DISCRETE SYSTEM SIMULATION WITH ADA

by

R. M. Bryant

Computer Sciences Technical Report #458

November 1981

Discrete System Simulation with Ada

R. M. Bryant*
Department of Computer Science
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, Wisconsin 53706

Ada, the new Department of Defense "standard" language, contains many features designed to facilitate the rapid construction of software for embedded computer system applications. Two of these features are "tasks" and "packages". Even though Ada is not designed to be a simulation language, these features allow the construction of powerful, process-oriented, discrete-system simulation packages in Ada. In this paper we provide a design for such a package and illustrate its use in a simple simulation program.

---

* This work was supported in part by the Wisconsin Alumni Research Foundation and through NSF grant MCS-800-3341.

Author's present address: IBM T. J. Watson Research Center, Yorktown Heights, N. Y.

1. Introduction

Ada* [10,2,11] is a new programming language designed at the initiative of the United States Department of Defense. While primarily intended to support embedded computer system applications, Ada is also suitable for more general programming tasks. Furthermore, the size of the Ada project, not to mention its backing by DOD, implies that Ada will have significant impact on the state of the art in program construction.

In this paper we provide a tutorial introduction to the features of Ada applicable to the construction of discrete-system simulation programs. We will concentrate on the concepts of "tasks" and "packages". Tasks represent independent threads of execution in a program and can be used to implement simulation processes similar to those of SIMULA [5,6]. However, a task cannot be assumed to run to completion before the physical processor is reassigned to another task. Thus, the tasks of Ada are more akin to the parallel computation model of processes implemented in MODULA [12] than the co-routine model of processes in SIMULA. The latter can be implemented in Ada through synchronization primitives that only allow one task to run at a time.

Packages allow the definition and use of separately compiled libraries of commonly used routines. Packages

---

* Ada is a trademark of the Department of Defense.

ensure that implementation details of a library routine can be hidden from the user and that the strong typing of Ada is not violated across separate compilation units. Additionally, Ada allows the definition of generic packages that can be parameterized at compile time to represent abstract data operations on user defined types. In this paper we outline an Ada package to implement queues of entities similar to those supported by SIMPAS [3] (originally inspired by the "sets" of SIMSCRIPT II.5 [8]), and a task-scheduling package called "simulation" that implements a task scheduler and a simulation delay statement "hold" like the one in SIMULA [5].

We assume that the user is familiar with the concepts of strong typing as implemented, for example, in Pascal [7]. Since Ada is based on Pascal, many of these concepts carry over in a straightforward way to Ada. We do not, however, intend to provide a complete introduction to Ada. The reader is directed to the existing literature for a more complete introduction [10,2,11].

2. A Queue of Entities Package in Ada

We will use the attribute-entity terminology of SIMSCRIPT II.5 [8] to describe objects that are present in a simulation. We distinguish between two types of entities: temporary and permanent. Permanent entities are those present throughout a simulation run. The standard way to represent permanent entities in Pascal is by records or

arrays of records [4]. Fields of the records represent the attributes of the entities. This representation can be used in Ada as well as in Pascal.

Pointer types are the natural way to represent temporary entities, since the object pointed to can be created, moved through the simulation and destroyed as necessary. Pointer types are called access types in Ada. One can thus create SIMSCRIPT II.5-like temporary entities using records and access types as shown below*:

```
declare

    current_time : real;
    when_arrived : real;

    type job_rec;              -- incomplete type declaration
    type ptr_job is access job_rec;

    type job_rec is            -- that is completed down here
        record
            arrival_time : real;
            memory_size  : integer;
        end record;

    new_job : ptr_job;

begin

    . . .
    new_job := new ptr_job;
        new_job.arrival_time := current_time;

    . . .
    when_arrived := new_job.arrival_time;
    . . .
end;
```

*We will adopt the conventions that keywords of Ada will be underlined, that names enclosed in "<" and ">" will represent identifiers to be supplied by the user, and items enclosed in braces ("{" and "}") are optional and may be omitted.

Notes:

(1) Comments in Ada begin with "--" and continue to the right margin.

(2) The duplicate declaration of job is necessary to resolve the forward reference problem, i. e. we want ptr_job to be of type access job, but the record job_rec also depends on ptr_job.

(3) Variable declarations are not preceded with the keyword var as in Pascal. Thus the variables declared in this block are new_job, current_time, and when_arrived. Type and variable declarations can be intermixed and are not restricted to type and var declaration parts as in Pascal.

(4) The "allocator" new serves the function of the procedure new in Pascal. Unlike Pascal, access objects in Ada exist so long as they are accessible. It is assumed that some type of garbage collection or reference count mechanism is used to identify and delete inaccessable objects. Hence there is no "dispose" procedure in Ada.

(5) Unlike Pascal, Ada does not (normally) require the use of an explicit pointer dereferencing operator. Thus one uses new_job.arrival_time to refer to this field of the record while in Pascal, one would have had to use new_job^.arrival_time.

Normally, one wants to maintain lists of entities ("queues" in SIMPAS and "sets" in SIMSCRIPT II.5). Lists can be implemented by allocating link fields in each entity and creating a record type to represent the head of the queue. Queue insertion and removal routines can then be constructed. In Pascal, the queue data structure and the queue maintenance routines would be declared separately and no connection between the declarations would be apparent. In Ada, a queue package can be declared that encapsulates the data structure and maintenance routines into a single unit.

2.1. Packages in Ada The main structuring unit of Ada is the package. It (normally) consists of two parts: a specification part and an implementation part. The specification part describes the interface to the package, and the implementation part contains the procedures and internal data structures that implement the package. A package is declared as follows:

```
package <name> is                    -- specification part
    . . .
{private                             -- private declarations
    . . .
}
end;

package body <name> is               -- package body
    . . .
end <name>;
```

The package specification declares the externally accessible procedures, types and variables defined by the package. Bodies of these procedures are contained in the package body. Outside the package, variables, procedures and types declared within the package specification are accessed by a dot notation like that used to access fields of a record in Pascal.

Types declared within the private part of the package are made available to the user without revealing their internal structure. The user may declare instances of the private type but still be shielded from changes in the implementation. For example, a package to implement a queue of jobs might be declared as follows:

```
package job_queue is -- FIFO queue

    type queue is private;

    procedure init(q : out queue);

    procedure put(job : in ptr_job; q : queue);

    function take(q : in out queue) return ptr_job;

private

    type link_record;
    type link is access link_record;
    type link_record is
        record
           next, prev : link;
           job        : ptr_job;
        end;

    type queue is
        record  -- queue is doubly linked
                -- list with head node
           head  : link;
           size  : integer
           empty : boolean;
        end;
end job_queue;

package body job_queue is

    procedure init(q : out queue) is
    begin       -- initialize q
        q.head          := new link;
        q.head.next     := q.head;
        q.head.last     := q.head;

        q.size  := 0;
        q.empty := true;

    end;

    procedure put(job : in ptr_job; q : queue) is
        holder : link;
    begin       -- put job at end of q
        q.size  := q.size + 1;
        q.empty := false;

        holder           := new link;
        holder.job       := ptr_job;
        holder.prev      := q.head.prev;
        holder.prev.next := holder;
        holder.next      := q.head;
```

```
        q.head.prev      := holder;
    end;

    function take(q : in out queue) return ptr_job is
        holder : link;
    begin       -- take first job from q
        if not q.empty then -- see note
            q.size   := q.size - 1;
            q.empty  := q.size = 0;

            holder           := q.head.next;
            q.head.next      := holder.next;
            q.head.next.prev := q.head;
            return(holder.job);
        else
            return(null);
        end if;
    end;

end job_queue;
```

Note: The control structures of Ada are essentially the same as those of Pascal. However, like Modula, Ada allows the body of a control statement (such as the if statement given above) to contain a statement list rather than a single statement. This rule eliminates most of the begin-end pairs used in Pascal.

Since "queue" is declared as a private type, the user of this package can declare variables of type "queue", but cannot access the internal structure of the record. The package implementor may change how the queue is represented without effecting the user program.

The queue itself is represented by a list of link records that point to the job. The reason for not placing the link fields in the job record itself will become apparent in the next section. Also we see that because the link records are declared only in the private type of the specification, they cannot be accessed outside of the package.

As an example of the use of this package, the following code declares two queues and moves the contents of one queue into the other:

```
declare
    job     : ptr_job;
    queue_1 : job_queue.queue; -- use the type declared in
    queue_2 : job_queue.queue; -- package job_queue
begin

    -- put some jobs into queue_1
    . . .

    -- now copy queue_1 to queue_2
    while not queue_1.empty loop
        job := job_queue.take(queue_1);
        job_queue.put(job, queue_2);
    end loop;

end;
```

One can avoid qualifying all of the procedures and types defined in package "job_queue" with the package name by placing the statement

```
use job_queue;
```

in the declaration part of the block. The effect of this specification is that references to the identifier "queue" are interpreted as job_queue.queue, "take" as job_queue.take and so forth.

2.2. Generic Packages One of the problems with the job_queue package is that it can only be used for queues of job entities. In order to use this package for queues of another type of entity, the entire package declaration would have to be repeated with a new type identifier inserted where appropriate.

Ada solves this problem through generic packages. The idea of a generic package is that the package declaration can be parameterized in terms of types, variables, or procedures to be specified at package instantiation time. A generic package thus specifies a template used to create a particular package.

For example, let us suppose we want to generalize the package given above so that it can be instantiated as a queue of jobs, or as a queue of messages, where a message is declared as:

```
type message is
    record
        size            : integer;
        sequence_number : integer;
        . . .
    end record;

type ptr_message is access message;
```

We can now change the declaration of job_queue to:

```
generic

    type entity_rec is private;
    type ptr_entity is access entity_rec;

package queue is -- FIFO queue

    type queue is private;

    procedure init(q : out queue);

    procedure put(entity : in ptr_entity; q in out : queue);

    function take(q : in out queue) return ptr_entity;

private

    type link_record;
    type link is access link_record;
    type link_record is
```

```
        record
           next, prev : link;
           entity     : ptr_entity;
        end;

    type queue is
        record
           head  : link;
           size  : integer
           empty : boolean;
        end;
end job_queue;

package body queue is
    . . .
end;
```

The package declaration is essentially as before except that every occurrence of "ptr_job" has been replaced by "ptr_entity".

This generic package has two generic parameters: the record type representing the entity we wish to place in the queue and an access type for that record type. Since the entity record is declared as a private type, the body of the package cannot use fields of the record. (Exactly as the user of the package cannot access fields of the queue record). This is the reason for using the link_records in the package declaration rather than placing the "next" and "prev" fields in the job or message declarations themselves. If the link fields were placed in the entity record, they could not be accessed inside of the package body.

One can now instantiate a queue package for each of jobs and messages:

```
declare

    job_queue is new queue(job,ptr_job);
```

```
    message_queue is new queue(message,ptr_message);

    waiting_jobs     : job_queue.queue;
    waiting_messages : message_queue.queue;

    job      : ptr_job;
    message  : ptr_message;

begin

    . . .

    job_queue.put(job, waiting_jobs);

    message_queue.put(job, waiting_messages);

    . . .
begin;
```

In SIMPAS or SIMSCRIPT II.5 separate routines are declared by the language processor for each type of queue or set declared. While an Ada implementation may handle instantiation of a generic package by duplicating the code, for our example the compiler could use exactly the same procedures for each type of queue. Thus the package concept not only hides the implementation of queues from the user, it also can allow different types of queues to share the same maintenance routines.

3. Tasks and Task Types

A task represents an independent execution that runs in parallel (at least conceptually) with all other tasks. A task is declared by a syntax similar to that of packages:

```
declare
    task <name> is     -- specification
        entry <name> {(<parameterlist>)};
        . . .
    end <name>;
```

```
task body <name> is -- body
    {declarative part}
begin
    . . .
end <name>;

begin -- body of block in which task is declared
  . . .
end;
```

Entry declarations are the only declarations allowed in a task specification.

Execution of a task begins as soon as the task that "elaborated"* the task declaration reaches the begin of the body of the block where the task was declared. Task execution continues until the task reaches the end of its body or until another task executes an abort statement for the task:

```
abort <task>;
```

(There are other ways to terminate a task; abort is sufficient for our purposes. See [10] for further details.) The elaborating task is not allowed to leave the block until all tasks declared in the block have terminated.

Tasks communicate by calling entries in other tasks. Corresponding to each entry declaration in a task are one or more accept statements in the task body. For example, consider two tasks, the first one having the specification:

```
task synch is
```

*When a block is entered various actions need to be performed to bring the objects declared in a block into existence. For example, array bounds may need to be evaluated, generic packages need to be instantiated, and in the present case, tasks need to be created and made ready to run. This process is referred to as "elaboration".

```
    entry wait;
end synch;

task body synch is
begin
    . . .
    accept wait do
        . . .
    end;
    . . .
end synch;
```

and the second containing the call:

```
    synch.wait;
```

Whichever task reaches its call or accept statement first waits for the other. When both tasks have reached the corresponding statements (this situation is called a "rendezvous"), parameters in the call are transferred and the body (if any) of the accept statement is executed. The calling task is delayed until the body of the accept statement has been completed.

Task declarations do not allow task instances to be dynamically allocated. To do this, one must declare a task type:

```
task type <name> is
    . . .
end <name>;
task body <name> is
    . . .
end <name>;
```

To declare a particular task instance one declares a variable of type <name>. One can also declare access types of type task. The tasks referenced by an access type become

active when they are allocated via the new operator. This allows the programmer to dynamically create a variable number of tasks and will be useful to us in the implementation of package "simulation".

4. Package 'Simulation'

In process-oriented simulation languages like SIMULA, each process runs until it blocks by calling procedure "hold". There is no need to use synchronization primitives to guarantee mutually exclusive access to global variables since there is only one process active at any given time. In Ada, tasks execute in parallel (or perhaps quasi-parallel if there is only one physical processor) under the control of Ada run-time routines. At any time, several processes may be active. To avoid data synchronization problems and follow standard process-oriented simulation practice, we will need a mechanism to make sure that only one task is running at any time. This is the purpose of package "simulation".

This package provides the following services:

(1) It maintains the current simulation time in a user-accessible variable called "sim_time".

(2) It implements the procedure hold:

```
procedure hold(delay : float);
```

whose purpose is to delay the current task until simulation time sim_time + delay.

(3) It guarantees that only one simulation task will be active at a time, provided that the user follows certain conventions in writing the simulation task.

To implement this package, we will use task types to create synchronizing objects [1]:

```
task type synch is
    entry send;
    entry wait;
end;

task body synch is
begin
    accept send;
    accept wait;
end;
```

A task can now delay itself until a signal is sent as follows:

```
declare
    signal : synch;
begin
    -- signal was activated when the block was entered
    . . .
    -- wait for some other process to do a signal.send
    signal.wait;
    . . .
end;
```

When the block is entered, a new synch task becomes active. It runs until it reaches the "accept send" statement and stops. When the creating task executes the call "signal.wait", it is delayed because the synch task is not ready to execute the "accept wait" statement yet. Until some other task executes a "signal.send" call, both tasks will be delayed.

4.1. Implementing the 'hold' Procedure

Given synchronizing objects, the procedure hold is implemented by maintaining a linked list of records declared as follows:

```
type future_event_notice;

type f_event_link is access future_event_notice;

type future_event_notice is
    record
        sched_time : float;          -- task reactivation time
        signal     : synch;          -- used to delay task
        next       : f_event_link; -- points to next notice
    end record;

first_f_event : f_event_link;       -- head of future event list
```

Each task delayed by a "hold" procedure call is associated with one such record. Procedure "hold" executes as follows:

(1) Calculate the simulation time when the calling task should be reactivated:

```
reactivate_time := sim_time + delay
```

(2) Create a new future event notice and set its time:

```
new_event            := new future_event_notice;
new_event.sched_time := reactivate_time;
```

(3) The future event list is maintained in increasing event time order. Search the list and insert new_event in the appropriate place.

(4) Wake up the simulation task scheduler (see below):

```
schedule.next;
```

(5) Delay the calling process by executing the entry call:

```
new_event.signal.wait;
```

A complete implementation of procedure "hold" is given in the Appendix.

4.2. Implementing the Simulation Task Scheduler

In order for a delayed task to proceed, some other task must execute the call

```
new_event.signal.send;
```

The simulation task scheduler is the task responsible for executing this entry call.

Use of the scheduler is complicated by the requirement that only one simulation task be active at a time. Ideally, one would like the scheduler to start a simulation task, wait until that task blocks itself, then schedule the next task and so forth. Unfortunately, there is no direct way to do this in Ada. We will therefore adopt the conventions that (1) procedure "hold" will activate the scheduler before delaying the calling task and (2) if a simulation task terminates, the last thing it does before terminating is to activate the scheduler. Given these conventions we can implement the scheduler task as follows:

```
task scheduler is
    entry start;
    entry next;
end scheduler;

task body scheduler is
begin

    --Wait for simulation start call.
    accept start;
    loop -- forever

        --If no more notices then wait
        --until next scheduler.next call.
        if first_f_event /= null then

            --Ready to run next simulation task.

            --Advance clock.
            sim_time := first_f_event.sched_time;

            --Activate the task.
            first_f_event.signal;

        end if;
```

```
            --Wait until that task blocks itself.
            accept next;

            --Advance to next future event notice.
            first_f_event := first_f_event.next;

        end loop;

end scheduler;
```

The simulation can be initialized by creating some simulation tasks and then calling scheduler.start. We assume that the main task will always execute the procedure call "hold(run_time)" after starting the simulation. The future event set should only become empty if the scheduler becomes active before the main task can execute the "hold(run_time)" procedure call. Since the hold procedure does a scheduler.next call in this case, the package should function properly.

An outline of the entire simulation package is given in the Appendix.

5. An Example Simulation

As a combined illustration of the ideas presented in this paper, we discuss the simulation of a simple queueing system in Ada. For this example, we assume that the packages "queue" and "simulation" are available and that a package named "random" that implements an exponential random number generator has also been implemented. The program header indicates that these packages are to be made accessible within program "example_simulation":

```
with queue, simulation, random; -- Bring in necessary packages.
```

```
procedure example_simulation is

    use simulation, random;
```

The use statement means that we can say "hold(t)" rather than "simulation.hold(t)" and "exponential(rate)" rather than "random.exponential(rate)".

To simplify the presentation, we assume that all parameters of the simulation are compile-time constants:

```
run_time     : constant := 1000.0; -- run for 1000 time units
lambda       : constant := 1.0;    -- arrivals per second
mu           : constant := 2.0;    -- services per second
```

We will use two tasks in the simulation. One will represent the arrival process and one will represent the service process:

```
task arrival is
    entry start;
end arrival;

task service is
    entry start;
    entry wakeup;
end service;
```

Entry "start" is used to initiate the simulation. Entry "service.wakeup" is used to restart the service process at the end of an idle period.

Customers in the system are represented by type "job". A job has two attributes:

```
type job;
type job_ptr is access job;
type job is
    record
        arrival_time : float;
        job_id       : natural;
    end record;
```

A generic instantiation of package "queue" is used to represent a queue of jobs:

```
job_queue is new queue(job,job_ptr);
waiting_queue : job_queue.queue;
```

We assume that there are some statistics collection variables, but we will not specify them:

```
--statistics collection variables
 . . .
```

We are now ready to specify the arrival and service tasks:

```
task body arrival is
    new_job : job_ptr;
    arrival_count : integer := 0;
    use random;
begin

    accept start; -- Wait for simulation start.

    loop -- forever

        -- Wait for time of next arrival.
        hold(exponential(lambda)); -- exponential defined in
                                   -- package "random"

        -- Create a new job.
        new_job := new job;

        -- Set attributes of the new job.
        new_job.arrival_time := sim_time;
        arrival_count        := arrival_count + 1;
        new_job.job_id       := arrival_count;

        --Place new job in waiting queue.
        --If system is idle, wake up the service task.
        if waiting_queue.empty then
            job_queue.put(new_job, waiting_queue);
            service.wakeup;
        else
            job_queue.put(new_job, waiting_queue);
        end if;
```

```
    end loop;

end arrival;

task body service is
    departing_job  : job_ptr;
    time_in_system : float;
    use random;
begin

    accept start; --Wait for simulation start.

    loop  -- forever

        if waiting_queue.empty then
            --No jobs to service - sleep until job arrives.
            accept wakeup;
        else
            --Sleep for duration of job service time.
            hold(exponential(mu));

            --Take the first job out of waiting queue.
            departing_job := job_queue.take(waiting_queue);

            --Observe time in system and record statistics.
            time_in_system := sim_time - departing_job.arrival_time;
            . . .

            --Let storage allocator reclaim job.
            departing_job := null;

        end if;

     end loop;

end service;
```

The main procedure initializes the waiting_queue, starts the simulation tasks, and then blocks itself until the simulation run time has expired:

```
begin -- main procedure

    --Initialize the waiting_queue.
    job_queue.init(waiting_queue);

    --Scheduler, arrival, service all blocked now;
    --Start them and the simulation.
    scheduler.start;
    arrival.start;
```

```
    service.start;

    --Wait until simulation run time has expired.
    hold(max_run_time);
```

All that's left to do is stop the simulation tasks and print statistics:

```
    --Terminate simulation tasks.
    abort arrival;
    abort service;
    abort scheduler;

    --Print statistics.
    . . .

end main;
```

6. Concluding Remarks

Until the introduction of Ada, implementation of portable, discrete-system simulation packages in existing higher-level languages often required that the package be event- rather than process-oriented. As we have shown, the task construct of Ada allows one to create process-oriented simulation packages. Generic packages allow the definition of statistics and queue packages that can be parameterized to represent statistics for real, integer, or boolean variables, or that represent queues of arbitrary user types. The essential features of languages like SIMSCRIPT II.5 or SIMPAS can thus be supported within Ada. This flexibility should promote the creation of standard discrete-system simulation packages within Ada similar in impact to GASP-IV [9] but with the advantages of strong-typing for efficient and reliable construction of large discrete system

simulation programs [3]. The design we have presented should be a first step in this direction.

7. Acknowledgement

I would like to thank Raphael Finkel for his help in answering my questions about the Ada language specification and his suggestions for improving this paper.

Appendix -- Simulation Package Declarations

```
package simulation is

    procedure hold(delay_time : float);

    sim_time : float := 0.0;

    task scheduler is
        entry start;
        entry next;
    end scheduler;

end;

package body simulation is

    task type synch is
        entry send;
        entry wait;
    end;

    task body synch is
    begin
        accept send;
        accept wait;
    end;

    type future_event_notice;

    type f_event_link is access future_event_notice;

    type future_event_notice is
        record
            sched_time : float;         -- simulation time of this task
            signal     : synch;         -- used to delay task
            next       : f_event_link; -- points to next notice
        end record;

    first_f_event : f_event_link := null; -- head of future event list

    task body scheduler is
    begin

        accept start;

        loop -- forever
            if first_f_event /= null then
                sim_time := first_f_event.sched_time;
                first_f_event.signal;
            end if;
            accept next;
```

```
            first_f_event := first_f_event.next;
        end loop;

    end scheduler;

    procedure hold(delay_time : float) is
        new_notice,
        trailing_ptr,
        loop_ptr    : f_event_link;
    begin
        new_notice := new future_event_notice;
        new_notice.sched_time := sim_time + delay_time;

        if first_f_event = null then
            first_f_event    := new_notice;
            new_notice.next  := null;
        else
            trailing_ptr := null;
            loop_ptr     := first_f_event;
            loop

                -- Insert new_notice here?
                if new_notice.sched_time <
                        loop_ptr.sched_time then
                    if trailing_ptr = null then
                        --Insert at front of future event set.
                        new_notice.next := first_f_event;
                        first_f_event   := new_notice;
                        exit;
                    else
                        --Insert after trailing_ptr.
                        new_notice.next   := loop_ptr;
                        trailing_ptr.next := new_notice;
                        exit;
                    end if;
                end if;

                -- No. Advance down list.
                trailing_ptr := loop_ptr;
                loop_ptr     := loop_ptr.next;

                -- Check for insert at end of list.
                if loop_ptr = null then
                    trailing_ptr.next := new_notice;
                    new_notice.next   := null;
                    exit;
                end if;

            end loop;

            --Wake up the scheduler.
            scheduler.next;
```

```
            --Delay this task.
            new_notice.signal.wait;

        end hold;

    end;
```

REFERENCES

[1] Barnes, J. G. P., "An Overview of Ada," Software--Practice and Experience 10, pp. 851-887 (1980).

[2] Brender, R. F. and I. R. Nassi, "What is Ada?," IEEE Computer 14, 6, pp. 17-24 (June 1981).

[3] Bryant, R. M., "SIMPAS -- A Simulation Language Based on PASCAL," Proceedings of the 1980 Winter Simulation Conference, pp. 25-40 (December 3-5, 1980).

[4] Bryant, R. M., "A Tutorial for PASCAL Users on Simulation Programming with SIMPAS," Computer Sciences Technical Report #454, University of Wisconsin--Madison (October 1981). Also Proceedings of the 1981 Winter Simulation Conference, Atlanta, Georgia, December 9-11, 1981.

[5] Dahl, O. J., K. Nygaard, and B. Myhrhaug, "The Simula 67 Common Base Language,," Pub S-22, Norwegian Computing Center, Oslo. (1969).

[6] Franta, W. R., The Process View of Simulation, Elsevier North-Holland, Inc., New York (1977).

[7] Jensen, K. and N. Wirth, "Pascal: User Manual and Report," Lecture Notes in Computer Science 18, Springer-Verlag Berlin, New York, (1974).

[8] Kiviat, P. J., R. Villanueva, and H. M. Markowitz, SIMSCRIPT II.5 Programming Language, C. A. C. I., Inc., 12011 San Vicente Boulevard, Los Angeles, California (1974).

[9] Pritsker, A. A. B., The GASP IV Simulation Language, John Wiley and Sons, Inc., New York (1974).

[10] U. S. Department of Defense,, Reference Manual for the Ada Programming Language, U. S. Government Printing Office, Washington, D. C. 20802. (July 1980). Proposed Standard Document, GPO 008-000-00354-8.

[11] Wegner, P., "A Self-Assessment Procedure Dealing with the Programming Language Ada," Communications of the ACM 24, 10, pp. 647-678 (October 1981).

[12] Wirth, N., "Modula: A language for Modular Multiprogramming," Software Practice and Experience 7, 1, pp. 3-35 (1977).