The simulation language SIML/I

by M. H. MACDOUGALL

Amdahl Corporation
Sunnyvale, California

INTRODUCTION

The conceptual advantages of process-oriented simulation languages have become generally recognized. SIMULA\textsuperscript{1,4} is the best known and most widely available language of this type; others include ASPOL,\textsuperscript{3,12} SIMPL/I,\textsuperscript{9} and SOL,\textsuperscript{10,11} and processes have been retrofitted into SIMSCRIPT.\textsuperscript{14} SOL was the genesis of the process view of system behavior. SIMULA is an elegant general-purpose programming language; its particularization to system simulation is nicely described by Franta.\textsuperscript{7}

The process view introduced by SOL evolved more rapidly in the area of operating systems than in system simulation, with consequent influence on the development of process synchronization constructs. In SOL, processes could control their activities in accordance with the values of expressions involving global variables. In operating systems, considerations including efficiency and the need for hierarchical structures resulted in the development of more specialized constructs for process synchronization; these included events,\textsuperscript{2,5} semaphores,\textsuperscript{6} and monitors.\textsuperscript{8} Recent simulation languages often incorporate similar constructs.

Among the advantages of this approach is the extent to which similitude—the resemblance of the model to the design—can be realized in system-level models. (While it is possible that a simulation model can be a valid representation of a system in a behavioral sense and, at the same time, bear little resemblance to the system, it usually is difficult to extend such a model to represent increased detail or to reflect design changes. At the very least, a lack of similitude can hinder communication between designer and modeler.) This trend ultimately may result in the merger of sinumodel and design specification, as proposed by Randell.\textsuperscript{13}

The improvements such languages bring to system-level modeling, and the relative ease with which they permit expansion of the level of detail of models of software elements, may not be realized when the modeling objective is oriented more toward the hardware elements of the system. For example, ASPOL is a simulation language whose process synchronization facilities, based on entities called "events," derive from operating system constructs. It serves quite well for the development of various kinds of system-level models (perhaps its main shortcomings at this level are in the area of facility preemption and the associated process interruption and queueing). For models requiring a more detailed representation of hardware elements (instruction pipeline simulations, bus conflict models, etc.), the operating-system-oriented event facilities are much less satisfactory. For the sake of both simplicity and similitude in such models, it often is desirable to represent important control functions essentially at the logic level. Modeling these functions with events is awkward at best; less specialized constructs are needed at this level of detail.

SIML/I is designed to provide a simulation capability which extends into the current gap between system-level and register-transfer-level simulation languages. Its process synchronization facilities are logic-oriented: they permit straightforward representation of the logical expressions involved in modeling hardware structures, as well as meeting the generally simpler needs of system models. SIML/I takes the form of an extension of PL/I, so the general programming facilities of PL/I are available to the modeler. The basic simulation constructs of SIML/I—models, processes, and signals—are described in the following sections.

MODELS AND SUBMODELS

The procedural forms of SIML/I include model, submodel, and process descriptions, together with the procedure descriptions of PL/I. A SIML/I simulation program may comprise a single model description, or may comprise a model description together with one or more submodel descriptions. A model description begins with a MODEL declaration, which is analogous to the PL/I declaration PROCEDURE OPTIONS(MAIN), and ends with the delimiter END MODEL. Model execution is initiated by the system as directed upon completion of loading. Submodel descriptions are separately compiled components of a simulation model and constitute separate load modules. A submodel description begins with the declaration SUBMODEL, which may be accompanied by a formal parameter list, and ends with the delimiter END SUBMODEL. Execution of a submodel is initiated by a model or submodel via an INITIATE statement, which may include an actual parameter list. Once initiated, a submodel executes independently of its initiator unless their activities are explicitly coordinated. Parameters may be transmitted to submodels at the time of their initiation. Thereafter, a model and its submodels may communicate with one another only through variables and signals declared external in each description.
The bodies of model and submodel descriptions essentially are identical in form. All process descriptions appear directly in a model or submodel description; a process description may not itself contain process descriptions. Model, submodel, and process descriptions may contain PL/I procedures, but these may not contain simulation declarations or statements. Rules for the scope of names of simulation entities are similar to those defining the scope of PL/I names. Signals declared in a model or submodel description, as well as PL/I variables declared in that description, are global to all contained process descriptions. Only a single instance of execution of a model description occurs in a simulation, so only a single instance of variables and signals declared in a model description is created. Multiple instances of submodel description execution may be initiated; for each, a unique instance of the signals and variables declared in that description (except for those declared external) is created.

Once initiated, a model or submodel executes like and exists in the same states as any process. A model or submodel may initiate submodels and processes, wait for and set signals, etc. The current simulation time is maintained as the global variable TIME. A model, submodel, or process may suspend its execution for an interval \( T \) via the statement \( \text{HOLD}(T) \); it will be returned to execution at time \( \text{TIME} + T \). Execution of a \( \text{TERMINATE} \) statement in a model terminates the simulation; execution of a \( \text{TERMINATE} \) statement in a submodel terminates only that particular instance of submodel description execution. The delimiters \( \text{END MODEL} \) and \( \text{END SUBMODEL} \) are implicit \( \text{TERMINATE} \) statements.

Submodels are important in developing large simulation models (since model components can be separately compiled), in multi-level modeling (where submodels of different levels of detail can be constructed and combined as needed for a particular simulation), and in modeling parallel systems (since multiple instances of execution of submodel descriptions can be initiated). As an example of the last, suppose a submodel of a disk subsystem is developed as part of a computer system simulation model. This submodel might represent a data channel, the controllers connected to that channel, and the devices connected to each controller. To simulate a configuration containing several similar (but not necessarily identical) disk subsystems, several instances of execution of this submodel description could be initiated.

**PROCESSES**

A process description begins with the declaration \( \text{PROCESS} \) (which may have a formal parameter list) and ends with the delimiter \( \text{END PROCESS} \). A process is a particular instance of execution of a process description; a number of such instances may simultaneously exist at any point in a simulation. A process may be initiated by the model or submodel in which it is contained, or by some other process in that model or submodel, including a process of the same description. Upon initiation, a unique set of the variables and signals declared in the process description is created. After initiation, a process executes independently of its initiator and of processes of the same or different descriptions unless explicitly synchronized. Process execution is terminated via execution of a \( \text{TERMINATE} \) statement or the delimiter \( \text{END PROCESS} \); upon termination, variables and signals local to the process are destroyed.

A process (or model or submodel) exists in one of four states: execute, ready, hold, or wait (queue). A process in execute or ready state is active; a process in hold or wait state is suspended. Since the simulation of a system of concurrently-executing processes is carried out sequentially, only one simulation process is in execute state at any instant; any others able to execute at that instant are in ready state. For example, when a process is initiated, it is placed in ready state; its initiator continues in execute state.

A process may suspend its execution until a signal changes state via a \( \text{wait} \) (or \( \text{queue} \)) statement, in which case it is placed in wait state. When the selected signal changes state, the process is placed in ready state (activated). A signal's change of state may activate several processes, so that a number of processes may be in ready state at any instant; these processes are placed on the ready list. When the currently-executing process suspends its execution, the process at the head of the ready list is removed and placed into execution. The ready list is ordered on the basis of priority: equal-priority entries are ordered first-in, first-out. Priority is an attribute of a process (or model or submodel) which may be changed at any time via an assignment statement of the form \( \text{PRIORITY} = \text{expression} \). Submodels and processes inherit the priority of their initiators.

A process may suspend execution until a specified reactivation time is reached via a \( \text{HOLD} \) statement, in which case it is placed in hold state and some other process selected from the ready list and placed into execution. When a process suspends execution and the ready list is found empty, the process with the earliest-occurring reactivation time is selected from the set of processes in hold state, the simulation time \( \text{TIME} \) is advanced to that time, and the process is placed on the ready list. If several processes have the same reactivation time, all are placed on the ready list in priority order. The process at the head of the ready list then is placed in execution.

Parameters may be transmitted to a process or submodel by its initiator. Parameter transmission is uni-directional; parameters are passed by value at initiation time, and modification of a variable which is a formal parameter does not result in modification of the corresponding actual parameter. An actual parameter may be a signal local to (declared and created in) the initiator. The corresponding formal parameter is identified as a signal name by its appearance in a \( \text{SIGNAL} \) declaration in the process description of the initiated process.

**SIGNALS**

Processes (and models and submodels) in SIML/I coordinate their activities via operations on signal elements. A signal element is a two-state variable which may be assigned either the value '1' ("set") or '0' ("reset") via \( \text{SET} \) or \( \text{RESET} \) statements. A process may suspend execution until a signal element \( S \) becomes set or reset via the statements
WAIT(S) or WAIT(\neg S), or QUEUE(S) or QUEUE(\neg S). Throughout this discussion, signal elements are, for the sake of brevity, referred to simply as signals. However, a signal actually should be viewed as a dynamic entity—a particular occurrence of a signal element's change of state; the medium should be distinguished from the message. Signal elements and their associated operations provide a general mechanism for process coordination; a signal element may represent a gate or a latch in a hardware-oriented model, or a table lock or semaphore in a software-oriented model.

Signals are defined, named, and created via SIGNAL declarations. Both simple (single) signals and sets (one-dimensional arrays) of signals may be defined, and signals may be defined in terms of logical combinations of other signals. A signal which is defined in terms of other signals is called a derived signal; a signal which is not defined in terms of other signals is a basic signal. In the declaration

\[
\text{SIGNAL A, B, C, D(6), E INITIALLY SET,} \\
F = \neg A \land B, G = \neg B \land \neg \neg C;
\]

A, B, C, D, and E are basic signals; F and G are derived signals. Basic signals are placed in the reset state when created (unless placed in the set state via an initial clause, as in the previous case of signal E). The initial state of derived signals is determined from the states of the signals on which they are defined. In the preceding declaration, F will be initially placed in the reset state and G will be placed in the set state. Only basic signals can be operated on by SET and RESET statements.

Signals declared in a model or submodel description are global to all process descriptions contained in the model or submodel description. Several instances of execution of a submodel description may be initiated; a unique set of signal elements is created (unless placed in the set state via an initial clause, as in the previous case of signal E). The initial state of derived signals is determined from the states of the signals on which they are defined. In the preceding declaration, F will be initially placed in the reset state and G will be placed in the set state. Only basic signals can be operated on by SET and RESET statements.

Signals declared in a model or submodel description are global to all process descriptions contained in the model or submodel description. Several instances of execution of a submodel description may be initiated; a unique set of signal elements is created (unless placed in the set state via an initial clause, as in the previous case of signal E). The initial state of derived signals is determined from the states of the signals on which they are defined. In the preceding declaration, F will be initially placed in the reset state and G will be placed in the set state. Only basic signals can be operated on by SET and RESET statements.

Signals are created at execution time, not at compile time. Thus, the dimension of a signal set may be determined by computations within the model, and signal creation can be made dependent on model input parameters.

SIGNAL OPERATIONS

A simple basic signal S is set or reset by the statements SET(S) or RESET(S); this may cause a change of state of a derived signal directly or indirectly defined on the basic signal. The \(i^{th}\) element of a basic signal set S can be set or reset by the statements SET(S(I)) or RESET(S(I)). The statement SET(S) or SET(S(I)), where S is a basic signal set, causes the set to be searched for an element in the reset state; if one is found, it is placed in the set state and, in the second form, the index of the element is assigned as the value of the variable I. (If no element is found in the reset state, the statement is ignored.) The RESET statement functions similarly.

A process may suspend execution until a basic or derived signal is set or reset via WAIT or QUEUE statements. Execution of one of these statements is called signal selection; processes suspended while waiting for a signal to change state are called selectors of that signal. Those processes which selected the signal via wait statements all are activated (placed in ready state) when the signal changes state. Of those processes which selected the signal via queue statements, only one is activated when the signal state change occurs. This one is chosen on the basis of priority; among equal priority selectors, the first to enter the queue is chosen. Signal selections may be a mixture of wait and queue selections; when the signal changes to the selected state, all waiting selectors and one queued selector are activated.

When a process executes a set or reset statement, it is placed on the ready list; any wait selectors of the signal are placed on the ready list next, followed by the queued selector (if one was activated). The next process to execute is then selected from the ready list on the basis of priority, as described earlier. If a process executes a wait statement and the selected signal already is in the specified state, it continues in execution. If it executes a queue statement and the selected signal already is in the specified state, it continues in execution only if there are no higher-priority processes queued on that state of the signal; otherwise, it is suspended and enqued.

A process may select the \(i^{th}\) element of a basic or derived signal set S by selection statements such as WAIT(S(I)) or QUEUE(S(I)); such signal set element selections are processed identically to simple signal selections. It is also possible to select the set itself. Associated with each signal set is a signal of the same name representing the state of the set; by definition, the state of a set is the logical sum of the states of its elements. Thus,

\[
S = S(1) \lor S(2) \lor \ldots \lor S(n)
\]

and

\[
\neg S = \neg S(1) \land \neg S(2) \land \ldots \land \neg S(n)
\]

For a signal set S, the statement WAIT(S) causes a selector to be suspended until some element of S becomes set; the statement WAIT(\neg S) causes a selector to be suspended until all elements of S are reset. Queue selections function similarly. Selection statements of the form WAIT(S(I)) can be used to obtain the index of the element whose state change caused the selector to be returned to execution.

SIGNAL EXPRESSIONS

A derived signal definition defines a signal (simple, set, or set element) in terms of a signal expression. The signals
in this expression may themselves be derived, so signals of arbitrary complexity may be constructed. Signal expressions take the logical sum-of-products form; parenthetical sub-expressions are not permitted, and operators are restricted to ‘&’ ("and") and ‘|’ ("or"). This choice of signal expression form was based on two considerations; representational efficiency and simulation efficiency. The applications of SIML/I in hardware-oriented modeling often include control mechanisms, but rarely go beyond that. For example, it may be desired to model the ingating to an adder, but simulation of the adder itself is not likely to be required. A study of control logic in various systems of interest indicated that the need for parenthetical sub-expressions and other operators (even exclusive or) arose infrequently. Restricting signal expressions to the sum-of-products form greatly expedited SIML/I implementation and permitted a very efficient structure for propagating signal state changes.

A derived signal is called a sink signal, and the signals on which it is defined are called source signals. Both sink and source signals may be either simple signals or signal sets: the various combinations govern the mapping between source and sink. Some examples of source/sink mappings are diagrammed in Figure 1; these diagrams show the mappings resulting from the following signal declarations:

- **SIGNAL A,B,C(2),D(3);**
- (a) **SIGNAL W(3): W(1)=A, W(2)=C(1), W(3)=D(1);**
- (b) **SIGNAL X=D|B&C;**
- (c) **SIGNAL Y(3)=C&D;**

When the sink signal is a single element (either a simple signal or a signal set element) and a source signal also is a single element, sink and source signals are directly mapped, as shown in Figure 1a. When the sink signal is a single element and a source signal is a set, the source signal (by definition) is taken to be the signal representing the set, and the mapping illustrated in Figure 1b results. When sink and source signals both are sets, they are mapped element-by-element up to the dimension of each set, as shown in Figure 1c. When the sink signal is a set and the source signal is a single element, the latter is mapped to each element of the set.

Sink/source mappings for single elements are determined by the axioms of logical arithmetic. Mappings involving sets were, to some extent, chosen on consideration of the functions needed to simply construct hardware, firmware, and software control structures for various systems. One missing capability which appears desirable is source signal set concatenation (currently, this can be effected only by writing a term-by-term definition for the sink signal).

**APPLICATIONS**

While the signal facilities of SIML/I were designed to extend system simulation capabilities nearer to the hardware realm, they provide a general means of process coordination, and support models over a range of levels of abstraction. In current applications, signal representations range from logic gates to complete CPUs. The following declarations represent the control logic shown in Figure 2.

```
SIGNAL TR_ENABLE = ENABLE&RTR|ENABLE&WTR|ENABLE&HTR;
SIGNAL TR_GATE = TR_ENABLE&CLOCK;
```

Figure 3 shows a block diagram of a queueing network model of a computer system; a SIML/I simulation program...
for this network is as follows:

MODEL QNM;
SIGNAL CPU(2) INITIALLY SET, IO(8) INITIALLY SET;
DO 1=1,32;
INITIATE JOB;
END;
HOLD(5000.);
TERMINATE;
PROCESS JOB;
DO WHILE(TIME<5000.);
QUEUE(CPU(I)); RESET(CPU(I));
HOLD(I.);
SET(CPU(I));
J=IRANDOM(1.8);
QUEUE(IO(J)); RESET(IO(J));
HOLD(EXPNTL(5.));
SET(IO(J));
END;
END PROCESS JOB;
END MODEL QNM;

In the foregoing example, there are 32 jobs circulating in the system. Job CPU times are constant and equal to one time unit, while I/O service times are exponentially distributed with a mean of 5 time units. When a job completes CPU service, it selects an I/O device and queues for it; when it completes I/O service, it queues for the set of CPUs. The set state of the CPU and I/O signals in this model corresponds to the nonbusy state.

Signal definitions are used to vary the level of detail of model components. The declarations below illustrate a signal defined at three different levels of detail.

(1) SIGNAL INTERRUPT;
(2) SIGNAL IO, FAULT, CHECK, TIMER:
   SIGNAL INTERRUPT=IO[FAULT][CHECK][TIMER];
(3) SIGNAL END_RD(8), END_WR(8),
    RD_FAULT, WR_FAULT;
    SIGNAL IO=END_RD[END_WR, FAULT=RD_FAULT][WR_FAULT];
    SIGNAL CHECK, TIMER;
    SIGNAL INTERRUPT=IO[FAULT][CHECK][TIMER];

Definition expansions of this form provide part of the mechanism for the vertical communication between components of different levels of detail in a multi-level model.

As an incidental note, SIML/I does not use the PL/I multitasking and event facilities for process and signal control; these functions are performed by the SIML/I run-time system.

ACKNOWLEDGMENTS

The author would like to acknowledge the fine work of Carol Realini and Allan Rhodes in developing the SIML/I language processor, and of Scott Spadafore in developing the interface between the SIML/I and PL/I run-time systems.

REFERENCES