SIMPL/1: A SIMULATION PROGRAMMING LANGUAGE

John W. Rettenmayer

University of Montana

ABSTRACT

SIMPL/1 is a set of SIMSCRIPT-like extensions to PL/1 which facilitate the direct expression of concepts inherent in discrete simulation programming. SIMPL/1 statements, which are intermixed with PL/1 statements in the source program and are syntactically compatible with PL/1, are translated into PL/1 code by a translator which is also written in PL/1. Completely dynamic treatment of entities, sets, and events is accomplished via PL/1 list processing using based variables. The paper also discusses some features unique to PL/1 itself which are particularly desirable and useful for simulation programming.

I. INTRODUCTION

SIMPL/1 (SIMulation PL/1 is a set of SIMSCRIPT-like extensions to the PL/1 language which facilitate the programming of discrete-time, event-oriented simulations. The development of SIMPL/1 was undertaken not to create a new simulation language, but to make available a basic simulation capability to educational institutions (and others) which do not have access to SIMSCRIPT or other commercially available languages, but which do have a PL/1 compiler.

The author was motivated in part by the experience of trying to teach simulation concepts and techniques to undergraduates. The concepts could, of course, be proferred to the class in lecture, but reinforcement of the concepts by implementation of a simulation project in a standard procedural language was unsatisfactory. These languages, such as FORTRAN, BASIC, or PL/1, do not allow direct expression of the notions inherent in discrete-time, event-oriented simulation, namely, sets of related entities and, in particular, events.

Acknowledgements

The author is indebted to the University of Alberta for the provision of computer time and a small grant for a research assistant. I am grateful for the assistance provided by Mr. Fred Rogers and for the fine programming of Mr. Ping Wong, whose time was provided by the Faculty of Commerce.

II. SIMULATION CONCEPTS

Typically, a simulation is designed to study the behavior of some system under various circumstances. The simulator (i.e., person conducting the study) often perceives the system as consisting of sets, or collections, of entities, each entity having various attributes, or characteristics. The sets are dynamic in that their membership changes over time; any given entity associates and disassociates with various other entities as it moves through the system. For example, if the system being studied is a carwash, the entities might be customers, each of whom might have the attributes (1) arrival time, (2) size of car, and (3) whether he wants a wax application. Each entity, in this case, will join, successively, two sets as it passes through the system, namely the queue of customers waiting to be washed and the queue of customers being washed.

The behavior of the system may be described in terms of the way in which the system state changes over time, where 'state' is a description of the entities and sets at any instant in time. Changes in state are caused by activities in which entities either are engaged or are acted upon. Most changes in state may be conceptualized as occurring either smoothly or abruptly. For example, the activity of turning up the flame of a gas stove certainly causes a rather smooth or continuous increase in the output of heat. However, it may be quite satisfactory to think of the heat state as having changed from low at the initiation of the activity to high at the conclusion of the activity. In the latter case, we may conceptually replace the activity by an event, which we might call 'change flame to high' and which is considered to occur at an instant in time. Changes in state are then thought to result from events and therefore may occur only at those instants in simulated time when events happen. Such is the approach taken in discrete simulation.

Thus, SIMPL/1, like other languages designed for event-oriented simulation programming, provides a timing mechanism whereby a 'clock' advances from the time of one event immediately to the time of the next event, as determined by a chronologically ordered list of event notices. Each time the clock is advanced, the timing mechanism calls a subroutine of the same name as that of the event, the purpose of which is to update the state description of the system in an appropriate way. Of course,

EXHIBIT 1

```
?DCL event EVENT ARGUMENTS (attr.1, . . . .,attr.m);
?CAUSE event AT expression PASS (expr.1, . . . .,expr.n);
?DCL entity ENTITY ATTRIBUTES (attribute list)
                  SET_MEMBERSHIP (member list)
                  SET_OWNERSHIP  (owned list);
?CREATE entity;
?CREATE entity CALLED name;
?DESTROY entity;
?DESTROY entity CALLED name;
?attribute(entity) = expression;
?attribute(entity CALLED name) = expression;
?variable = attribute(entity);
?variable = attribute(entity CALLED name);
?PUT LIST (entity);
?FILE entity LAST IN set(owner_entity);
?FILE entity CALLED name LAST IN set(owner_entity);
?REMOVE FIRST entity FROM set(owner_entity);
?IF set IS EMPTY THEN clause;
?IF set(owner) IS EMPTY THEN clause;
```

EXHIBIT 2

```
01  SIMPL/1:  DCL entity ATTRIBUTES (attr.1, . . .,attr.n)
02                  SET_MEMBERSHIP (membership list)
03                  SET_OWNERSHIP  (owned list);

04  ==>PL/1:  DCL 1 entity__STRUCTURE  BASED (entity_PTR),
05                 2 attr.1,
06                 2    .
07                 2    .
08                 2 attr.n,
09                 2 P_set  POINTER,              /* THERE WILL BE ONE SUCH */
10                 2 S_set  POINTER,              /* TRIPLET FOR EACH SET   */
11                 2 M_set  BIT(8),               /* MEMBERSHIP.            */

12                 2 F_set  POINTER,              /* ONE TRIPLET FOR EACH   */
13                 2 L_set  POINTER,              /* SET OWNED BY ENTITY.   */
14                 2 N_set  FIXED BINARY;         /* ; ONLY FOR LAST LINE.  */

15            DCL entity          POINTER,
16                set_OWNER       POINTER,        /* IF entity OWNS A set */
17                MEMBERSHIP_set  POINTER;        /* IF entity BELONGS TO set */

18  SIMPL/1:  CREATE entity CALLED name;

19  ==>PL/1:  ALLOCATE entity_STRUCTURE;
20            name = entity_PTR;          /* name MUST HAVE BEEN DECLARED POINTER */

21  SIMPL/1:  REMOVE FIRST entity FROM set(owner_entity);

22  ==>PL/1:  owner_entity_Ptr = owner_entity;
23            entity = owner_entity_STRUCTURE.F_set;
24            entity_PTR = owner_entity_STRUCTURE.F_set;
25            IF entity_PTR = NULL THEN
26               PUT LIST ('*** CANNOT REMOVE FROM EMPTY SET ***');
27            owner_entity_STRUCTURE.F_set = entity_STRUCTURE.S_set;
28            IF owner_entity_STRUCTURE.F_set = NULL THEN
29               owner_entity_STRUCTURE.L_set = NULL;
30            entity_STRUCTURE.M_set = '0'B;

31  SIMPL/1:  attribute(entity) = expression;

32  ==>PL/1:  entity -> entity_STRUCTURE.attribute = expression;
```

SIMPL/1 and other simulation programming languages also provide facilities for the natural and easy use of entities, attributes, and sets.

III. SIMPL/1 STATEMENTS

Insofar as possible, SIMPL/1 statements follow the syntax pattern of PL/1. This design choice was made not only to achieve some measure of linguistic aesthetics, but also to facilitate the assimilation of SIMPL/1 by the PL/1 programmer and to avoid confusing those who try to learn PL/1 and SIMPL/1 concurrently.

SIMPL/1 statements, which are intermixed with standard PL/1 statements in the source program, are translated into PL/1 statements by a pre-processor which is itself written in PL/1. No use is made of the pre-processor phase of the PL/1 compiler; the translation is a distinct step in a job control sense. The SIMPL/1 statements must begin with a '?' so that the translator can distinguish them from PL/1 statements. Any number of blanks may follow the '?'. Also, each SIMPL/1 statement must begin on a new line, although it may extend over several lines, and a PL/1 statement may not follow a SIMPL/1 statement on a line.

The extant set of SIMPL/1 statements is given in Exhibit 1 and, for a few of them, the corresponding target PL/1 statements generated by the translator are shown in Exhibit 2. In general, of course, each SIMPL/1 statement translates into several PL/1 statements. This is particularly true for the set operations, i.e. filing entities in sets and removing entities from sets.

The notational conventions used in Exhibits 1 and 2 are as follows. Words in all upper case letters are keywords and must be used as shown. Words in lower case letters represent identifiers to be chosen by the programmer. These identifiers are substituted verbatim ac litteratim in the corresponding target PL/1 statements.

For example, (see Exhibit 2) the SIMPL/1 statement,

```
?CREATE DOG CALLED FIDO;
```

would produce the following PL/1 statements:

```
ALLOCATE DOG_STRUCTURE;
FIDO = DOG_PTR;
```

Note that the identifiers, or variable names, DOG and FIDO, which are chosen by the programmer, are substituted directly into the target PL/1 statements.

IV. SIMPL/1 EXAMPLES

The following examples illustrate the uses of various SIMPL/1 statements. I extend a small apology for the usual but still unsettling fact that examples of a size and complexity suitable for a conference paper do not satisfactorily demonstrate the strength or usefulness of the language.

The program EXAMPLE_1, Exhibit 3, is a simulation of a single-server, single-channel queueing system, assuming Poisson arrival rates and exponential service time. This is a good problem for a simulation class since the results of the simulation may be compared with the well-known analytic solution. One real system which fits this model is a car wash where only one car can be washed at a time and the washing time, i.e., service time, varies with car size.

The program consists of a main procedure and five nested procedures, or subroutines. Procedure EXP generates exponentially distributed pseudo-random numbers for use as either inter-arrival times or service times. The distribution parameters may be different for the two times. EXP in turn calls on IRANDOM, which generates pseudo-random values from a uniform 0-1 distribution.

Lines 5-19 are standard PL/1 declarations. They give the internal storage type and initial values of some of the identifiers. Lines 20-22 are SIMPL/1 declaration statements which declare 3 events -- ARRIVAL, SERVICE, and STOPSIM. The state changes corresponding to the occurrence of these events are recorded by the procedures of the same names. In this first example, the notions of entities, attributes, and sets are not utilized.

The simulation is quite straight-forward, particularly with regard to the scheduling of events, which is the aspect of primary interest. The GET DATA statement in line 23 gives the user the opportunity to input any of the parameters; there are default initial values for those which are not input by the user (more will be said about this later). $TIME (line 25) is the identifier used by SIMPL/1 to denote the simulated clock value; it is automatically updated to the time of each current event as the simulation progresses. In line 27, an event notice for the event STOPSIM is placed in the event list to be executed at time SIM_PERIOD, which is the length of the simulation run. The very next statement (line 28) inserts in the event list a notice for an ARRIVAL event to occur at time zero (0).

The ?START SIMULATION statement in line 30 essentially transfers control to the SIMPL/1 control procedure, which in turn calls the event procedures according to the chronological order of the event notices in the event list. Thereafter, the simulation consists of a series of procedure calls made by the control routine.

At this point ($TIME=0) there are two event notices in the list -- one to stop the simulation at the lapse of SIM_PERIOD time units on the simulated clock, and one to cause an ARRIVAL event at time 0. Hence, the procedure ARRIVAL will be called immediately. It first generates another arrival to occur at some future point in simulated time, i.e., an event notice is inserted in the list in the proper chronological position. If the number of customers already in the queue, which includes the customer being serviced, is zero, then service activity for the customer is initiated; otherwise, the just-arrived customer is placed in the queue (line 40).

EXHIBIT 3

```
01  /* SIMPL/1 PROGRAM USING EVENTS */

02  EXAMPLE_1:  PROCEDURE OPTIONS(MAIN);

03  /* THIS IS A SIMULATION OF A SINGLE SERVER - */
04  /* SINGLE CHANNEL QUEUEING SYSTEM.          */

05  DCL EXP   ENTRY (FIXED BIN(31),FIXED BIN(15)) RETURNS(FIXED BIN(15)),
06      IRANDOM ENTRY (FIXED BIN(31), FLOAT);
07  DCL (#_QUEUE,                /* LENGTH OF QUEUE, INCLUDING
                                     THE UNIT BEING SERVICED   */
08       TIDT,                   /* TOTAL IDLE TIME           */
09       LAST_EVENT_TIME,
10       #_ARRIVALS,
11       TQL,                    /*TOTAL QUEUE LENGTH         */
12       #_SAMPLES )          FIXED BIN(15),
13       AQL         FLOAT  INITIAL(0), /* AVERAGE QUEUE LENGTH*/
14       SIM_PERIOD FIXED BIN(15) INITIAL(20);
15  DCL (SEED1 INIT(44444),
16       SEED2 INIT(33333) )  FIXED BIN(31),
17      (MAT   INIT(4),          /* MEAN INTER-ARRIVAL TIME   */
18       MST   INIT(6)           /* MEAN SERVICE TIME         */
19                          )  FIXED BIN(15);

20  ?DCL ARRIVAL EVENT;
21  ?DCL SERVICE EVENT;
22  ?DCL STOPSIM EVENT;

23  BEGIN:  GET DATA (MAT, MST, SIM_PERIOD, SEED1, SEED2) COPY;
24          IF MAT = 0 THEN GO TO FINI;
25          $TIME = 0;           /* $TIME IS THE SIMULATED CLOCK. */
26          #_QUEUE, TIDT, TWT, TQL, #_SAMPLES,#_ARRIVALS = 0;
27  ?       CAUSE STOPSIM AT SIM_PERIOD;
28  ?       CAUSE ARRIVAL AT 0;
29          LAST_EVENT_TIME = 0;
30  ?       START SIMULATION;        /* START THE CLOCK MECHANISM */

31  ARRIVAL: PROCEDURE;
32          #_ARRIVALS = #_ARRIVALS + 1;
33  ?       CAUSE ARRIVAL AT ($TIME + EXP(SEED1,MAT));
34          IF #_QUEUE = 0 THEN DO;
35                              TIDT = TIDT + ($TIME - LAST_EVENT_TIME);
36                            ? CAUSE SERVICE AT ($TIME + EXP(SEED2,MST));
37                              END;
38          TQL = TQL + #_QUEUE;
39          #_SAMPLES = #_SAMPLES + 1;
40          #_QUEUE = #_QUEUE + 1;
41          LAST_EVENT_TIME = $TIME;
42          RETURN;
43          END ARRIVAL;
```

```
44  SERVICE:  PROCEDURE;              /* END OF SERVICE ACTIVITY */

45          TQL = TQL + #_QUEUE;
46          #_SAMPLES = #_SAMPLES + 1;
47          #_QUEUE = #_QUEUE - 1;
48          IF #_QUEUE ¬= 0 THEN DO;    /* ¬ MEANS 'NOT' */
49                               ? CAUSE SERVICE AT $TIME + EXP(SEED2,MST);
50                                 END;
51          LAST_EVENT_TIME = $TIME;
52          RETURN;
53          END SERVICE;

54  EXP:  PROCEDURE (SEED, MEAN_TIME)  RETURNS (FIXED BIN);
55  /* DRAWS AN OBSERVATION FROM EXPONENTIAL DISTRIBUTION WITH MEAN 'MEAN_TIME' */
56  DCL MEAN_TIME  FIXED BIN(15),
57      SEED       FIXED BIN(31),
58      X          FIXED BIN(15)  STATIC,
59      RNUM       FLOAT          STATIC;

60          CALL IRANDOM (SEED, RNUM);
61          X = -MEAN_TIME * LOG(RNUM);
62          RETURN (X);
63          END EXP;

64      (NOFIXEDOVERFLOW):
65  IRANDOM:  PROCEDURE (SEED, RNUM);  /* GENERATES A PSEUDO-RANDOM */
66  DCL SEED  FIXED BIN(31),           /* NUMBER FROM A UNIFORM 0-1 */
67      RNUM  FLOAT;                   /* DISTRIBUTION.             */

68          SEED = SEED * 65539;
69          IF SEED<0 THEN SEED = SEED + 2147483647 + 1;
70          RNUM = SEED * 0.4656613E-9;
71          RETURN;
72          END IRANDOM;

73  STOPSIM:  PROCEDURE;

74          AQL = TQL / #_SAMPLES;
75          UTILIZATION = 1 - (TIDT / SIM_PERIOD);
76          PUT SKIP DATA (#_ARRIVALS, #_QUEUE, #_SAMPLES);
77          PUT SKIP DATA (TQL, AQL);
78          PUT SKIP DATA (UTILIZATION);
79          PUT SKIP(4);
80          GO TO BEGIN;
81          END STOPSIM;
82  FINI:  END EXAMPLE_1;
```

Initiating the service activity means that an event marking the end of the service activity must be scheduled at some future time (line 36). The other statements in the procedure are for the purpose of updating the state description of the system.

ARRIVAL and SERVICE events are intermixed in simulated time in an order randomly determined by the exponential distributions of the inter-arrival and service times, respectively. The procedure EXP draws a random sample from one of two different exponential distributions, depending on the value of its second parameter.

The STOPSIM event is scheduled for a time later than SIM_PERIOD, but they will be ignored. Thus, the simulation may end with a customer being serviced but never finishing service. In the STOPSIM procedure we calculate the average queue length and the facility utilization factor and print them along with some of the cumulative statistics. Then we branch back to the GET DATA statement in line 23 to possibly perform another iteration with one or more different parameters.

EXAMPLE_2 (Exhibit 4) is a simulation of the same single-server, single channel queueing system as was treated in the preceding program. It is essentially the same as EXAMPLE_1, but, in addition, it illustrates the use of entities and sets. Hopefully, it also demonstrates that these concepts allow a more natural expression of the processes under study.

In line 22, the entity 'customer' is declared to have a single attribute, 'arrival_time' (which is stored as a fixed binary number), and to belong to two sets, 'queue' and 'bay'. Unlike the previous example, we now consider the waiting queue to be exclusive of the entity which is being serviced. Note, too, that set membership is a permissive concept; the entity may belong to either set, both sets, or neither set at any point in simulated time. Line 24 declares the entity 'system', whose only purpose is to be the owner of the two sets. Line 26 then creates, or brings into existence, the 'system' and that entity stays in existence for the remainder of the simulation.

In contrast to the unchanging nature of the entity 'system', there may be several customer entities in existence at the same time and each one will be created and later destroyed as the simulation progresses. As one would expect, each ARRIVAL event occasions the creation of a customer (line 38), and each SERVICE event entails the destruction of the customer who has been serviced (line 59).

Immediately after creating the customer, the time of his arrival is stored in his attribute 'arrival_time' (line 39). In line 34, we check to see if the bay of the system is empty; if so, we file the newly arrived, i.e., created, customer in the bay and cause a service event to occur at some future time. If the service bay is not empty, we file the customer in the last position of the queue and increment the queue length counter by 1. In either case, we add the queue length to the cumulative queue length for later use in calculating the average queue length (line 50).

Because each customer entity carries its arrival time as an attribute, the calculation of the average waiting time (lines 57-58, 87) is direct and exact, in contrast to the circumspect method that would be required in the first example. Similarly, the use of sets allows a conceptually explicit 'movement' of entities from the waiting queue to the service bay rather than just a numerical accounting for them. We conjecture that this explicitness has pedagogical advantages and is probably less error prone than implicit methods of monitoring the state of the system.

V. DESIGN OF THE TRANSLATOR

The SIMPL/1 translator is written in PL/1 and makes extensive use of the PL/1 character manipulation facilities. Essentially, the SIMPL/1 source program is input to the translator as a single character string and the PL/1 target code is output as a character string. PL/1 statements in the input string are transferred unaltered to the output string. SIMPL/1 statements in the input string are replaced by the appropriate PL/1 statement(s), into which the programmer-selected identifiers are substituted, as indicated in Exhibit 2. This translation process requires only a single pass through the source string.

Actually, the first action taken by the translator is the insertion in the output string of a number of PL/1 statements immediately following the PROCEDURE OPTIONS (MAIN) statement. These statements provide the event list structure, the structure for event notices, the procedure (subroutine) for filing notices in the event list, initialization of variables, and so forth.

SIMPL/1 depends heavily on the list processing facilities of PL/1. All event notices and entities are BASED structures. Based variables and structures are allocated storage space dynamically by an ALLOCATE statement in the program. The storage location which is obtained by a particular allocation is assigned as a value to a particular POINTER variable. The storage location obtained by a subsequent allocation also will be assigned to that pointer variable, so the programmer is responsible for 'remembering' the location if he will want to use the variable after ensuing allocations. That is, he must assign the value of the particular pointer variable to another pointer variable or to a member of a pointer array, where it is available for subsequent use. In the last line of Exhibit 2, the variable 'entity' is a pointer variable whose value is the location of the variable 'entity_STRUCTURE.attribute'. The '->' symbol means that entity 'points to' the location of entity_STRUCTURE.attribute. In line 4, entity_PTR is the particular pointer variable to which is assigned the location obtained whenever the statement ALLOCATE entity_STRUCTURE (line 19) is executed.

Obviously, the basic capabilities for dynamic simulation programming are contained in PL/1. However, the use of based variables in programs tends to become rather complicated and error prone, particularly for novice programmers. SIMPL/1 provides the dynamic mechanisms for the programmer so that he can concentrate on describing in a rather direct way the system he is modeling.

EXHIBIT 4

```
01     /* SIMPL/1 PROGRAM USING ENTITIES AND SETS AS WELL AS EVENTS */

02  EXAMPLE_2:  PROCEDURE OPTIONS(MAIN);

03  DCL (TIDT,
04       TWT,                  /* TOTAL WAITING TIME, INCL. SERVICE */
05       TQL,                  /* TOTAL (CUMULATIVE) QUEUE LENGTH */
06       LAST_EVENT_TIME,
07       #_QUEUE,              /* QUEUE LENGTH, EXCLUDING SERVICE BAY */
08       #_ARRIVALS,
09       #_SAMPLES  ) INIT(0)  FIXED BIN(15),
10      (AWT,                  /* AVERAGE WAITING TIME */
11       AQL)          FLOAT,  /* AVERAGE QUEUE LENGTH */

12       SIM_PERIOD INIT(20)  FIXED BIN(15),
13      (SEED1  INIT(44444),
14       SEED2  INIT(33333) ) FIXED BIN(31),
15      (MAT    INIT(4),                    /* MEAN INTER-ARRIVAL TIME */
16       MST    INIT(6)     ) FIXED BIN(15); /* MEAN SERVICE TIME */
17  DCL  EXP ENTRY (FIXED BIN(31),FIXED BIN(15)) RETURNS(FIXED BIN(15)),
18       IRANDOM ENTRY (FIXED BIN(31), FLOAT);

19  ?DCL ARRIVAL EVENT;
20  ?DCL SERVICE EVENT;
21  ?DCL STOPSIM EVENT;

22  ?DCL CUSTOMER ENTITY ATTRIBUTES (ARRIVAL_TIME FIXED BIN(15))
23                       SET_MEMBERSHIP (QUEUE,BAY);
24  ?DCL SYSTEM ENTITY SET_OWNERSHIP (QUEUE,BAY);

25   BEGIN:
26  ?       CREATE SYSTEM;

27          GET DATA (MAT, MST, SIM_PERIOD, SEED1, SEED2) COPY;
28          IF MAT = 0 THEN GO TO FINI;
29          $TIME = 0;
30          #_QUEUE, #_ARRIVALS, #_SAMPLES, TQL, TWT, TIDT = 0;
31  ?       CAUSE STOPSIM AT SIM_PERIOD;
32  ?       CAUSE ARRIVAL AT $TIME;
33          LAST_EVENT_TIME = $TIME;
34  ?       START SIMULATION;

35  ARRIVAL:  PROCEDURE;

36          #_ARRIVALS = #_ARRIVALS + 1;
37  ?       CAUSE ARRIVAL AT $TIME + EXP(SEED1,MAT);
38  ?       CREATE CUSTOMER;
39  ?       ARRIVAL_TIME(CUSTOMER) = $TIME;
40  ?       IF BAY(SYSTEM) IS EMPTY
41             THEN DO;
42                  TIDT = TIDT + ($TIME - LAST_EVENT_TIME);
43                ?  FILE CUSTOMER LAST IN BAY(SYSTEM);
44                ?  CAUSE SERVICE AT $TIME + EXP(SEED2,MST);
45                  END;
46             ELSE DO;
47                ? FILE CUSTOMER LAST IN QUEUE(SYSTEM);
48                  #_QUEUE = #_QUEUE + 1;
49                  END;
```

```
50          TQL = TQL + #_QUEUE;
51          #_SAMPLES = #_SAMPLES + 1;
52          LAST_EVENT_TIME = $TIME;
53          RETURN;
54          END ARRIVAL;

55  SERVICE:  PROCEDURE;

56  ?       REMOVE FIRST CUSTOMER FROM BAY(SYSTEM);
57  ?       X = ARRIVAL_TIME(CUSTOMER);
58          TWT = TWT + ($TIME - X);
59  ?       DESTROY CUSTOMER;
60          TQL = TQL + #_QUEUE;
61          #_SAMPLES = #_SAMPLES + 1;
62  ?       IF QUEUE(SYSTEM) IS EMPTY THEN;
63             ELSE DO;
64                ? REMOVE FIRST CUSTOMER FROM QUEUE(SYSTEM);
65                  #_QUEUE = #_QUEUE - 1;
66                ? FILE CUSTOMER LAST IN BAY(SYSTEM);
67                ? CAUSE SERVICE AT $TIME + EXP(SEED2,MST);
68                  END;
69          LAST_EVENT_TIME = $TIME;
70          RETURN;
71          END SERVICE;

72  EXP:  PROCEDURE (SEED, MEAN_TIME);  /* SEE EXAMPLE_1 FOR A         */
73  IRANDOM:  PROCEDURE (SEED, RNUM);   /* LISTING OF THESE PROCEDURES */

74  STOPSIM:  PROCEDURE;
75          AQL = TQL / #_SAMPLES;
76        ? IF BAY(SYSTEM) IS EMPTY THEN;
77             ELSE DO;
78                ? REMOVE FIRST CUSTOMER FROM BAY(SYSTEM);
79                ? X = ARRIVAL_TIME(CUSTOMER);
80                  TWT = TWT + ($TIME - X);
81                  END;
82          DO I = 1 TO #_QUEUE;
83           ? REMOVE FIRST CUSTOMER FROM QUEUE(SYSTEM);
84           ? X = ARRIVAL_TIME(CUSTOMER);
85             TWT = TWT + ($TIME - X);
86          END;
87          AWT = TWT / #_ARRIVALS;
88          UTILIZATION = 1 - TIDT / SIM_PERIOD;
89          PUT SKIP DATA (#_ARRIVALS, #_QUEUE, #_SAMPLES);
90          PUT SKIP DATA (TQL, AQL);
91          PUT SKIP DATA (TWT, AWT);
92          PUT SKIP DATA (UTILIZATION);
93          PUT SKIP(4);
94        ? DESTROY SYSTEM;
95          GO TO BEGIN;
96          END STOPSIM;

97  FINI:   END EXAMPLE_2;
```

VI. GENERAL OBSERVATIONS CONCERNING SIMPL/1

SIMPL/1 does not offer many of the amenities which are included in other simulation languages such as SIMSCRIPT or GPSS. For example, it includes no built-in facilities for statistical distribution generation, gathering of descriptive information, or report writing. However, it does provide those mechanisms which are universally required for discrete-time, event-oriented simulation programming, which are also the most difficult for individual programmers to construct for themselves. This is particularly true for the non-professional programmer or student who is receiving his first exposure to computer simulation.

A few technical observations might be of interest to those who are familiar with SIMPL/1's progenitor, SIMSCRIPT. SIMPL/1 utilizes a single heterogenous event list rather than a separate list for each event type. Also, event notices do not have attributes; thus, all notices have the same form. These two design factors make the processing of the event list simpler and, at least for small programs, more efficient (less overhead). However, another consequence is that events which have the same time of occurrence are executed in the order in which they appear in the event list, i.e., no tie-breaking is done. For example, if an arrival event and a service event are scheduled for the same time, the programmer has no way of specifying that one type of event takes precedence over the other type. Such a capability could be added, but it does not currently exist.

Still another consequence of the single event list is that the list is likely to become longer than any one list in a multi-list system. The expected time required for insertion of an event notice in the list is, of course, proportional to the length of the list. No measurement of the relative efficiency of SIMPL/1 and other simulation languages has been attempted, but one would expect that the single event list mechanism of SIMPL/1 would be at least as efficient as other types for small simulations with only a few different types of events. As the number of different kinds of events increases, we would expect the number of event notices filed in the list at any given time to increase also.

Thus the factor which is critical to the efficiency of SIMPL/1 is the length of the event list, which in turn is a function of the number of different event types and the degree of advanced scheduling of those events. For instance, in the examples discussed in this paper, there will be a maximum of one service event notice and one arrival event notice in the event list at any instant during the simulation. For such cases where the length of the list has a low limit (not necessarily as low as 2, of course), SIMPL/1 should suffer no disadvantage in comparative efficiency.

It is expected that all subroutines will be nested within the main procedure. $TIME and other identifiers used by the SIMPL/1-provided code are global within the main procedure, but they do not have the EXTERNAL attribute. Hence, the programmer should recognize that use of external procedures in addition to the main procedure will require special care. Some users might want to alter the translator to accommodate external procedures.

$TIME and other identifiers used in the SIMPL/1 mechanisms have a $ as the initial character of the identifier. They may be used as variables by the programmer; however, care should be exercised in doing so, and they should very seldom, if ever, be assigned values by the user. In particular, the declaration of these variables within an internal procedure would render their scope local to that procedure--an action of unlikely benefit.

VII. PL/1 SIMULATION FEATURES

As mentioned earlier, PL/1 itself offers some features which are quite helpful for simulation programming. With Data-directed input (GET DATA) the user may choose which and how many data items to input. Each data item in the input stream is self-identifying, i.e., it may be of the form X=3, which inputs a value of 3 for the variable X. Those variables for which no values are input maintain the values they had prior to execution of the GET DATA statement. If the GET DATA is included in a loop, several repetitions of the simulation may be made easily since the user needs to input only the parameters to be changed from case to case. Furthermore, the COPY verb in the GET DATA statement will produce on the output file a copy of the input stream so that the simulation output contains a record of which variables were changed from case to case.

The CHECK facility is normally used to print out the value of a variable in the check list each time its value changes. Additionally, the programmer may, via the ON CHECK statement, specify other actions to be taken when the value of any variable is altered. Using this feature, the programmer can make the execution of a subroutine, which may represent an event, conditional upon either any alteration of a particular variable or the attainment by that variable of a certain value (to be tested in the ON CHECK unit).

VIII. CONCLUDING REMARKS

It should be noted that the SIMPL/1 statement set shown in this paper certainly should not be considered complete; other statements can and should be added. Extension of the translator should be relatively easy since much of the existing code could be used either directly or as a pattern for the additional code. Unfortunately, the author no longer has access to a PL/1 processor and cannot carry out further development himself. However, copies of the translator may be obtained by request to the author.

BIBLIOGRAPHY

1. International Business Machines Corporation, PL/1 Reference Manual, Form No. C28-8201, 1968.

2. Kiviat, P.J., R. Villanueva, and H. M. Markowitz, The SIMSCRIPT II Programming Language, Prentice-Hall, 1969.

3. Naylor, T.H., J.L. Balintfy, D. C. Burdick, and K. Chu, Computer Simulation Techniques, John Wiley & Sons, Inc., New York, 1966.