definition of modular program units each of which encapsulates the data structures and operations of a single abstract data type (e.g. a stack with the operations 'pop' and 'push').

ADA's basic unit of concurrency is the 'task', which allows a problem to be solved as a series of parallel activities. It is this construct that SAMOA uses to represent the actions of each active component, or entity, in a simulation. Two points should be noted about the use of tasks in conjunction with SAMOA. First, even though tasks running in a multi-processor ADA environment can run concurrently they have been restricted in SAMOA to run as coroutines. This is so that the effect of a SAMOA simulation can be guaranteed to be deterministic, that is, so that the results are reproducable. Second, even though tasking is one of the most complicated features of ADA, the SAMOA package has been designed so that when describing the actions of each entity, or active component, in a simulation, the user is not required to use any of ADA's tasking facilities. Instead, a set of simulation routines is provided by SAMOA (to be described later) which provide a level of abstraction between the user and the implementation details.

Generic units allow the definition of a program unit to be independent of the type of value which it is to manipulate. This allows a single piece of code to serve as a template for the creation of program units which perform the same operations but perform them on different data types. This allows us to write a general purpose programming tool at one level but delay its full definition until the point where the user creates it.

Libraries and subunits support the design and implementation of large systems. ADA allows predefined libraries of program units to be constructed and accessed so that a system can be built in a bottom-up fashion. Alternatively, the use of subunits allows the definition of program units to be delayed so that a system can be specified and built top-down.

It should be noted that the SAMOA user is not required to have an extensive knowledge of generic units, libraries, or subunits. They are mentioned here because these features are available to the SAMOA user, and their use can simplify the design and development of medium to large scale software projects. The fact that many simulation projects fall into this size range strengthens ADA's potential as the base language for a discrete event simulation package.

These ADA language constructs are now used to describe the SAMOA package. This discussion can be broken down into several parts. First the central component of SAMOA, the entity, is defined. Secondly we examine the set of routines provided by SAMOA to manipulate these entities (e.g. hold( ) cancel( )). The third part focuses on the synchronisation operations which allow entities to coordinate their activites in up to four different ways: 1) competing for limited resources, 2) establishing producer/consumer relationships, 3) waiting for other entities to provide services, and 4) waiting until arbitrary conditions are satisfied. And in the fourth part we survey additional features of SAMOA such as automatic statistics collection, random number generation, automatic tracing and report generation. Throughout this discussion features of SAMOA will be illustrated via example.

The principle example used to illustrate SAMOA is the simulation of an open-stack policy library. "At such a library, anyone wanting a book must present a checkout slip to a clerk working behind the checkout desk. The clerk then goes into the stacks to find the book and returns to the desk with it. The rest of the checkout procedure then takes place, and the person leaves with the book. If more than one person is waiting for service, a clerk often economizes on the time required to travel between the checkout desk and stacks by picking up checkout slips from more than one waiting person. Because the number of books a clerk can conveniently carry is limited, however, the number of slips a clerk is willing to pick up at any one time is also limited" [18]. Furthermore: each person requests only one book; the requests are satisfied on a first-come, first-serve basis; if more than one clerk is available and there is more than one slip then the clerks do not divide the work evenly but rather the first clerk takes as many slips as possible before the second clerk can take any; and a clerk will take a maximum of four slips. Additionally, the example specifies 1) various random number distributions concerning how long it takes clerks to perform their actions, and 2) that the simulation is to run until 100 customers have been completely served. This example is drawn from Schriber [18]. Its GPSS solution is reproduced in Figure 5 and an initial definition of a process style solution is shown in Figure 1.

Figure 1
Process Style Solution to the Library Example

```
entity customer is
begin
  -- get checkout slip for one book
  -- wait in line until book is checked out
  -- leave library
end customer;


entity clerk is
begin
  -- wait until a customer needs service
  -- get up to four book slips
  -- get each book
  -- return to check out counter
  -- checkout each customer in order
  -- repeat
end clerk;
```

When applying the process view to the library example we can view the customer/clerk relationship in one of two ways. The first view is as a producer/consumer relationship in which the customers produce checkout slips that the clerks consume, and the clerks produce books that the customers consume. The second view would be as cooperating entities, where the customers wait while the clerks perform services for them. A solution based on this second approach is outlined in Figure 1, this example will be fleshed out as we discuss each relevant feature of SAMOA.

ENTITY REPRESENTATION

In SAMOA each active simulation component is represented as an entity. An entity is defined in two parts. The first part, the 'entity_record', is defined within SAMOA and contains data structures relevant to all entities. SAMOA uses the information in these entity_records to implement the various entity manipulation facilities. The entity_record is, in effect, that portion of an entity that SAMOA assumes is present in all cases for compilation and execution. The second part of an entity is user defined and it consists of the following two structures: 1) the entity's globally accessible attributes or data structures, and 2) the entity's set of actions.

The reason that the user portion of an entity must be defined in two parts is that ADA does not allow a task to directly access the data structures inside of another task. Because the activities of an entity are defined within task objects the only way to allow an entity to access status information about another entity is to place it in a separate structure. An example of this would be an 'entity dumptruck' where we need to know both its total capacity and its remaining capacity. If these values were internal to the task which represents a dumptruck's activities then they could not be inspected from other entities, therefore we place them in a separate structure. A refinement of the library problem taking this basic entity representation into account is shown in Figure 2.

The activities of an entity are defined as an ADA 'task type'. The task type is used so that many instances of that entity type can be created. The activities of each different entity type, or each task type, are specified in the 'task body' definition. A particular entity type can have its activities described within the task body using three kinds of statements:

1) Basic ADA statements. This includes such constructs as loop statements, case statements, assignment statements, and if statements. It would also include use of arrays, records, strings, and simple variables.

2) The routines provided in SAMOA (and described in the next two sections) offer a flexible set of primitives for creating and manipulating entities.

3) The entity actions could also be represented by user defined routines which are specific to the model which is under consideration (e.g. an 'unload' routine for a simulation involving dumptrucks).

Figure 2
Library Example Outlined in ADA

```
procedure library is

    type customer_rec is record
      null;  -- No global data structures
    end record;

    task type customer;

    task body customer is
    begin
    -- get checkout slip for one book
    -- wait in line until book is checked
       out
    -- leave library
    end customer;


    type clerk_rec is record
      null;  -- No global data structures
    end record;

    task type clerk;

    task body clerk is
    begin
      loop
        -- wait until a customer needs
           service
        -- get up to four book slips
        -- get each book
        -- return to check out counter
        -- checkout each customer in order
      end loop;
    end clerk;

begin
  -- Start of Main Body of Simulation

  -- create entities
  -- wait for 100 customers to be served
end library;
```

At any point in a simulation there may be some entities which are active and others which are suspended (either because they have terminated or because they are awaiting the actions of other entities). All active entities are kept, in order of increasing 'event_time', on the 'events_list'. An entity's event_time is the simulation time at which its next activity is to take place. The entity at the head of the events_list has an event_time which is at least as low as any other active entity. This entity is said to be the 'current' entity and it is the entity whose actions are being performed. The events_list is continually being updated as each entity performs its actions (which may affect other entities). The main program of a simulation (in Figure 2 this is the main body of the procedure 'library') is also treated as an entity. This allows the main program to use the same routines and synchronisation devices which are provided for other entities.

Both portions (the system defined section and the user defined section) of an entity are linked together and the entire entity is accessible by an access value (ADA's term for a pointer or reference value) which is initialised and returned when the entity is created. This access value must then be passed to SAMOA routines which manipulate entities so that these routines can identify who they are operating on.

ENTITY MANIPULATION ROUTINES

To manipulate entities, SAMOA provides a set of simulation primitives. These routines can be classified into the following categories: 1) the creation of entities, 2) providing status information about a particular entity, 3) routines that allow an entity to manipulate the events_list by starting, stopping, or interrupting other entities, and 4) routines which allow an entity to suspend itself either to represent the passage of simulation time or to await the actions of other entities.

1) Creation of Entities

* PROCEDURE CREATE_ENTITY (entity_access; title);
  - this procedure creates all the portions of an entity, links them together, initialises any relevant values, and returns a single access value which points to the new entity. A string must be passed as a parameter to this procedure and the returned entity will have as its title that string concatenated with a two digit serial number. This title is then used in reports, trace listings and error messages.

2) Entity Status Routines

* FUNCTION ACTIVE (entity_access) RETURN BOOLEAN; - this function returns 'true' if the entity in question is 'active', that is, if it is currently on the events_list.

* FUNCTION PRIORITY (entity_access) RETURN INTEGER; - this function returns the relative priority of an entity. All queues in SAMOA are maintained in priority order and these priorities are determined and set by the user.

* FUNCTION TERMINATED (entity_access) RETURN BOOLEAN; - this function returns 'true' if the entity in question has run to completion and is therefore no longer eligible for execution.

3) Event List Manipulation Routines

* PROCEDURE SCHEDULE (entity_access; activate_time); - places the entity pointed to by entity_access onto the events_list. The entity's new event_time will be the current simulation time plus activate_time. If an entity is scheduled now (i.e. SCHEDULE (some_entity, now);), that entity will preempt the currently running entity.

* PROCEDURE CANCEL (entity_access); - removes an entity from the events_list.

* PROCEDURE INTERRUPT (entity_access; int_type); - interrupts an entity's current activity. Int_type is a user defined range of values which allow the interrupted entity to determine why it was interrupted and, therefore, what actions are appropriate.

4) Entity Suspension Routines

* PROCEDURE HOLD (hold_time); - is used to model the passage of simulation time while an entity completes an action. The amount of simulation time that will elapse before control is returned to the entity (barring interrupts) is determined by the value passed as hold_time.

* PROCEDURE PASSIVATE; - suspends an entity. An entity may wish to be suspended when it can no longer continue executing until another entity satisfies some requirement.

* PROCEDURE TERMINATE_ENTITY; - is called by an entity when it has finished its activities or there is no longer any need for it to continue with its actions. This procedure cleans up the entities data structures and ensures that it terminates gracefully.

With the addition of these entity manipulation routines we can continue our definition of a solution to the library problem. In Figure 3 we have added to the example calls to the above routines. Notice that the main simulation routine only creates the clerks and one customer, each subsequent customer is created and scheduled by its predecessor.

Figure 3
Library Example with Entity Manipulation Routines

```
procedure library is

    type customer_rec is record
      null;  -- No global data structures
    end record;

    task type customer;

    task body customer is
      next : entity_ref := null;
    begin
      create_entity (next, "Customer");
      schedule (next, 'sometime in the future');
      -- get checkout slip for one book
      -- wait in line until book is checked out
      terminate_entity;
    end customer;


    type clerk_rec is record
      null;  -- No global data structures
    end record;

    task type clerk;

    task body clerk is
    begin
      loop
        -- wait until a customer needs service
        -- get up to four book slips
        hold (travel_time);
        hold (get_books_time);
        hold (travel_time);
        -- checkout each customer in order
      end loop;
    end clerk;

  clerk    : entity_ref := null;
  customer : entity_ref := null;
  clerk_num: integer := 4;

begin
  for 1..clerk_num loop
    create_entity (clerk, "Clerk");
    schedule (clerk, 0.0);
  end loop;

  create_entity (customer, "Customer");
  schedule (customer, 0.0);

  -- wait until 100 customers have been served

  -- report statistics
  -- terminate simulation
end library;
```

ENTITY SYNCHRONISATION ROUTINES

Besides the entity manipulation routines, SAMOA provides another level for describing the actions and interactions of entities. This second level of abstraction is the entity synchronisation routines and it consists of four basic sets of operations for specifying entity communication and cooperation. These four sets of routines allow entities to 1) compete for limited resources, 2) establish producer/consumer relationships, 3) wait

while other entities perform services (a master/slave relationship), and 4) wait until arbitrary conditions are satisfied. Each of these four functions implies blocking an entity (i.e. suspending the execution or operation of an entity) if the resources or conditions it requires cannot be satisfied. Because of this each of these sets of operations is based on a set of queueing primitives.

The queueing facility for entities consists of a set of routines for creating queue objects, for manipulating entities on a queue, for reporting and resetting a queue's usage statistics, and for listing the entities currently waiting in a queue. Queues are implemented as two-way linked lists, and an entity is kept on a queue in priority order. The following is an outline of the queueing facilities:

* PROCEDURE CREATE_QUEUE (queue_access; title); - returns an access value that points to a new queue object. The queue object is initialised as empty and it uses the string passed to this procedure as its title for trace listings, statistics reports, and error messages.

* PROCEDURE INSERT (entity_access; queue_access); - places the entity denoted by entity_access into the queue denoted by queue_access. The entity is placed into the queue in priority order (priority is a value which is local to an entity). If the entity is already in another queue when this procedure is called it will be removed from that queue, and placed on this queue.

* PROCEDURE REMOVE (entity_access); - deletes the entity from any queue on which it is currently residing.

* FUNCTION LENGTH (queue_access) RETURN INTEGER; - this function returns the number of entities which are currently residing in this queue. The length of a queue is keep as a running count so this function call is not required to traverse the entire list to determine the queue's current length.

* Procedures REPORT ( ), RESET ( ), and LIST ( ) - this set of procedures allow the user to report a queue's current statistics, reset a queue's statistics, and list the entities currently in a queue.

Each set of synchronisation operations described below manages an implicit queue. This implicit queue is maintained on a priority basis and is used to determine which entity is next eligible for service. Also, each set of operations includes procedures such as REPORT ( ), RESET ( ), and LIST ( ), and functions such as LENGTH ( ), which are modelled after the routines defined above for queues.

The first set of synchronisation routines define the manipulation of a 'res'. A res represents a limited number of resources for which mutual exclusion must be provided. This mutual exclusion is implemented on a semaphore basis. That is, an entity must request a portion of a res before using it, and must also return that res when it has finished with it.

A res is created and returned by a call to the procedure

```
CREATE_RES (res_access; title; total_number);
```

The parameter 'title' is used for tracing and reporting purposes while 'total_number' defines the maximum number of this res which is available. An entity may request any portion of this total number by calling the procedure 'ACQUIRE (res_access; num_requested);'. If enough of the res is presently available to satisfy this request then the entity is allowed to continue after the res count is decremented by 'num_requested'. If the request cannot be satisfied then the entity is blocked on the res until 1) enough of the res is available, and 2) the entity is next in line for service (remember that the entities are queued for service on a priority basis). An entity returns portions of a res by calling 'RELEASE (res_access; num_returned);'. This call will increment the res count by 'num_returned' and, if enough resources are now available, will unblock the first entity waiting on this res. This process is illustrated by an example where a number of entities need exclusive access to a file while updating it. After the file res is created:

```
CREATE_RES (file_res, "File", 1);
```

Each entity must access the file as follows:

```
ACQUIRE (file_res, 1);
-- update file
RELEASE (file_res, 1);
```

In the period between the call to ACQUIRE ( ) and the call to RELEASE ( ) the entity will have exclusive access to the file.

The second set of synchronisation routines define how a 'bin' is manipulated. The bin facility allows the user to establish producer/consumer relationships between entities. In such a relationship one set of entities, the producers, make items available to a second set of entities, the consumers (and hence the name bin, which represents a storage bin into which producers place items and from which consumers retrieve items).

A bin is created and returned by a call to the procedure

```
CREATE_BIN (bin_access; title; initial_number);
```

The parameter 'title' is used for tracing and reporting purposes while 'initial_number' defines the number of items initially in this bin. A consumer may request items from a bin by calling the procedure 'TAKE (bin_access; num_requested);'. If enough items are presently available to satisfy this request then the consumer is allowed to continue after the bin count is decremented by 'num_requested". If the request cannot be satisfied then the consumer is blocked on the bin until

1) enough of the bin becomes available, and 2) the consumer is next in line for service (remember that the entities are queued for service on a priority basis). A producer entity places items into a bin by calling 'GIVE (bin_access; num_given);'. This call will increment the bin count by 'num_given' and will, if enough of the bin is now available, unblock the first entity waiting on this bin.

Whereas, the res and bin synchronisation operations can be termed entity - resource synchronisation, the third type is more appropriately labelled as entity - entity synchronisation. This third facility is implemented as a 'waitq' and it allows the user to model master/slave relationships between entities. This type of coordination device is useful when one set of entities (the masters) provides a set of services for a second set of entities (the slaves). To implement this "we arrange for one set of the entities to dominate and let it treat the other as a resource to be coopted, retained as a passive slave throughout the period of cooperation, and then be released for independent progress at the end of this period of cooperation" [2].

A waitq is created and returned by a call to the procedure

```
CREATE_WAITQ (waitq_access; title);
```

A waitq is implemented internally as a set of two queues, one for blocked masters and one for blocked slaves. A master entity acquires a slave by calling the function 'COOPT (waitq_access) RETURN entity_access;'. If a slave is available then it is returned to the master who is allowed to continue, otherwise it must wait until one becomes available. Slaves block themselves for masters by calling the procedure 'WAIT (waitq_access);'. This process is illustrated by an example where boy scouts (masters) do their good deed for the day by escorting other people (slaves) across a busy roadway. After the waitq is created:

```
CREATE_WAITQ (crosswalk, "Crosswalk");
```

People wait for a boy scout as follows:

```
WAIT (crosswalk);
```

and boy scouts acquire people as follows:

```
my_current_person := COOPT (crosswalk);
```

When boy scouts are finished providing their services to people they allow them to go on their way by scheduling them to start up at the current simulation time:

```
SCHEDULE (my_current_person, 0.0);
```

This relationship is also illustrated in the library example in Figure 4 where the customers are the slaves and the clerks, who are providing the services, are the masters.

The last set of operations for specifying synchronisation within a simulation is the 'condq'. A condq (short for 'conditional queue') is a facility for allowing an entity to wait until an arbitrary condition is satisfied. Because of its wide scope the condq is both the most difficult synchronisation facility to learn and the most powerful to use.

A condq is created and returned by a call to the procedure

```
CREATE_CONDQ (condq_access; title);
```

An entity uses a condq by calling the procedure 'WAITUNTIL (condq_access; condition);'. If the condition evaluates to 'true' then the entity continues execution, otherwise the entity is blocked. Because of the extreme expense inherent in having the simulation continually testing to see if these conditions have become 'true' after every simulation state change, the responsibility for 'signalling' relevant state changes is rested squarely with the user (for a complete discussion of this see Birtwistle [2]). Whenever the simulation changes in such a way that one of these conditions could have become 'true' the user must insert a call to the procedure 'SIGNAL (condq_access);'. This results in the re-evaluation of the waituntil conditions of the entities waiting on that condq. If the condition has become 'true' then the entities are allowed to continue.

OTHER SAMOA FACILITIES

We now describe SAMOA's data collection, random number generation, and report facilities. SAMOA offers the user another set of data collection devices besides those provided by queues and by synchronisation devices. These include facilities for recording time dependent data (tally objects), for recording time independent data (accumulate objects), and for recording and displaying data in histogram form (histogram objects). Also, at any place in the simulation the user may request a report ranging from a report of all data collection devices, to a report of one collection type (e.g. all waitqs), to a report of one specific collection object.

The pseudo-random number generation routines provided in SAMOA are taken from DEMOS [2] and there they are described as "a Lehmer generator" published by Downham and Roberts [6]. It is

$$X_0 = \text{some seed generated by DEMOS}$$
$$X_{k+1} = (8192 * X_k) \bmod 67099547$$

and has a cycle length of 67099546" [2]. The random number routines are defined in three sets: six return a float value (NORMAL, UNIFORM, NEGEXP, ERLANG, CONSTANT, and EMPIRICAL), two return an integer value RANDINT, and POISSON), and one returns a boolean value (DRAW). In a simulation each random number generator is created separately and each can have different initial parameters.

SAMOA also provides an automatic trace facility. The procedure TRACE turns this facility on, while NOTRACE turns it off. Each trace listing entry is placed on a separate line and traces a single SAMOA defined event. Also included in the trace listing is the simulation time and current

entity associated with the traced event. The user can direct output to different files (trace, error, and report) so that different types of data can be easily identified.

With the addition of these other simulation facilities and the synchronisation routines we can finish constructing our library example (Figure 4). As previously mentioned, we use the waitq facility to allow customers to wait while the clerks provide the service of retrieving requested books from the stacks. Also, a bin is used to determine when the simulation has terminated. Because the simulation is to continue until 100 customers have been completely served we will have each customer deposit an item (their checkout slip?) into a bin as they leave. By having the main program (remember it is also represented as an entity) take 100 items from this bin we can guarantee that it will wait until exactly 100 customers have been served. Also four random number generators have been added, values are returned from these distributions by passing the proper access value to the procedure SAMPLE ( ). Note that because neither a customer nor a clerk has any global data structures we have omitted the definition of both 'customer_rec' and 'clerk_rec' from Figure 4.

In Figure 5 we see the GPSS solution, from Schriber [18], to the library example. The GPSS solution models the transactions within the simulated system by following the path of each customer's request for a book. The GPSS model is very similar in many respects to the SAMOA solution: the definition of customers and clerks are separated, these entity types synchronise their activities with one another, and their basic set of actions are similar to the SAMOA solution. Specifically, the GPSS code between the comments MODEL SEGMENT 2 and CONTROL CARDS represents the clerks. In the GPSS example the code which allows the entities to synchronise is difficult to decipher and the logic of this synchronisation is implemented in both entities. This destroys the clarity, modularity, and extensibility of this solution, qualities which are desirable in large simulation projects. Another problem is that code for simple control structures like if-then-elses is hard to recognize and the structure of the program is hard to follow. In contrast, the SAMOA example models each active component in the simulation in terms of how it sees the rest of the world and its variety of synchronisation primitives allow a more natural and modular representation of the entity - entity cooperation (in this case the waitq provides the necessary level of abstraction). Also, SAMOA has access to the full syntax of its base language, ADA. This makes the coding of loops and other control structures trivial because such constructs are built directly into ADA itself.

In the GPSS example statistics are gathered by the statements TABULATE DELAY and TABULATE SLIPS. In SAMOA statistics are automatically gathered by all queue and synchronisation objects. In the case of the library example the statistics that would be collected and reported include: 1) average customers waiting time, 2) average

Figure 4
Library Example with Synchronisation Added

```
procedure library is

  next_cust_time : poisson;
  travel_time    : uniform;
  get_books_time : normal;
  checkout_time  : uniform:
  custs_served   : bin;
  checkout_line  : waitq;

  task body customer is
    next : entity_ref := null;
  begin
    create_entity (next, "Customer");
    schedule (next, sample (next_cust_time));

    wait (checkout_line):
    give (custs_served, 1):
    terminate_entity;
  end customer;

  task body clerk is
    slip_num : integer := 0;
    cust : array(1..4) of entity_access:
  begin
  loop
    slip_num := 1;
    cust (slip_num) :- coopt (checkout_line):

    while length (checkout_line)>0 and slip_num<4
    loop
      slip_num := slip_num + 1;
      cust(slip_num):=coopt(first(checkout_line));
    end loop;

    hold (sample(travel_time));
    hold (sample(get_books_time));
    hold (sample(travel_time));

    for i in 1..slip_num loop
      hold (sample(checkout_time));
      schedule (cust(i), 0.0);
    end loop;
  end loop;
  end clerk;

  clerk_num : integer := 4;
  clerk     : entity_access := null;
  customer  : entity_access := null;

begin
  create_poisson (next_cust_time, 0.5);
  create_uniform (travel_time, 0.5, 1.5);
  create_normal  (get_books_time, 3, 0.2);
  create_iniform (checkout_time, 1.0, 3,0);

  create_waitq (checkout_line, "Checkout");
  create_bin (custs_served, "Custs_Served", 0);

  for 1..clerk_num loop
    create_entity (clerk, "Clerk");
    schedule (clerk, 0.0);
  end loop;

  create_entity (customer, "Customer");
  schedule (customer, 0.0);

  take (custs_served, 100);
  report;
end library;
```

number of customers waiting, 3) average clerk waiting time, and 4) average number of clerks waiting. Also, the user can gather statistics explicitly by using SAMOA's built in data collection devices (e.g. tally objects for time dependent data). All statistics are printed by calling the procedure REPORT.

Major advantages, not readily available in a GPSS approach, can be derived by implementing a simulation in SAMOA. Two of these advantages relate to the extensibility which can be achieved using ADA and SAMOA. In the first case, a specific, large scale library simulation could be written in two sections. The first section would define a simulation context (in this case a library) which would include definitions of all types of system processes (clerks and customers). The second section would specify a particular simulation environment, including the number of clerks, the number of customers, and appropriate values for the random number generators. This logical separation of the description of a system's components and the creation of a particular simulation environment illustrates the modular design which can be achieved using ADA's packages and which is especially important in the development of large models. The second advantage derived from extensibility is that because each entity is represented as a logically distinct objects its actions are independent, except at synchronisation points, of other entities. For example, if we changed the definition of a customer in the library problem so that they go to both school and the store before coming to the library we would only have to change the definition of customer, the definition of a clerk would be unaffected. This is because SAMOA allows us to construct the model so that a customer and a clerk are only dependent on one another when they synchronise at the checkout counter. In large simulations this allows the simulationist to build sets of predefined entities and routines whose definitions are independent of the model being constructed and who can therefore be plugged into a simulation whenever necessary.

Before concluding, two points have to be noted. First, all SAMOA entities, no matter what they represent, have the same information structures (i.e. entity_record, global data, and activities) and each of these structures has its own basic format. This consistent format allows SAMOA to be augmented by a separate, interactive program which generates the outline of these structures from a list of entity names supplied by the user. In essence, this rudimentary program generator creates the skeleton of the final SAMOA program into which the user simply inserts the simulation code. For example, by applying this skeleton program generator to the library example it would generate outline code similar to that in Figure 2 (without the comments). In a large simulation this could result in considerable savings in time and effort. The second point to note is that the SAMOA package owes much of its style, syntax, and philosophy to DEMOS [2]. DEMOS (Discrete Event Modelling on Simula), is a general purpose discrete event simulation package written in SIMULA by G. M. Birtwistle.

The use of the process style in the construc-

Figure 5
GPSS Solution to Library Example

```
*LOC     OPERATION       A,B,C,D,E,F,G
*        EQUIVALENCE DEFINITION(S)
 slip    equ             10,1
*        FUNCTION DEFINITION(S)
 snorm   function        rn1,c25
20,-5/.00003,-4/.00135,-3/.00621,-2.5/.02275,-2
.06681,-1.5/.11507,-1.2/.15866,-1/.21186,-.8
.27425,-.6/.34458,-.4/.42074,-.2/.5,0/.57926,.2
.65542,.4/.72575,.6/.78814,.8/.84134,1/.88493
1.2/.93319,1.5/.97725,2/.99379,2.5/.99865,3
.99997,4/1,5
 xpdis   function        rn1,c24
0,0/.1,.104/.2,.222/.3,.355/.4,.509/.5,.69/.6
.915/.7,1.2/.75,1.38/.8,1.6/.84,1.83/.88,2.12/.9
2.3/.92,2.52/.94,2.81/.95,2.99/.96,3.2/.97,3.5
.98,3.9/.99,4.6/.995,5.3/.998,6.2/.999,7/.9998,8
*        STORAGE CAPACITY DEFINITION(S)
         storage         s$busy,3
*        TABLE DEFINITION(S)
 delay   table           m1,360,60,26
 slips   table           x$count,1,1,5
*        VARIABLE DEFINITION(S)
 doubl   bvariable       x$count'e'4+w$wait'e'0
 gnorm   fvariable       (fn$snorm/5+1)*180*p2
*        MODEL SEGMENT 1
         generate        120,fn$xpdis,,,1
 wait    advance
         gate ls         slip
*
         assign          1,x$clerk
         savevalue       count+,1
         test e          bv$doubl,1,bypas
*
         logic r         slip
         TABULATE        DELAY
         terminate       1
*        MODEL SEGMENT 2
 bloka   generate        ,,,3
         assign          1,n$bloka
 blokb   test g          w$wait,0
         enter           busy
         save value      count, 0
         savevalue       clerk,p1
*
         logic s         slip
         buffer
*
         assign          2,x$count
         TABULATE        SLIPS
         advance         60,30
         advance         v$gnorm
         advance         60,30
 blokc   advance         120,60
         logic s         p1
         buffer
*
         loop            2,blokc
*
         leave           busy
         transfer        ,blokb
*        CONTROL CARDS, AND STG CAPACITY RE-DEFS
         start           100
         clear
         end
```

tion of system models is a procedure by which each system component is modelled in terms of how it sees the world. A one to one mapping of actual system components to simulation components supports the modular description of large systems. Further, our process based approach can be extended to represent special purpose simulation environments or packages. For example, the process approach as implemented in OASIS, [21], [22], [23] an extension of SIMULA, supports: the modelling of computer system hardware, the implementation of concurrent program units, and the simulated execution of these program units by the modelled hardware. If a base language compiler were available for the modelled hardware components, e.g. for the Motorola M68000 and the system being modelled was a local network of M68000s, then the model's program units could ultimately become the actual network software. This same modelling power would be available in an ADA based simulation environment and it is probable that the Motorola M68000 compiler that SIMULA lacks will soon be available for ADA.

The use of ADA as the base language for a discrete event simulation package could thus provide large improvements in modelling power over the current GPSS, SIMSCRIPT, and GASP approaches. Looking beyond SAMOA, it is possible to visualize a complete simulation environment as described by Birtwistle et al [3] based on ADA. Such an environment would provide the simulationist with special purpose tools, some of which would: 1) generate simulations and documentation from formal simulation specifications, 2) provide the user with animated traces of running simulations, and 3) simplify project management and software configuration. Furthermore, ADA's potential availability and the portability of ADA programmers and ADA programs make both a simulation package such as SAMOA and a simulation environment as outlined above very attractive proposals.

BIBLIOGRAPHY

1. Barnes, J.G.P.; Programming in ADA, Addison-Wesley, 1982.

2. Birtwistle, G.; DEMOS - A System for Discrete Event Modelling on Simula, MacMillan, 1979.

3. Birtwistle, G.; Liblong, B.; Unger, B.; and Witten, I.; "Simulation Environments", invited paper for 'Conference on Simulation: A Research Focus', ORSA, SIGSIM, Rutger's University, May 1982.

4. Bryant, R.M.; "Discrete System Simulation with Ada", submitted for publication to Simulation, 1983.

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

6. Dowhham, D.Y.; and Roberts, F.D.K; "Multiplicative congruential pseudo-random number generators", Computer Journal, Vol. 10, #1, 1967, pp. 74-77.

7. Franta, W.; A Process View of Simulation, Elsevier North-Holland, 1977.

8. Holback-Haussen, E.; Handlykken, P.; and Nygaard, K.; System Description and the DELTA Language, Norwegian Computing Center, Oslo, 1975.

9. Houle, P.; and Franta, W.; "On the Structural Concept of Simula", Australian Computer Journal, Vol. 7, #1, March, 1975, pp. 39-45.

10. Kernighan, B.; and Mashey, J.; "The UNIX Programming Environment", Computer, Vol. 14, #4, April 1981.

11. Kiviat, P.; Villaneuva, R.; and Markowitz, H.; The SIMSCRIPT II Programming Language, Prentice-Hall, 1969.

12. Ledgard, Henry, F.; ADA: An Introduction, Springer-Verlag, New York, 1981.

13. Norwegian Computing Center, "S-port, a portable Simula compiler", Available from the Norwegian Computing Center, PO Boks 335, Blindern, Oslo 3., Norway, 1980.

14. Norwegian Computing Center, "Vax Unix Simula", Simula Newsletter, Vol. 10, #3, August, 1982, p. 7.

15. Pritsker, A.; The GASP IV Simulation Language, John Wiley, 1974.

16. Pritsker, A.; and Pegden, C.; Introduction to Simulation and SLAM, John Wiley, 1979.

17. Russell, E.; Simulating with Processes and Resources in SIMSCRIPT II.5, CACI Inc., Arlington, 1974.

18. Schriber, T.; Simulation using GPSS, John Wiley, 1974.

19. "Stoneman", Requirements for the Ada Programming Support Environments, United States Department of Defense, February 1980.

20. Unger, B.; "Programming Languages for Computer System Simulation", Simulation, Vol. 30, #4, April, 1978, pp. 101-111.

21. Unger, B.; and Bidulock, D.; "The Design and Simulation of a Multi-Computer Network Message Processor", Computer Networks, Vol. 6, #5, August 1982.

22. Unger, B.; Bidulock, D.; Lomow, G.; Balanger, P.; Hankins, C.; and Jain, N.; "An OASIS Simulation of the ZNET Microcomputer Network", IEEE Micro, Vol. 2, #6, August 1982.

23. Unger, B.; and Lomow, G.; OASIS 4.0 Reference Manual, University of Calgary, Research Report 82/93/12.