AN OPERATING-SYSTEM-BASED SIMULATION LANGUAGE

M. H. MacDougall
Control Data Corporation
Sunnyvale, California

INTRODUCTION

The concept of process has become increasingly important in describing and constructing computer operating systems. It provides a basis for representing the dynamic structure of a system, and for decomposing a complex system into simpler and more easily understood parts. A number of process-based operating system designs have been developed, and various mechanisms for control of process interaction (based on entities called signals, semaphores, events, etc.) have been proposed. The process concept is important not only in operating system design, but has fundamental importance in dealing with any complex system. As a consequence, a number of process-based simulation languages have been developed.

ASPOL (1-3) is a process-based simulation language developed specifically for computer system simulation. The process and process coordination facilities of ASPOL derive from those developed in various computer operating system designs; hence, ASPOL provides a natural vehicle for operating system simulation. However, the generality of these facilities is such that ASPOL can be used in simulating any discrete system. Other important features of ASPOL include the ability to operate on sets of entities, which simplifies modeling parallel systems, and macro facilities, which provide language extensibility.

This paper describes the process and process coordination (event) facilities of ASPOL, and discusses their application to computer system modeling problems. A number of computer operating systems, including those described in (4-7), provide process coordination facilities which often can be directly represented by those in ASPOL. Through the use of events and macros, ASPOL can be extended to represent a variety of operating system constructs, including process synchronization primitives such as Dijkstra's semaphore.

ASPOL comprises a language translator and a simulation run-time system called EXEC. An ASPOL simulation model essentially is composed of a set of procedures called process descriptions. These translate into machine-language subprograms; all the simulation declarations and operations appearing in these descriptions translate into calls on EXEC. EXEC is functionally equivalent to the executive nucleus of a computer operating system; its design is, in fact, based on the operating EXEC described in (7). As shown in Fig. 1, the translated process descriptions, together with EXEC, constitute the executable form of an ASPOL simulation model.

The operating system antecedents of ASPOL, then, are both functional and structural. ASPOL's process operations function much like those of various operating systems, and EXEC is a simplification of an actual operating system executive nucleus.

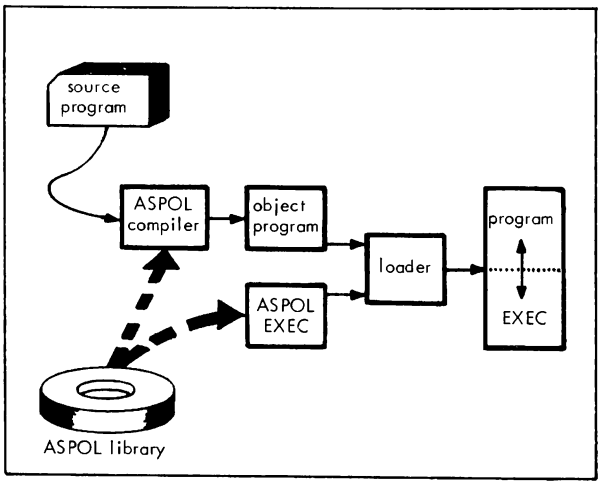

Fig. 1. Model Compilation and Execution in ASPOL

PROCESSES AND PROCESS DESCRIPTIONS

From a system point of view, a process is a dynamic entity, a singularly-occurring instance of execution of a set of logically related activities. In a computer system, execution of one particular disk request might be viewed as a process, composed of positioning, rotational delay, and data transfer activities. Processes composed of like sets of activities belong to the same class; a number of such processes simultaneously may exist in a system. Each is a particular instance of execution of that class, and is uniquely identifiable among members of that class by certain attributes (e.g., disk address, buffer address, request initiator's identity). The behavior of all processes of a given class may be specified by a single set of rules describing the activities common to all processes of that class, together with sets of attribute values for all members of that class. This set of rules is called a process description.

A system comprises, in dynamic form, a collection of interacting processes of one or more classes. A simulation model of a system is constructed as a set of process descriptions. In the simulation model, a process description is a procedure which apparently is executed simultaneously for all the existing processes of a class. In actuality, a process description is a multiply-reentrant procedure which is executed in a quasi-parallel fashion. As individual processes are initiated, execute, encounter delays, and terminate, control is switched to various parts of the procedure. The simulation run-time system keeps track of for which process a given part of the process description is being executed. From the simulation point of view, a process is a particular instance of execution of a process description.

An ASPOL simulation model comprises a main program, sim, and one or more process descriptions. sim begins with the declaration

sim model-name(f_1, f_2, . . . , f_n);

where the f_i represent file names, and ends with the delimiter

end sim;

which also terminates an instance of model execution. sim essentially is a process description of which only one instance of execution occurs during any one simulation run. Simulation entities, such as events and facilities, declared in sim are global to all process descriptions in the model, while those declared in a process description are created for each instance of execution of that description.

A process description begins with the declaration

process process-name (p_1, p_2, . . , p_n);

where the p_i represent formal parameters (e.g., variables, events) and ends with the delimiter

end process;

which also terminates process execution. Variables declared in real and integer declarations in a process description are local to each instance of execution of that description. Space for these local variables is allocated when a process is initiated and deallocated when the process terminates.

One process initiates another via the statement

initiate process-name (a_1, a_2, . . , a_n);

where the a_i represent actual parameters. Parameter transmission is uni-directional; a process may not return a value to its initiator by assigning a value to a parameter. Once initiated, a process executes independently of other processes, including its initiator, unless explicitly synchronized. Every process has an associated priority which it inherits from its initiator and which it may change via the assignment statement

priority = expression;

A process may suspend execution while awaiting termination of some process it has initiated via the statement

wait end (process-name);

Once initiated, a process exists in one of four states until it terminates. These states are

- execute - state of currently-executing process
- ready - state of other processes able to execute at the same instant in simulation time
- hold - state of processes which have suspended execution for a specified simulation time interval
- wait - state of processes which have suspended execution waiting for an event to occur or another process to terminate at some indefinite time

While processes in the actual system may execute concurrently, their counterparts in the simulation model must execute sequentially. At any instant during model execution, only one process in the model may be in execute state; others able to execute at that instant are in ready state. For example, when one process initiates another, the newly-initiated process is placed in ready state and the initiator continues in execute state. When the process in execute state suspends or terminates execution, another process is selected from those in ready state and placed into execution. This selection is made on the basis of process priority; among equal-priority processes, the earliest to enter ready state is selected.

A process may suspend execution for some simulation time interval 't' (as when representing the execution time of an actual system activity) via the statement

hold(t);

The process is placed in hold state and assigned a reactivation time t_e which is the sum of the current simulation time, time, and the interval 't'. Whenever the currently-executing process suspends or terminates execution and no other processes are in ready state, the process with the earliest t_e is selected from those in hold state, time is advanced to t_e, and the selected process is placed in execute state.

A process may suspend execution while waiting for some event to occur (e.g., for a facility to become free, or for another process to leave its critical section) via a wait or queue statement. The process is placed in wait state until the event occurs; it then is placed in ready state until selected for execution.

Each process in the model is represented by a process descriptor managed by EXEC. This descriptor is created when a process is initiated and destroyed when the process terminates. The data maintained in this descriptor includes the process priority, reactivation address, state, and a pointer to the start of the memory space allocated for local variables of the process. Initiation, execution, and suspension of a process (as a consequence of its execution of statements such as hold or queue) is reflected in EXEC by threaded the descriptor for that process on various lists, such as the time delay list, the ready list, or an event selector list.

EVENTS

Processes coordinate their activities by means of events. An event is a variable which may assume either the value "occurred" (set) or "not occurred" (clear) as the result of process actions. Events provide a general mechanism for process synchronization. ASPOL's events are similar to those of several operating system designs, including those described in (4-7).

Events may be explicitly defined and created by the declaration

event e_1, e_2, . . e_n;

where the e_i represent non-dimensioned (simple) or singly-dimensioned (set) event names. Events also are implicitly defined and created by the declaration of facilities and storages. Events declared in sim are global to all processes in the model and exist for the duration of the simulation run. Those declared in a process description are local to processes of that description; a unique instance of these local events is created each time a process of that description is initiated, and cancelled when the process terminates. Local events may be transmitted from one process to another as process initiation parameters.

Event declarations translate into calls on EXEC and result in EXEC's creation of an event descriptor. The data maintained in this descriptor includes the event state and pointers to lists of processes queueing and waiting for the event to occur.

Event Operations: Simple Events

An explicitly-defined event 'e' has the value "not occurred" when created; it is assigned the value "occurred" when a process executes the statement

set(e);

A process may suspend execution until event 'e' occurs via the statements

wait(e); or queue(e);

the execution of which is called event selection. Processes suspended awaiting the occurrence of an event are called the selectors of that event. Those processes which selected the event via a wait statement all are activated when the event occurs (i.e., they are placed in ready state). Of the processes which selected the event via queue statements, only one is activated when the event occurs. This one is the highest priority queued process; among equal-priority processes, the earliest selector is chosen. Event selections may be a mixture of queue and wait selections; occurrence of the event activates all all waiting selectors and one queued selector.

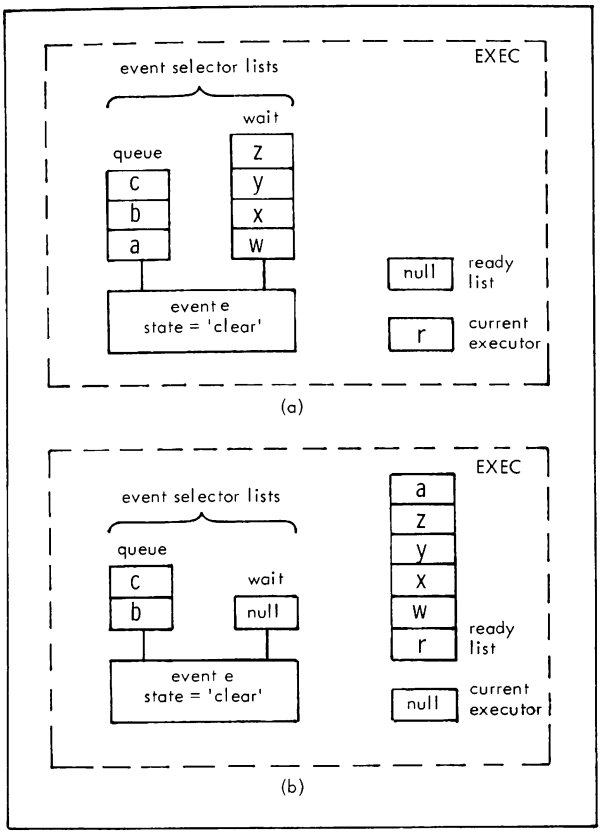

Fig. 2. Queue/Wait Selection Processing

The activation of event selectors upon occurrence of an event is called event recognition; recognition of an event causes the event to be extinguished: i.e., to be assigned the value "not occurred". Note that an event may be placed in the "occurred" state (via a set statement) and then be immediately returned to the "not occurred" state because of event recognition.

The processing of queue and wait selections may be clarified by an example. Suppose processes a, b, and c are queued on event e, processes w, x, y, and z are waiting on event e, and process r is executing. Assume all processes are of equal priority and the ready list is empty. Then the state of the relative parts of EXEC is as shown in Fig. 2(a). Now suppose process r executes a set statement, causing event e to occur. EXEC first places process r on the ready list. This is done because, in the general case, r's setting of event e may activate a higher priority process which should execute before r continues in execution. Next, event e is placed in the "occurred" state. Since the event has selectors, event recognition processing takes place; all the processes in the wait selector list are moved to the ready list, the process at the head of the queue selector list is moved to the ready list, and the event is extinguished. The new state of EXEC, just prior to its selecting a process from the ready list for execution, is shown in Fig. 2(b).

A parameter may be transmitted from the process which sets an event 'e' to processes selecting that event via a set statement of the form

set(e)expression;

and selection statements of the form

wait(e)variable; or queue(e)variable;

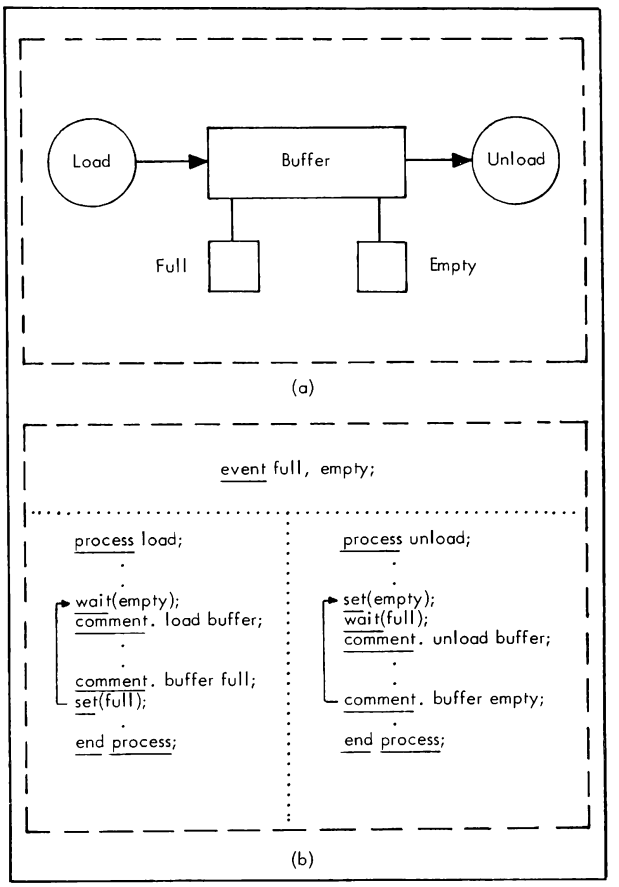

Fig. 3. Buffer Control Via Events

When the selectors of the event are returned to execution, the variable in each selecting process is assigned the value of the expression at the time the set statement was executed. If the event is set repeatedly without intervening selections and then selected, the variable will be assigned the value of the expression accompanying the most recently executed set statement.

A common situation requiring coordination of asynchronous processes is buffer control. In the buffering operation diagrammed in Fig. 3(a), one process loads the buffer at an indeterminate rate and another process unloads it, also at an indeterminate rate. The two processes are assumed to always completely load or completely unload the buffer, and share some signaling mechanism to notify one another when the buffer is full or empty. In an operating system, the two processes might be the disk control program and some user task; the signaling mechanism then would be based on the operating system's event facilities. At the hardware level, the two processes might be imbedded in the logic of a device controller and a data channel; the signaling mechanism then might be based on full and empty flip-flops.

Fig. 3(b) shows a prototype for an ASPOL model of the buffer control operation. The events 'full' and 'empty' can be shared between the two processes either by defining them in sim (and so making them global), or by defining them in a process and transmitting them as process initiation parameters.

One process may initiate another and, either immediately or at some later point in its execution, suspend execution while awaiting termination of the initiated process via the wait end statement described earlier. A general

coordination mechanism, permitting reactivation of the initiator at any desired point in the initiate's execution (rather than solely at its termination) is achieved by use of events. An example is shown below.

```
process e;                      process f(y, . . );
event x;                        event y;
   .                               .
initiate f(x, . . );               .
   .                               .
wait(x);                        set(y);
   .                               .
end process;                    end process;
```

Here, process e creates a local event 'x' and transmits it to process f as a process initiation parameter. The event declaration in process f defines formal parameter 'y' as an event, and does not itself cause creation of an event. Process e then suspends execution while waiting for f to progress to some desired point; f's setting of event y returns process e to execution.

A similar application of local events is the control of critical section execution. A critical section is a set of operations which are permitted to be performed by only one process at a time. Dijkstra, in (8), discusses this and other aspects of the mutual exclusion problem, and describes critical section control algorithms based on semaphores. A semaphore is an integer-valued variable on which only two operations, P and V, are allowed. The operation V(s) increments the value of semaphore s by 1. The operation P(s) decrements the semaphore by 1; however, this operation is blocked and the issuing process is suspended if the resulting semaphore value would be negative. The operation is allowed to complete and the issuing process returned to execution when a V(s) operation is performed by some other process.

An example of a process description in which a semaphore is used to prevent multiple instances of the description from simultaneously executing the critical section is shown below. In this example, the semaphore declaration identifies formal parameter s as a semaphore; the same semaphore is transmitted to all instances of process description e. It is assumed that the semaphore is set to 0 when created and that the initiator executes a V(s) operation to permit an initial instance of e to execute the critical section.

```
process e(s);
sempahore s;
   .
   .
comment.  enter critical section;
P(s);
   .
   .
comment.  leave critical section;
V(s);
   .
end process;
```

The above example could be implemented directly in ASPOL by using macros and events to construct the semaphore declaration and the P and V operations. The macro definitions might appear as follows.

```
macro semaphore $$;
  event [1];
endmacro;

macro P $($);
  queue([1]);
endmacro;

macro V $($);
  set([1]);
endmacro;
```

In these definitions, the "$-$" and "$-;$" delimit the delimiters of macro call arguments, and "[n]" specifies substitution of macro call argument n.

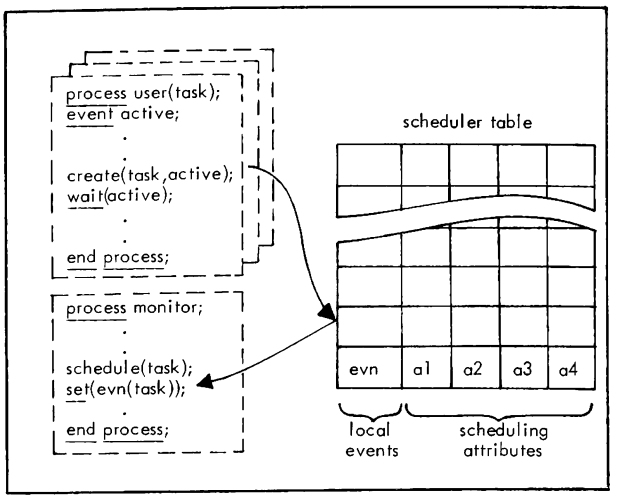

Fig. 4. Scheduler Example

Events and semaphores are similar in kind but are not identical; an exact representation of the general semaphore requires a slightly more elaborate macro definition then that shown here. However, the example illustrates the extensibility of ASPOL to encompass other forms of process coordination facilities appropriate to particular simulation environments.

ASPOL event queues are ordered on the basis of process priority; processes of equal priority are enqueued first-in, first-out. A wide variety of queueing disciplines can be accomodated by event queues through the appropriate computation of priorities. There are, however, various queueing cases which are not easily implemented by use of event queues. Also, it sometimes is desired to develop detailed models of actual system data structures. Both these situations may arise in the simulation of scheduling algorithms, where decisions are based on a number of attributes and may involve multiple queues.

A simple example of a model of a table-driven scheduler is sketched in Fig. 4. Here, user processes, upon initiation, are assigned a scheduler table entry 'task'. Each user process creates a local event 'active'; it then calls a 'create' procedure which computes the values of scheduling attributes a1 - a4 and enters these, together with event, in the assigned scheduler table entry. The user process then suspends execution waiting for event 'active' to be set. When the system monitor process is initiated to place a new user process into execution, it calls the "schedule' procedure. This procedure locates the best candidate for execution, based on the attributes recorded in the scheduler table, and returns a pointer to the table entry of the selected candidate. The monitor process then sets the local event of that candidate to place it into execution.

ASPOL provides the integer function procedures state(e) and length(e) which, respectively, return the state of event e and the number of processes enqueued on event e. A monitor(e) procedure initiates the collection of queueing statistics; these are reported at the end of the simulation run.

Event Operations: Set Events

A set of n events may be defined and created via a declaration of the form event e(n). In processing this declaration, EXEC creates a set of n + 1 event descriptors, one representing the set, the others representing elements of the set. The event representing the set will have the value "occurred" as long as any member of the set has that value; it will have the value "not occurred" only if all elements of the set have the value "not occurred". A simple event is equivalent to an event set of dimension 1; event set operations may reference simple events.

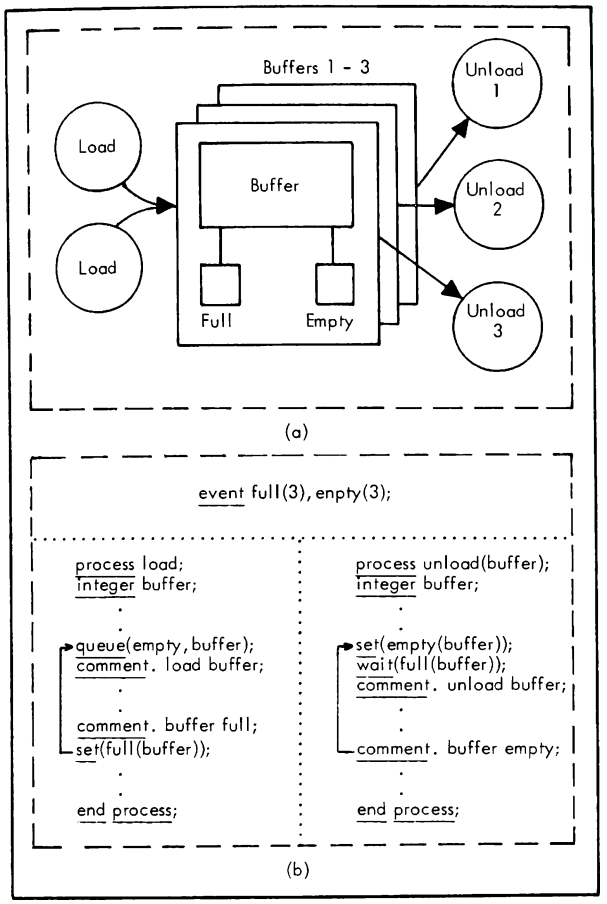

Fig. 5. Multiple Buffer Control Via Events

Processes may operate on elements of the set or on the set itself. The following forms of event selections may be used.

$$\underline{\text{wait}} \atop \underline{\text{queue}} \left\{ \begin{matrix} (e) \\ (e, i) \\ (e(i)) \end{matrix} \right\} ;$$

Here, e represents a set event name and i represents an integer variable. The first form (e.g., wait(e)) selects the set; the selector is returned to execution when any element of e occurs and is recognized. The second form (e.g., wait(e, i)) operates the same way with the difference that, upon return of the selector to execution, the variable i is set to the number of the element whose occurrence caused activation of the selector. The third form selects the ith element of the set; the selector is returned to execution only when that particular element occurs and is recognized.

set statements operate on set events in a similar fashion. The statements set(e) and set(e, i) cause the event set to be examined beginning with element 1. The first "not occurred" element found is assigned the value "occurred" and, in the case of the second of these two forms, the variable i is set to the number of that element. The statement set(e(i)) assigns the value "occurred" to the ith element of the set. In all cases, setting an element of the set causes the event representing the set to be assigned the value "occurred".

A set event may have processes queued and waiting both on elements of the set and on the set itself. When one of the events in the set occurs, event recognition processing is done as follows. First, the process setting the event

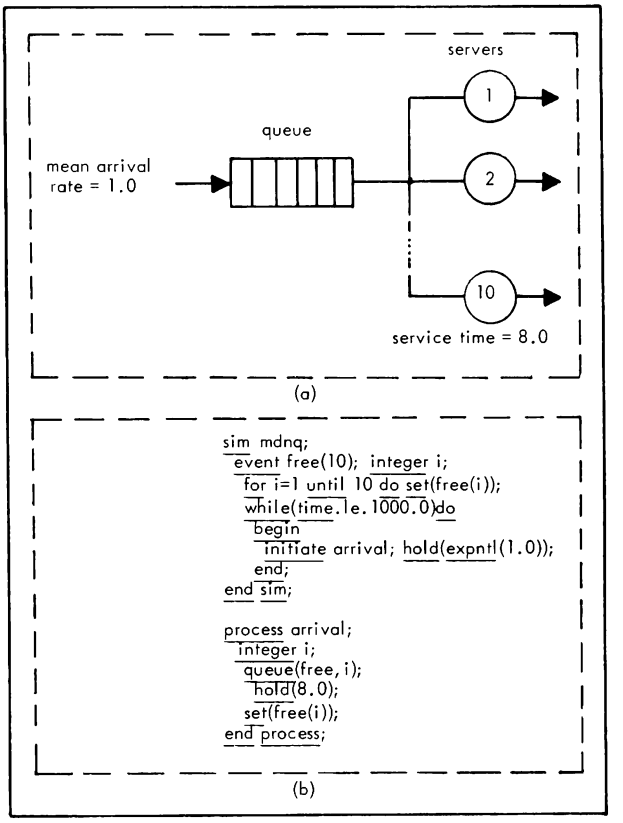

Fig. 6. Multi-Server Queue Model

is placed on the ready list, all element wait selectors are moved to the ready list, and the process at the head of the element's queue selector list (if any) is moved to the ready list. Event recognition for a set event element is identical to that for a simple event. Next, selections of the set are processed. All event set wait selectors are moved to the ready list; however, the process at the head of the event set's queue selector list is moved to the ready list only if there was no queue selector for the event set element. Thus, element selections are given preference over set selections.

A variation of the buffering problem discussed earlier will be used to illustrate event set operations. In the buffering operation diagrammed in Fig. 5(a), there are three buffers, each with an unload process assigned to it. These buffer are filled by two load processes which may operate on either buffer. A prototype for an ASPOL model of this operation is shown in Fig. 5(b). It is assumed that the initiator of the three instances of the unload process specifies the buffer to be emptied by each instance. This model may be compared with that of Fig. 3(b). Note that the two load processes in the multiple buffer model use a queue statement, rather than a wait statement, to suspend execution while waiting for a buffer to be emptied. This insures that, when both load processes are waiting and one buffer becomes empty, only one load process is activated to fill that buffer.

Fig. 6 shows a second example of event set operations. Fig. 6(a) is a diagram of an M/D/n queueing system. Customers arrive at the system at an average rate of 1 per unit time; their inter-arrival times are distributed in accordance with a negative exponential distribution. Service times are fixed and equal to 8 time units. A complete ASPOL simulation program for this system is shown in Fig. 6(b).

FACILITIES AND STORAGES

ASPOL provides entities called facilities and storages. From an implementation point of view, these are an extension

of events. Facilities are defined and created by the declaration

facility f_1, f_2, . . . , f_n;

where the f_i represent non-dimensioned (simple) or singly-dimension (set) facility names. Created with a facility is an event (or set of events, for a facility set) of the same name. A facility is a variable which may assume the states "busy" or "nonbusy". When created, it is placed in the "nonbusy" state, and the associated event is assigned the value "occurred". Facility operations include the following.

$$\left.\begin{array}{l}\underline{\text{reserve}}\\ \underline{\text{preempt}}\\ \underline{\text{release}}\end{array}\right\}\ \left\{\begin{array}{l}(f)\\ (f,\ i)\\ (f(i))\end{array}\right\}\ ;$$

Facility set and set member selections have the same form as those described for events: e.g., reserve(f.i) reserves the first-found nonbusy facility element in the set and returns the number of the element as the value of i.

The statements reserve(f) and preempt(f) cause a non-busy facility to be placed in the busy state and assign the value "not occurred" to the associated facility event. A release(f) statement returns the facility to the nonbusy state and sets the facility event, causing it to be assigned the value "occurred". Processes may explicitly suspend execution while waiting for a facility to become free by selecting the associated facility event via a queue(f) statement. Also, a process attempting to reserve a busy facility is automatically enqueued on the facility event. A process may preempt a facility reserved by a process of lower priority via a preempt statement; the preempted process is enqueued on the facility event and, if it was in hold state, its remaining hold time is saved. If the process reserving the facility is not of lowere priority, the preempt statement is treated as a reserve statement.

When processes are enqueued on a facility event, occurrence of that event (caused by a release statement) initiates event recognition processing, which places the process at the head of the queue in ready state. If the process was enqueued because it attempted to reserve a busy facility, the facility is reserved for it upon its return to execute state. If the process had been preempted, the facility is reserved for it and it is returned to hold state.

A variation on the example of Fig. 6 illustrates some of the facility declarations and operations. Assume that, in the queueing system of Fig. 6(a), arriving customers have priorities equiprobably distributed between 1 and 10, and that higher-priority arrivals finding all servers busy preempt a server reserved by a lower-priority process. An ASPOL model of this system might appear as follows.

```
sim mdnq;
  facility server(10);
    while(time.le.1000.0)do
      begin
        initiate arrival; hold(expntl(1.0));
      end;
end sim;

process arrival;
  priority=irandom(1,10);
    preempt(server);
      hold(8.0);
    release(server);
end process;
```

The arrival process descriptions in both this and the preceding example are independent of the number of servers. Note, in the example above, that the specific element reserved by an arrival process need not be specified in a release statement.

Storages are defined in ASPOL very much in the same way as facilities, and the allocation and deallocation of storage space functions much like the reservation and release of facilities. Storages also have events associated with them; a process attempting more than the available storage space is queued on the associated event until its requirements can be met. For a more detailed description of facilities and storages, see (2).

CONCLUSIONS

The success of events as a means of coordinating simulation processes has been demonstrated in hundreds of models. These have ranged from detailed models of complete, large-scale, computer operating systems down to hardware level models. Process coordination via events permits a high degree of process independence to be realized. Processes can control their activities by setting events and waiting for events with minimum connection to, and information about, the other parts of the model. This is advantageous in the hierarchical development of models, in constructing multi-level models, and in multi-person model development projects.

There are several areas in which ASPOL's event facilities could be improved; two such areas are set operation stacking and multiple event selection. As set statements currently are implemented, repeated setting of a particular event (a simple event or event set element) without intervening selections, followed by a selection, functions as if only the last set operation occurred, and parameters accompanying the preceding set statements are overwritten. It may be desirable to 'stack' set operations and execute them in first-in, first-out fashion as selections occur.

A desirable extension is multiple event selection: i.e., permitting a process to suspend execution while waiting for some arbitrary combination of events to occur. Two possible forms of multiple event selection, called the operative and declarative forms, are being considered. These forms are illustrated below.

OPERATIVE FORM	DECLARATIVE FORM
event e1, e2, e3;	event e1, e2, e3;
.	event e4 = e1∧e2∨¬e3;
.	.
.	.
wait(e1∧ e2∨ ¬ e3);	wait(e4);

Both forms have advantages and disadvantages from the implementation standpoint. The operative form requires evaluation of the event expression on each execution of the wait statement, the declarative form requires propagating event occurrence through a chain of event descriptors. From the standpoint of use, the declarative form has the attraction that a single event can be transmitted to a process, and that process then can proceed to execute unaware of the complexity of the conditions which control it.

REFERENCES

1. ASPOL Reference Manual, Control Data Corp., Pub. No. 17314200, 1972.
2. MacDougall, M. H., "Process and Event Control in ASPOL", Proc. Symposium on the Simulation of Computer Systems III, 1975, p39-51.
3. MacDougall, M. H., and McAlpine, J. S., "Computer System Simulation with ASPOL", Proc. Symposium on the Simulation of Computer Systems I, 1973, p93-103.
4. Dahm, D. M., Gerbstadt, F. H., and Pacelli, M. M., "A System for Resource Allocation", Comm. ACM 10, 12, p772-779, Dec. 1967.
5. Cleary, J. G., "Process Handling on the Burroughs B6500", Proc. Fourth Australian Computing Conf., 1969, p321-329.
6. Rossiensky, J., and Tixier, V., "A Kernel Approach to System Programming: SAM", Software Engineering, Vol. 1, J. Tou (ed.), Academic Press, New York, 1969, p205-224.
7. Earl, D. B., and Bugely, F. L., "Basic Time-sharing: a System of Computing Principles", Proc. 2nd Conf. on Operating System Principles, 1969, p75-79.
8. Dijkstra, E. W., "Cooperating Sequential Processes", Programming Languages, F. Genuys (ed.), Academic Press, New York, 1968, p43-112.