SOL-20

BY

GILBERT J. HANSEN

Carnegie Institute of Technology
April 22, 1965

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-146)

ACKNOWLEDGMENTS

The author thanks David M. Blocher, Janet W. Fierst, Richard B. Grove and Carol H. Thompson for taking time to explain the internal workings of Algol-20 and for their suggestions and patience in helping to debug the system.

Finally, thanks goes to Dr. Alan J. Perlis for his many fine suggestions and guidance in developing this implementation of SOL.

CONTENTS

* The Knuth and McNeley Papers from THE IEEE TRANSACTIONS, August, 1964 are reprinted with permission.

Chapter 1
Introduction

This manual is a supplement to the original article A FORMAL DEFINITION OF SOL by Knuth and McNeley. The version of SOL described here, known as SOL-20, was implemented by procedures written in Algol-20 and G-20 machine language. It is the purpose of this documentation to describe in detail the exact differences and changes in syntax between SOL and SOL-20.

With some limitations, the full power of ALGOL* is available for programming in SOL-20. A SOL-20 program is written using SOL-20 system procedures which implement SOL declarations, expressions, relations and statements.

The sample problem given by Knuth and McNeley has been re-written in SOL-20 and is included in Chapter 7. Sample problem output for 15 time units of simulation is also appended. The results differ from those of Knuth and McNeley only because a different random number generator was used.

*ALGOL, as used hereafter, will refer to ALGOL-20, the local C.I.T. version of the international language ALGOL-60.

Chapter 2
Differences in SOL-20

This section describes, in detail, the differences and syntax changes between the originally-proposed SOL and the version implemented as SOL-20. The format and organization is that as given in the formal Knuth and McNeley document on the definition of SOL. All examples are from the sample program appended.

Appendix I contains a complete list of all the differences between SOL and SOL-20 discussed below.

I. GENERAL DESCRIPTION

II. SYNTAX AND SEMANTICS OF SOL

A. Identifiers and Constants

All rules for writing identifiers and constants in ALGOL-20 are applicable.

Since a process is a block, the same identifier can be used in different processes with different meanings. The ALGOL rules for local and global variables apply.

B. Declarations

⟨declared item⟩ ::= ⟨identifier⟩

⟨variable declaration⟩ ::= ⟨all variable declarations, e.g., half, boolean, logic, and array that can be used in ALGOL-20 are permissible⟩

⟨facility declaration⟩ ::= ⟨variable declaration⟩

⟨store declaration⟩ ::= ⟨variable declaration⟩

⟨table declaration⟩ ::= half array ⟨identifier⟩ [1:2,0: ⟨number⟩ , 1: ⟨number⟩]

The first ⟨number⟩ represents the maximum length of the table while the second ⟨number⟩ represents the number of different tables under the same identifier.

Examples:

```
line 6: real array TU[1:6],SB[1:3] ;     ! facilities
line 2: real LINE,COMPUTER;      ! facilities
line 7: real array QUEUE[1:6];      ! stores
line 9: half array TAB [1:2,0:27,1:6];     ! tables
```

There is no monitor declaration.

When an arithmetic expression is assigned to an integer variable, the value assigned is rounded or truncated to the nearest integer, depending on whether the "←" or " := " is used.

Statistics for stores, facilities and tables are obtained by special procedure calls. See Chapter 3 on statements.

The size of a store is declared by the procedure call

STORE(⟨store identifier⟩ , ⟨constant⟩);

where the constant represents the capacity of the indicated store.

Likewise, the bounds on the histogram for a table are given by a special procedure call

TABLE(⟨number⟩ , ⟨table identifier⟩ , ⟨number⟩ , ⟨number⟩ , ⟨number⟩ , ⟨number⟩);

The first ⟨number⟩ represents the table number, i.e., the desired table specified by the table identifier. The other three ⟨number⟩'s give the starting point for histogram intervals, the increment between intervals, and the highest value, respectively.

Example:

```
lines 13-16:  for I ← 1 step 1 until 6 do begin
                  STORE(QUEUE[I],10);   ! declare
                                           stores
                  TABLE(1,TAB,2000,500,15000);
                                        ! declare tables
              end;
```

It is up to the user to ensure that the number of histogram intervals does not exceed the size of the table.

C. Expressions and Relations

Listed below are the SOL relations and expressions with the corresponding SOL-20 procedure specifications.

The format is:

SOL construction(s):

SOL-20 procedure specification

followed by comments when needed.

1. expressions

a. time is declared by the system and, therefore, is a global variable.

Example:

line 41: START.TIME ← TIME;

b. $(e_1,e_2,\ldots,e_n)$:

half procedure RAND(N,E); value N; half N;
half array E;

N represents the number of inputs e_i, or equivalently the dimension of the array E. E contains the values of the e_i's.

Examples:
line 37: MESSAGE.TYPE ← RAND(10,TYPE);
line 97: WAIT(RAND(10,WAITS));

c. normal(<expression> , <expression>):

real procedure NORMAL(M,S); value M,S; real M,S;

d. exponential(<expression>):

real procedure EXPONENTIAL(M); value M; real M;

e. poisson(<expression>):

integer procedure POISSON(M); value M; real M;

f. geometric(<expression>):

integer procedure GEOMETRIC(M); value M; real M;

g. random:

real procedure RANDOM(A,B,C); value A,B,C;
real A,B,C;

RANDOM represents a random number between A and B (A< B) where A and B may be any real numbers. C is always zero.

h. e_1:e_2 :

integer procedure UNIFORM(E1,E2); value E1,E2;
real E1,E2;

Examples:
line 36: Q ←UNIFORM(1,6);
line 40: WAIT(UNIFORM(6000,8000));

2. relations

a. ⟨facility name⟩ busy, ⟨facility name⟩ not busy:

boolean procedure BUSY(FACILITY); real FACILITY;

The value of this function is true if the facility is busy and false if it is not busy.

Example:
line 59 : if BUSY(SB[S]) then begin

b. ⟨store name⟩ full, ⟨store name⟩ not full:

boolean procedure FULL(STORE); real STORE;

If the facility is full, the value of the function is true, otherwise it is false.

c. ⟨store name⟩ empty, ⟨store name⟩ not empty:

boolean procedure EMPTY(STORE); real STORE;

If the store is empty, the value of the function is true, otherwise it is false.

d. pr(⟨expression⟩):

boolean procedure PR(PROB); value PROB; real PROB:

The value of the function is true with probability given by the value of the variable PROB.

Example:
line 65: if PR(0.02) then begin

III. STATEMENTS

A. Processes

A process is written as an ALGOL block, each separate process being a self-contained block. Blocks may not be nested. The name of a process is a statement label occurring at the head of the block, e.g.:

⟨process identifier⟩ : begin ⟨process declaration list⟩ ; ⟨statement list⟩
end;

Example: See USERS process, lines 27-52 in the sample program.

Processes are declared by the switch statement PROCESSES, which appears at the very beginning of the program, i.e.,

switch PROCESSES := *⟨process identifier⟩*;

(NOTE: the abbreviation *⟨A⟩* means "a list of ⟨A⟩ ", i.e., *⟨A⟩* ::= ⟨A⟩ | *⟨A⟩*, ⟨A⟩).

All the rules for a label given in ALGOL-20 obviously apply.

Example:

line 3 : switch PROCESSES := CONTROL,USERS,PBU,OTHER.PBUS;

Procedures may not be declared in a process if the procedures contain the SOL statements wait, seize,enter or wait until.

The for clause may be used, and can only be of the form

for ⟨simple variable⟩ ::= ⟨for list element⟩ do

Multiple for lists may be used if the for statement does not contain any of the SOL statements just mentioned in its scope.

B. Labels

C. Creation of Transactions

All labels representing the name of transactions must be declared by a label declaration either at the start of a program or in the declarations of the process in which they occur, i.e.,

label * ⟨transaction label⟩*;

Example:

line 11 : label START,SCAN,COMPUTATION,COMPUTE;

A new copy of a transaction is created by the procedure call

NEW.TRANSACTION(⟨label⟩);

Example:

line 55 : NEW.TRANSACTION(SCAN);

D. Disappearance of Transaction

Transactions are cancelled by a call to the procedure CANCEL. The end of a process does not imply a cancel statement; cancel statements must be explicitly written.

Example:

line 86 : CANCEL;

E. Replacement Statements

In a replacement statement, if the variable is an integer variable, the expression is truncated or rounded depending on whether a "←" or a " := " is used. A direct assignment to a store will produce incorrect results. This error will not be detected.

F. Priority

The integer variable PRIORITY is declared by the system, thus making it a global variable and not a variable local to each process. It is initially zero and if a transaction is to have a different priority, then

PRIORITY← ⟨expression⟩ ;

must precede each new transaction statement. It will retain this assigned value until changed. The priority must be between 0 and $2^{17} - 1$.

Assigning a priority to a process, which is equivalent to assigning a priority to the first transaction occurring in the process, must be done by special means. The programmer must declare in his program the integer array PROC.PRIO[1:n] and then initialize it. The elements of this vector are in a 1-1 correspondence with the process identifiers as written in the PROCESSES switch statement.

G. Wait Statements

procedure WAIT(WAIT.TIME); value WAIT.TIME; half WAIT.TIME;

Example:

```
line 59 :  WAIT(5);
line 40 :  WAIT(UNIFORM(6000,8000);
```

H. Wait-Until Statements

procedure WAIT.UNTIL(P); boolean P;

The relation P may be either a relation or a boolean procedure which is parameterless.

Example:

```
line 46:   WAIT.UNTIL(TUSTATE [Q] = 0);
```

I. Enter Statements

procedure ENTER(STORE,UNITS); value UNITS; half UNITS; real STORE;

STORE represents the store name which may be a simple identifier or an array element. UNITS represents the number of units requested of the store and is rounded to the nearest integer.

Example:

line 36: ENTER(QUEUE [Q],1);

J. Leave Statements

procedure LEAVE(STORE,UNITS); value UNITS; half UNITS; real STORE;

STORE and UNITS are the same as in the enter statement.

Example:

line 47: LEAVE(QUEUE [Q],1);

K. Seize Statements

procedure SEIZE(FACILITY,CONTROL.STRENGTH); value CONTROL.STRENGTH;
half CONTROL.STRENGTH; real FACILITY;

The control strength may be any integer between 0 and $2^{22} - 1$.

Examples:

line 38: SEIZE(TU [Q],0);
line 58: SEIZE(LINE,0);

L. Release Statements

procedure RELEASE(FACILITY); real FACILITY;

An error is given if the transaction releasing the facility is not controlling it.

Examples:

line 47: RELEASE(TU [Q]);
line 68: RELEASE(LINE);

M. Go To Statement

The form go to (*⟨label⟩*), ⟨expression⟩ must be written as a switch statement. Go to statements can only be used to transfer to another point in a transaction. It is not possible to transfer control to another process or to another transaction within the same process.

Examples:

line 35: go to ORIGIN;
line 84: if WORDS > 0 then go to OUTPT;

N. Compound Statements

O. Conditional Statements

P. Tabulate Statements

```
procedure TABULATE(EXPRESSION,Q,TABLE); value EXPRESSION,Q;
                half EXPRESSION; integer Q; half array TABLE;
```

Tables are contained in 3-dimensional arrays as mentioned in declarations, section II. Q represents the particular page of the array, i.e., the table in question. TABLE is the name of the 3-dimensional array. The value of EXPRESSION represents the variable to be recorded as a statistical observation in the Qth page of TABLE.

Example:

```
line 48:  TABULATE((TIME-START.TIME),Q,TAB);
```

Q. Output Statements

All output must be done in ALGOL-20 using NAME and PRINT statements.

Example:

```
lines 49-50:  NAME(Q,TIME);
              PRINT (<20B,'TU',20,1B,'RECEIVES REPLY AT
                    TIME',1B,7D,1B,E>);
```

R. Stop Statements

```
procedure STOP;
```

A call to this procedure terminates the simulation immediately. Statistics for all stores, tables and facilities are not automatically outputted by the system. The information is obtained by special procedure calls. This code must start at the label RESULTS and occurs in the outermost block (see lines 100-113 of sample program).

Example:

```
line 25 : STOP;
```

1. statistics for facilities

```
procedure PRT.FAC(INDEX,FACILITY,NAME); value INDEX;
            integer INDEX; real FACILITY; string NAME;
```

If INDEX is 0, the facility is not an array element; if it is ≠ 0, then INDEX represents the element in the vector. NAME is a string with a maximum length of 4. This name will appear on the output listing opposite the corresponding statistic.

Examples:

lines 102-103: `for I ← 1 step 1 until 6 do PRT.FAC(I,TU[I],'␣␣TU');`

line 106: `PRT.FAC(0,LINE,'LINE');`

2. statistics for stores

`procedure PRT.ST(INDEX,STORE,NAME); value INDEX; integer INDEX; real STORE; string NAME;`

INDEX and NAME have the same meanings as for 1 except now they apply to stores.

Example:

lines 109-110: `for I ← 1 step 1 until 6 do PRT.ST(I,QUEUE[I],'␣QUE');`

3. statistics for tables

`procedure PRT.TAB(Q,TABLE,NAME); value Q, integer Q; half array TABLE; string NAME;`

The table number Q indicates which table in the 3-dimensional array given by TABLE is to be used to calculate the statistical information. The string NAME is again of maximum length 4.

Example:

lines 112-113: `for I ← 1 step 1 until 6 do PRT.TAB(I,TAB,'␣TAB');`

S. Procedures

Procedures may not contain any of the SOL statements wait, wait until, enter or seize. They are not recursive.

T. Transaction Input-output

For the transaction read statement, the set of values of local variables for a transaction is inputted by the ALGOL-20 NAME and READ statements. These statements are coded in a

parameterless procedure declared inside the process in which the transaction occurs. Variables global to the process may also be changed.

procedure READIN(PROC,LAB); procedure PROC; label LAB:

The label must be the name of the transaction in which the transaction read statement occurs or the start of a statement occurring within the transaction. If the latter, then the label must be declared in the label statement declaring all transactions.

The transaction write statement can be achieved by NAME and PRINT statements in ALGOL-20.

IV. THE MODEL AS A WHOLE

The SOL system is written in ALGOL and must be loaded from the library by a system card. Preceding this card, the switch statement PROCESSES must be coded. Also the integer array PROC.PRIO and any facilities, stores, and global SOL variables that are simple variables must be declared. Facilities and stores must be declared as real variables.

After the system card, all stores, facilities (that are arrays), tables and global variables are declared. All labels used as parameters to the procedures NEW.TRANSACTION and READIN must be declared via a ALGOL-20 label declaration.

Next, the procedures STORE and TABLE must be called to give the maximum size of the stores and the bounds of the histogram for the tables. Also the vector PROC.PRIO must be initialized.

The simulation is started by the procedure call:

START.SIMULATION;

Then, each process is coded as an ALGOL block.

Finally, the code for outputting the statistics for all stores, tables and facilities occurs. It must be labeled with the predefined label RESULTS.

All of the above rules are exhibited on the on-line communication system problem of Appendix II.

Chapter 3
Tracing

In order to facilitate debugging, a tracing feature has been added to SOL. There are nine trace switches which can be turned on and off at will. Due to the fact that the names dictionary is no longer available at run time, symbolic tracing is not possible. Only the address at which the traced statement occurs is given and this is actually the starting address of the statement immediately after the traced statement. At present no information is given about which copy of the transaction is being executed at the time of the trace. The trace switches are contained in a boolean vector TRACE. The switch is set if the corresponding value is true and unset if it is false. All switches are initially unset.

Trace switch	Traces
TRACE[1]	everything
TRACE[2]	new transaction statements
TRACE[3]	cancel statements
TRACE[4]	seize statements
TRACE[5]	release statements
TRACE[6]	enter statements
TRACE[7]	leave statements
TRACE[8]	wait statements
TRACE[9]	wait until statements

Note: The trace switches are global quantities. Once a switch is set, the corresponding statement is traced in all transactions and all copies of a transaction.

Chapter 4
Run Time Errors

Errors may occur at run time, which, if the simulation were allowed to continue, would produce meaningless results. An error message along with the address at which it occurred is printed on the output and the run is aborted.

The following errors are detected:

Error 2: WAIT TABLE EXCEEDED

The simulation has accumulated more than 64 wait statements that need to be processed.

Error 3: WAIT UNTIL TABLE EXCEEDED

More than 64 wait until statements have accumulated.

Error 4: SEIZE TABLE EXCEEDED

More than 64 seize statements have accumulated.

Error 5: ENTER TABLE EXCEEDED

More than 64 enter statements have accumulated.

Error 6: IMPROPER RELEASE STATEMENT

The transaction releasing the facility is not controlling it.

Error 7: IMPROPER RELEASE STATEMENT

A facility is being released which has not been previously seized.

Error 8: NEW TRANSACTION TABLE EXCEEDED

More than 40 new transactions have been started.

Error 9: IMPROPER ENTER STATEMENT

The number of units requested for the store is less than 1.

Error 10: IMPROPER LEAVE STATEMENT

The number of units being released is more than the store already has in use.

Error 11: IMPROPER PR CALL

The input to the procedure PR is greater than 1 or less than 0.

Error 12: MEMORY EXCEEDED

There is no more space left in core to hold the local variables for the transactions.

Error 13: PROGRAM TOO LARGE

The system and SOL program take up all of memory, leaving no space for holding copies of local variables.

There are also many internal checks in the system. If a system error occurs, an internal error message will appear on the output and the run aborted.

It is very possible for the run time to expire before the simulation has finished. In order to obtain all statistics accumulated up to that point, the following card should be inserted into the SOL program:

```
RUN.ERROR(RESULTS,'TIMR');
```

Chapter 5

Reserved Identifiers

Since SOL-20 is written in ALGOL-20, certain identifiers have been reserved by the system besides the identifiers used for the SOL procedures and those reserved by ALGOL-20 itself. The total list of reserved identifiers for SOL-20 is listed below.

BUSY	NORMAL	SLT
CANCEL	POISSON	SRT
CTD	POP	S.TAB
EC	PR	START.SIMULATION
EMPTY	PRIORITY	STOP
ENTER	PROCESSES	STORE
ERROR	PROC.PRIO	TABLE
E.TAB	PRT.FAC	TABULATE
EXPONENTIAL	PRT.ST	TC
FULL	PRT.TAB	TIME
GEOMETRIC	PUSH	TRACE
H2	RAND	TR.DS
LBIT	RANDOM	T.TAB
LEAVE	READIN	UBIT
MASTER.CONTROL	RELEASE	UNIFORM
MAXAD	RESULTS	WAIT
MOVE	RETURN	WAIT.UNTIL
MTC	RUN.ERROR	WC
MT.TAB	SC	W.TAB
NEW.TRANSACTION	SCAL	WUC
	SEIZE	WU.TAB

Appendix I

Summary of SOL-20 Differences

I. Declarations

SOL	SOL-20
store *⟨constant⟩⟨declared item⟩*	real *⟨identifier⟩* / real array *⟨identifier⟩ [⟨number⟩:⟨number⟩]* STORE(⟨declared item⟩, ⟨constant⟩);
table *(⟨number⟩step⟨number⟩until ⟨number⟩)⟨declared item⟩*	half array *⟨identifier⟩[1:2, 0:⟨number⟩,1:⟨number⟩]* TABLE(⟨table number⟩,⟨table identifier⟩,⟨number⟩, ⟨number⟩,⟨number⟩);
facility *⟨declared item⟩*	real *⟨identifier⟩* real array *⟨identifier⟩ [⟨number⟩:⟨number⟩]*

II. Expressions

SOL	SOL-20
time	TIME
$(e_1,e_2,\ldots,e_n)$	RAND(⟨number⟩,⟨array identifier⟩);
$e_1:e_2$	UNIFORM(⟨number⟩,⟨number⟩);
normal(⟨expression⟩,⟨expression⟩)	NORMAL(⟨expression⟩, ⟨expression⟩);
exponential(⟨expression⟩)	EXPONENTIAL(⟨expression⟩);
poisson(⟨expression⟩)	POISSON(⟨expression⟩);
geometric(⟨expression⟩)	GEOMETRIC(⟨expression⟩);
random	RANDOM(⟨expression⟩, ⟨expression⟩,0);

III. Relations

SOL	SOL-20
⟨facility name⟩busy ⟨facility name⟩not busy	BUSY(⟨facility name⟩);
⟨store name⟩full ⟨store name⟩not full	FULL(⟨store name⟩);
⟨store name⟩empty ⟨store name⟩not empty	EMPTY(⟨store name⟩);
pr(⟨expression⟩)	PR(⟨expression⟩);

Appendix I

IV. Statements

SOL	SOL-20
process⟨identifier⟩; begin ⟨process declaration list⟩; ⟨statement list⟩end	switch PROCESSES := *⟨process identifer⟩*; integer array PROC.PRIO [1:⟨number⟩]; ⟨process identifier⟩: begin ⟨process declaration list⟩; ⟨statement list⟩end;
new transaction to label	NEW.TRANSACTION(⟨label⟩);
cancel	CANCEL;
wait⟨expression⟩	WAIT(⟨expression⟩);
wait until⟨relation⟩	WAIT.UNTIL(⟨relation⟩ | ⟨parameter-less boolean procedure⟩);
enter⟨store name⟩	ENTER(⟨store name⟩,⟨expression⟩);
leave⟨store name⟩	LEAVE(⟨store name⟩,⟨expression⟩);
seize⟨facility name⟩	SEIZE(⟨facility name⟩,⟨expression⟩);
release⟨facility name⟩	RELEASE(⟨facility name⟩);
go to (*⟨label⟩*),⟨expression⟩	use a switch statement
tabulate⟨expression⟩ in ⟨table name⟩	TABULATE(⟨expression⟩,⟨number⟩, ⟨table identifier⟩);
output statement	NAME and PRINT statements
stop	STOP;
read⟨constant⟩ to ⟨label⟩	READIN(⟨procedure identifier⟩, ⟨label⟩);
write⟨constant⟩	NAME and PRINT statements
statistical output(automatic)	PRT.FAC(⟨number⟩,⟨facility identifier⟩,⟨string⟩); PRT.ST(⟨number⟩,⟨store identifier⟩, ⟨string⟩); PRT.TAB(⟨number⟩,⟨table identifier⟩, ⟨string⟩);

1 4 0 1 C A R D L I S T

```
SYST LIST    MIN0005   PGS0200   CDS0000   TAP    HANSEN: S O L

SY PAGE                    S O L  TEST PROGRAM

   BEGIN                                                                    1
      REAL LINE, COMPUTER;     | FACILITIES                                 2
      SWITCH PROCESSES := CONTROL,USERS,PBU,OTHER,PBUS; | DECLARE PROCS.     3
      INTEGER ARRAY PROC.PRIO[1:4];     | PRIORITY TABLE FOR PROCESSES      4
SY LIBRARY SOL                                                              5
AL    REAL ARRAY TU[1:6], SB[1:3];     | FACILITIES                        6
      REAL ARRAY QUEUE[1:6];     | STORES                                   7
      INTEGER ARRAY TUSTATE,SBNUMBER,TUMESSAGE[1:6];                        8
      HALF ARRAY TAB[1:2,0:27,1:6];     | TABLES                           9
      INTEGER I;     | GLOBAL INDEX COUNTER                                10
      LABEL START,SCAN,COMPUTATION,COMPUTE;     | DECLARE TRANSACTIONS    11
            RUN.ERROR(RESULTS,'TIMR');                                      12
      FOR I ← 1 STEP 1 UNTIL 6 DO BEGIN                                     13
         STORE(QUEUE[I],10);     | DECLARE STORES                           14
         TABLE(I,TAB,2000,500,15000);     | DECLARE TABLES                 15
      END;                                                                  16
      FOR I ← 1 STEP 1 UNTIL 4 DO                                           17
         PROC.PRIO[I] ← 0;     | SET PRIORITIES FOR PROCESSES               18
      START.SIMULATION;                                                     19

      CONTROL: BEGIN                                                        20
         INTEGER I;     | DUMMY DECLARATION TO MAKE PROCESS A BLOCK         21
         SBNUMBER[1] ← 1; SBNUMBER[2] ← 2;                                  22
         SBNUMBER[3] ← 1; SBNUMBER[4] ← 2;                                  23
         SBNUMBER[5] ← 1; SBNUMBER[6] ← 3;                                  24
         WAIT(15*60*1000); STOP;     | STOP SIMULATION                      25
         END;                                                               26

      USERS: BEGIN                                                          27
         INTEGER Q,START.TIME,MESSAGE.TYPE;                                 28
         HALF ARRAY TYPE[1:10];                                             29
         TYPE[1] ← TYPE[2] ← 1;                                             30
         FOR I ← 3 STEP 1 UNTIL 7 DO TYPE[I] ← 2;                           31
         TYPE[8] ← TYPE[9] ← TYPE[10] ← 3;                                  32
         NEW.TRANSACTION(START); NEW.TRANSACTION(START);                    33
         ORIGIN: NEW.TRANSACTION(START); WAIT(UNIFORM(0,5000));             34
            GO TO ORIGIN;                                                   35
         START: Q ← UNIFORM(1,6); ENTER(QUEUE[Q],1);                        36
            MESSAGE.TYPE ← RAND(10,TYPE);                                   37
            SEIZE(TU[Q],0);                                                 38
            TUMESSAGE [Q] ← MESSAGE.TYPE;                                   39
            WAIT(UNIFORM(6000,8000));                                       40
            START.TIME ← TIME;                                              41
            NAME(Q,MESSAGE.TYPE,TIME);                                      42
            PRINT((10B,'TU',2D,1B,'SEND MESSAGE',2D,1B,'AT TIME',1B,7D,     43
                  1B,E));                                                   44
            TUSTATE[Q] ← 1;                                                 45
            WAIT.UNTIL(TUSTATE[Q] = 0);                                     46
            RELEASE(TU[Q]); LEAVE(QUEUE[Q],1);                              47
            TABULATE((TIME-START.TIME),Q,TAB);                              48
            NAME(Q,TIME);                                                   49
```

```
            PRINT(<208,'TU',2D,18,'RECEIVES REPLY AT TIME',18,7D,18,E>);      50
            CANCEL;                                                           51
         END;                                                                 52

      PBU: BEGIN                                                            53
         INTEGER S,T,WORDS;                                                 54
         NEW.TRANSACTION(SCAN); T ← 3;                                      55
         SCAN: T ← T + 1; IF T > 6 THEN T ← 1; WAIT(1);                     56
            S ← SBNUMBER[T];                                                57
            SEIZE(LINE,0);                                                  58
            WAIT(5); IF BUSY(SB[S]) THEN BEGIN                              59
               WAIT(80); RELEASE(LINE); GO TO SCAN END;                     60
            SEIZE(SB[S],0); WAIT(15); IF TUSTATE[T] ≠ 1 THEN BEGIN          61
               WAIT(65); RELEASE(LINE); RELEASE(SB[S]);                     62
               GO TO SCAN END;                                              63
            WAIT(225);                                                      64
            SEND: WAIT(170); IF PR(0.02) THEN BEGIN                         65
               WAIT(20); GO TO SEND END;                                    66
            NEW.TRANSACTION(COMPUTATION); WAIT(20); RELEASE(SB[S]);         67
            RELEASE(LINE); TUSTATE[T] ← 2;                                  68
            CANCEL;                                                         69
         COMPUTATION: SEIZE(COMPUTER,0); WORDS ← TUMESSAGE[T] + 2;          70
            WAIT(IF WORDS = 3 THEN 250 ELSE IF WORDS = 4 THEN 300 ELSE      71
                 400);                                                      72
            RELEASE(COMPUTER);                                              73
            OUTPT: WAIT(1); SEIZE(LINE,0); WAIT(5);                         74
               IF BUSY(SB[S]) THEN BEGIN                                    75
                  WAIT(80); RELEASE(LINE); GO TO OUTPT END;                 76
               SEIZE(SB[S],0); WAIT(75);                                    77
            RECEIVE: WAIT(80); IF PR(0.01) THEN BEGIN                       78
                  WAIT(20); GO TO RECEIVE END;                              79
            RELEASE(LINE);                                                  80
            WORDS ← WORDS - 1;                                              81
            IF WORDS = 0 THEN NEW.TRANSACTION(SCAN);                        82
            WAIT(325); RELEASE(SB[S]); WAIT(170);                           83
            IF WORDS > 0 THEN GO TO OUTPT;                                  84
            TUSTATE[T] ← 0;                                                 85
            CANCEL;                                                         86
         END;                                                               87

      OTHER.PBUS: BEGIN                                                     88
         INTEGER I; HALF ARRAY WAITS[1:10];                                 89
         WAITS[1] ← WAITS[2] ← 250;                                         90
         FOR I ← 3 STEP 1 UNTIL 7 DO WAITS[I] ← 300;                        91
         WAITS[8] ← WAITS[9] ← WAITS[10] ← 400;                             92
         I ← 6;                                                             93
         CREATE: NEW.TRANSACTION(COMPUTE);                                  94
            I ← I - 1; IF I > 0 THEN GO TO CREATE; CANCEL;                  95
         COMPUTE: WAIT(UNIFORM(3200,5000)); SEIZE(COMPUTER,0);              96
            WAIT(RAND(10,WAITS));                                           97
            RELEASE(COMPUTER); GO TO COMPUTE;                               98
         END;                                                               99

      RESULTS:                                                              100
CO PRINT FRACTION OF TIME USED BY FACILITIES                                101
      FOR I ← 1 STEP 1 UNTIL 6 DO                                           102
```

```
            PRT.FAC(I,TU[I],'  TU');                                    103
         FOR I ← 1 STEP 1 UNTIL 3 DO                                    104
            PRT.FAC(I,SB[I],'  SB');                                    105
         PRT.FAC(0,LINE,'LINE');                                        106
         PRT.FAC(0,COMPUTER,'COMP');                                    107
CO PRINT INFORMATION ON STORES                                          108
         FOR I ← 1 STEP 1 UNTIL 6 DO                                    109
            PRT.ST(I,QUEUE[I],' QUE');                                  110
CO PRINT STATISTICS ON TABLES                                           111
         FOR I ← 1 STEP 1 UNTIL 6 DO                                    112
            PRT.TAB(I,TAB,' TAB');                                      113
   END;                                                                 114
```

```
03 FEB 65
TU 2 SEND MESSAGE 1 AT TIME     7752
TU 4 SEND MESSAGE 2 AT TIME     7784
TU 3 SEND MESSAGE 2 AT TIME    10302
          TU 2 RECEIVES REPLY AT TIME    10690
TU 1 SEND MESSAGE 2 AT TIME    14052
          TU 3 RECEIVES REPLY AT TIME    14710
          TU 4 RECEIVES REPLY AT TIME    15842
          TU 1 RECEIVES REPLY AT TIME    18547
TU 5 SEND MESSAGE 2 AT TIME    19792
TU 4 SEND MESSAGE 2 AT TIME    21169
TU 6 SEND MESSAGE 3 AT TIME    23988
          TU 5 RECEIVES REPLY AT TIME    23999
          TU 4 RECEIVES REPLY AT TIME    25270
TU 1 SEND MESSAGE 2 AT TIME    25935
TU 2 SEND MESSAGE 1 AT TIME    26675
          TU 6 RECEIVES REPLY AT TIME    28700
          TU 1 RECEIVES REPLY AT TIME    30038
          TU 2 RECEIVES REPLY AT TIME    30759
TU 3 SEND MESSAGE 3 AT TIME    31488
TU 4 SEND MESSAGE 3 AT TIME    31842
TU 5 SEND MESSAGE 3 AT TIME    32952
          TU 3 RECEIVES REPLY AT TIME    35719
          TU 4 RECEIVES REPLY AT TIME    36137
TU 1 SEND MESSAGE 2 AT TIME    36767
          TU 5 RECEIVES REPLY AT TIME    39946
TU 2 SEND MESSAGE 2 AT TIME    41957
TU 3 SEND MESSAGE 1 AT TIME    42977
TU 4 SEND MESSAGE 3 AT TIME    43320
          TU 1 RECEIVES REPLY AT TIME    44198
          TU 2 RECEIVES REPLY AT TIME    46048
TU 5 SEND MESSAGE 3 AT TIME    46994
          TU 3 RECEIVES REPLY AT TIME    48702
          TU 4 RECEIVES REPLY AT TIME    49620
TU 6 SEND MESSAGE 3 AT TIME    49811
TU 1 SEND MESSAGE 2 AT TIME    52131
          TU 5 RECEIVES REPLY AT TIME    53783
          TU 6 RECEIVES REPLY AT TIME    54201
TU 2 SEND MESSAGE 2 AT TIME    54502
TU 3 SEND MESSAGE 1 AT TIME    54982
TU 4 SEND MESSAGE 1 AT TIME    55320
          TU 1 RECEIVES REPLY AT TIME    57192
          TU 2 RECEIVES REPLY AT TIME    58113
TU 5 SEND MESSAGE 3 AT TIME    60533
          TU 3 RECEIVES REPLY AT TIME    60541
          TU 4 RECEIVES REPLY AT TIME    60787
          TU 5 RECEIVES REPLY AT TIME    65575
TU 6 SEND MESSAGE 1 AT TIME    66013
TU 2 SEND MESSAGE 2 AT TIME    66100
TU 3 SEND MESSAGE 3 AT TIME    67310
TU 4 SEND MESSAGE 2 AT TIME    68017
          TU 6 RECEIVES REPLY AT TIME    69241
          TU 2 RECEIVES REPLY AT TIME    70229
TU 5 SEND MESSAGE 3 AT TIME    72289
          TU 3 RECEIVES REPLY AT TIME    73515
```

```
TU 2 SEND MESSAGE 3 AT TIME  844629
TU 3 SEND MESSAGE 3 AT TIME  847312
                TU 1 RECEIVES REPLY AT TIME  847442
                TU 5 RECEIVES REPLY AT TIME  848778
                TU 2 RECEIVES REPLY AT TIME  851642
                TU 6 RECEIVES REPLY AT TIME  851938
TU 1 SEND MESSAGE 2 AT TIME  854122
                TU 3 RECEIVES REPLY AT TIME  855056
TU 5 SEND MESSAGE 2 AT TIME  855247
TU 2 SEND MESSAGE 2 AT TIME  857768
TU 6 SEND MESSAGE 2 AT TIME  858154
                TU 1 RECEIVES REPLY AT TIME  858529
                TU 2 RECEIVES REPLY AT TIME  861612
TU 3 SEND MESSAGE 1 AT TIME  861631
                TU 6 RECEIVES REPLY AT TIME  862380
                TU 3 RECEIVES REPLY AT TIME  865660
                TU 5 RECEIVES REPLY AT TIME  866160
TU 1 SEND MESSAGE 2 AT TIME  866164
TU 6 SEND MESSAGE 2 AT TIME  869291
                TU 1 RECEIVES REPLY AT TIME  870313
TU 3 SEND MESSAGE 1 AT TIME  872544
TU 5 SEND MESSAGE 1 AT TIME  872835
TU 2 SEND MESSAGE 1 AT TIME  873270
                TU 6 RECEIVES REPLY AT TIME  873446
TU 4 SEND MESSAGE 3 AT TIME  876671
                TU 3 RECEIVES REPLY AT TIME  877296
                TU 5 RECEIVES REPLY AT TIME  877476
TU 1 SEND MESSAGE 3 AT TIME  879085
TU 6 SEND MESSAGE 2 AT TIME  880052
                TU 2 RECEIVES REPLY AT TIME  881920
                TU 4 RECEIVES REPLY AT TIME  882420
TU 3 SEND MESSAGE 1 AT TIME  884231
                TU 6 RECEIVES REPLY AT TIME  885674
                TU 1 RECEIVES REPLY AT TIME  887470
TU 2 SEND MESSAGE 2 AT TIME  888155
                TU 3 RECEIVES REPLY AT TIME  888645
TU 4 SEND MESSAGE 2 AT TIME  890303
TU 6 SEND MESSAGE 2 AT TIME  892395
                TU 2 RECEIVES REPLY AT TIME  892920
                TU 4 RECEIVES REPLY AT TIME  894775
TU 3 SEND MESSAGE 1 AT TIME  895978
                TU 6 RECEIVES REPLY AT TIME  896629
                TU 3 RECEIVES REPLY AT TIME  898917

CLOCK TIME AT END OF SIMULATION WAS .90000000 E+06
```

03 FEB 65

NAME OF FACILITY	FRACTION OF TIME IN USE
TU[01]	0.6645
TU[02]	0.7802
TU[03]	0.8633
TU[04]	0.8137
TU[05]	0.7230
TU[06]	0.8529
SB[01]	0.5843
SB[02]	0.4166
SB[03]	0.2327
LINE	0.8761
COMP	0.5168

03 FEB 65

NAME OF STORE	CAPACITY	MAXIMUM USED	AVERAGE OCCUPANCY	AVERAGE UTILIZATION
QUE[001]	10	4	1.9796	0.1980
QUE[002]	10	8	2.4839	0.2484
QUE[003]	10	10	3.6486	0.3649
QUE[004]	10	5	2.5833	0.2583
QUE[005]	10	4	1.8545	0.1855
QUE[006]	10	6	3.0441	0.3044

03 FEB 65

TABLE NAME IS TAB10011

NUMBER OF TABLE ENTRIES 49 SUM OF ALL ENTRY VALUES .2520930000

MEAN OF TABLE 5144.7551 STANDARD DEVIATION 1501.1870

UPPER LIMIT	NUMBER	PER CENT	CUMULATIVE	MULTIPLE OF MEAN
2000.0	0	0.00	0.00	0.3887
2500.0	0	0.00	0.00	0.4859
3000.0	0	0.00	0.00	0.5831
3500.0	4	8.16	8.16	0.6803
4000.0	8	16.33	24.49	0.7775
4500.0	10	20.41	44.90	0.8747
5000.0	6	12.24	57.14	0.9719
5500.0	7	14.29	71.43	1.0690
6000.0	3	6.12	77.55	1.1662
6500.0	1	2.04	79.59	1.2634
7000.0	2	4.08	83.67	1.3606
7500.0	4	8.16	91.84	1.4578
8000.0	1	2.04	93.88	1.5550
8500.0	1	2.04	95.92	1.6522
9000.0	1	2.04	97.96	1.7494
9500.0	1	2.04	100.00	1.8465
10000.0	0	0.00	100.00	1.9437
10500.0	0	0.00	100.00	2.0409
11000.0	0	0.00	100.00	2.1381
11500.0	0	0.00	100.00	2.2353
12000.0	0	0.00	100.00	2.3325
12500.0	0	0.00	100.00	2.4297
13000.0	0	0.00	100.00	2.5268
13500.0	0	0.00	100.00	2.6240
14000.0	0	0.00	100.00	2.7212
14500.0	0	0.00	100.00	2.8184
15000.0	0	0.00	100.00	2.9156

03 FEB 65

TABLE NAME IS TAB10021

NUMBER OF TABLE ENTRIES 60 SUM OF ALL ENTRY VALUES .2784510000 E+06

MEAN OF TABLE 4640.8500 STANDARD DEVIATION 1365.2080

UPPER LIMIT	NUMBER	PER CENT	CUMULATIVE	MULTIPLE OF MEAN
2000.0	0	0.00	0.00	0.4310
2500.0	0	0.00	0.00	0.5387
3000.0	3	5.00	5.00	0.6464
3500.0	5	8.33	13.33	0.7542
4000.0	16	26.67	40.00	0.8619
4500.0	12	20.00	60.00	0.9696
5000.0	8	13.33	73.33	1.0774
5500.0	5	8.33	81.67	1.1851
6000.0	1	1.67	83.33	1.2929
6500.0	3	5.00	88.33	1.4006
7000.0	2	3.33	91.67	1.5083
7500.0	2	3.33	95.00	1.6161
8000.0	1	1.67	96.67	1.7238
8500.0	0	0.00	96.67	1.8316
9000.0	2	3.33	100.00	1.9393
9500.0	0	0.00	100.00	2.0470
10000.0	0	0.00	100.00	2.1548
10500.0	0	0.00	100.00	2.2625
11000.0	0	0.00	100.00	2.3703
11500.0	0	0.00	100.00	2.4780
12000.0	0	0.00	100.00	2.5857
12500.0	0	0.00	100.00	2.6935
13000.0	0	0.00	100.00	2.8012
13500.0	0	0.00	100.00	2.9089
14000.0	0	0.00	100.00	3.0167
14500.0	0	0.00	100.00	3.1244
15000.0	0	0.00	100.00	3.2322

03 FEB 65

TABLE NAME IS TAB(0031)

NUMBER OF TABLE ENTRIES 65 SUM OF ALL ENTRY VALUES .3233110000 .+06

MEAN OF TABLE 4974.0154 STANDARD DEVIATION 1298.1728

UPPER LIMIT	NUMBER	PER CENT	CUMULATIVE	MULTIPLE OF MEAN
2000.0	0	0.00	0.00	0.4021
2500.0	0	0.00	0.00	0.5026
3000.0	1	1.54	1.54	0.6031
3500.0	1	1.54	3.08	0.7037
4000.0	13	20.00	23.08	0.8042
4500.0	15	23.08	46.15	0.9047
5000.0	9	13.85	60.00	1.0052
5500.0	9	13.85	73.85	1.1057
6000.0	5	7.69	81.54	1.2063
6500.0	3	4.62	86.15	1.3068
7000.0	3	4.62	90.77	1.4073
7500.0	3	4.62	95.38	1.5078
8000.0	1	1.54	96.92	1.6084
8500.0	0	0.00	96.92	1.7089
9000.0	1	1.54	98.46	1.8094
9500.0	1	1.54	100.00	1.9099
10000.0	0	0.00	100.00	2.0104
10500.0	0	0.00	100.00	2.1110
11000.0	0	0.00	100.00	2.2115
11500.0	0	0.00	100.00	2.3120
12000.0	0	0.00	100.00	2.4125
12500.0	0	0.00	100.00	2.5131
13000.0	0	0.00	100.00	2.6136
13500.0	0	0.00	100.00	2.7141
14000.0	0	0.00	100.00	2.8146
14500.0	0	0.00	100.00	2.9151
15000.0	0	0.00	100.00	3.0157

03 FEB 65

TABLE NAME IS TAB(004)

NUMBER OF TABLE ENTRIES 59 SUM OF ALL ENTRY VALUES .3095450000 E+06

MEAN OF TABLE 5246.5254 STANDARD DEVIATION 1595.8324

UPPER LIMIT	NUMBER	PER CENT	CUMULATIVE	MULTIPLE OF MEAN
2000.0	0	0.00	0.00	0.3812
2500.0	0	0.00	0.00	0.4765
3000.0	0	0.00	0.00	0.5718
3500.0	2	3.39	3.39	0.6671
4000.0	10	16.95	20.34	0.7624
4500.0	12	20.34	40.68	0.8577
5000.0	10	16.95	57.63	0.9530
5500.0	6	10.17	67.80	1.0483
6000.0	4	6.78	74.58	1.1436
6500.0	4	6.78	81.36	1.2389
7000.0	2	3.39	84.75	1.3342
7500.0	2	3.39	88.14	1.4295
8000.0	4	6.78	94.92	1.5248
8500.0	0	0.00	94.92	1.6201
9000.0	1	1.69	96.61	1.7154
9500.0	0	0.00	96.61	1.8107
10000.0	1	1.69	98.31	1.9060
10500.0	0	0.00	98.31	2.0013
11000.0	1	1.69	100.00	2.0966
11500.0	0	0.00	100.00	2.1919
12000.0	0	0.00	100.00	2.2872
12500.0	0	0.00	100.00	2.3825
13000.0	0	0.00	100.00	2.4778
13500.0	0	0.00	100.00	2.5731
14000.0	0	0.00	100.00	2.6684
14500.0	0	0.00	100.00	2.7637
15000.0	0	0.00	100.00	2.8590

03 FEB 65

TABLE NAME IS TAB(0051)

NUMBER OF TABLE ENTRIES 54 SUM OF ALL ENTRY VALUES .2770320000 ,+06

MEAN OF TABLE 5130.2222 STANDARD DEVIATION 1558.8936

UPPER LIMIT	NUMBER	PER CENT	CUMULATIVE	MULTIPLE OF MEAN
2000.0	0	0.00	0.00	0.3898
2500.0	0	0.00	0.00	0.4873
3000.0	2	3.70	3.70	0.5848
3500.0	5	9.26	12.96	0.6822
4000.0	5	9.26	22.22	0.7797
4500.0	11	20.37	42.59	0.8772
5000.0	6	11.11	53.70	0.9746
5500.0	7	12.96	66.67	1.0721
6000.0	5	9.26	75.93	1.1695
6500.0	4	7.41	83.33	1.2670
7000.0	4	7.41	90.74	1.3645
7500.0	1	1.85	92.59	1.4619
8000.0	1	1.85	94.44	1.5594
8500.0	1	1.85	96.30	1.6568
9000.0	1	1.85	98.15	1.7543
9500.0	0	0.00	98.15	1.8518
10000.0	0	0.00	98.15	1.9492
10500.0	0	0.00	98.15	2.0467
11000.0	1	1.85	100.00	2.1442
11500.0	0	0.00	100.00	2.2416
12000.0	0	0.00	100.00	2.3391
12500.0	0	0.00	100.00	2.4365
13000.0	0	0.00	100.00	2.5340
13500.0	0	0.00	100.00	2.6315
14000.0	0	0.00	100.00	2.7289
14500.0	0	0.00	100.00	2.8264
15000.0	0	0.00	100.00	2.9238

03 FEB 65

TABLE NAME IS TAB:0061

NUMBER OF TABLE ENTRIES 64 SUM OF ALL ENTRY VALUES .3116160000 *+06

MEAN OF TABLE 4869.0000 STANDARD DEVIATION 1310.2439

UPPER LIMIT	NUMBER	PER CENT	CUMULATIVE	MULTIPLE OF MEAN
2000.0	0	0.00	0.00	0.4108
2500.0	0	0.00	0.00	0.5135
3000.0	0	0.00	0.00	0.6161
3500.0	3	4.69	4.69	0.7188
4000.0	10	15.63	20.31	0.8215
4500.0	25	39.06	59.38	0.9242
5000.0	2	3.12	62.50	1.0269
5500.0	11	17.19	79.69	1.1296
6000.0	5	7.81	87.50	1.2323
6500.0	1	1.56	89.06	1.3350
7000.0	1	1.56	90.62	1.4377
7500.0	1	1.56	92.19	1.5404
8000.0	2	3.12	95.31	1.6430
8500.0	2	3.12	98.44	1.7457
9000.0	0	0.00	98.44	1.8484
9500.0	0	0.00	98.44	1.9511
10000.0	1	1.56	100.00	2.0538
10500.0	0	0.00	100.00	2.1565
11000.0	0	0.00	100.00	2.2592
11500.0	0	0.00	100.00	2.3619
12000.0	0	0.00	100.00	2.4646
12500.0	0	0.00	100.00	2.5673
13000.0	0	0.00	100.00	2.6700
13500.0	0	0.00	100.00	2.7726
14000.0	0	0.00	100.00	2.8753
14500.0	0	0.00	100.00	2.9780
15000.0	0	0.00	100.00	3.0807

TIME USED: 00:44:06 PAGES USED: 26 02:24:03

Appendix III

SOL—A Symbolic Language for General-Purpose Systems Simulation

D. E. KNUTH AND J. L. McNELEY

Summary—**This paper illustrates the use of SOL, a general-purpose algorithmic language useful for describing and simulating complex systems. Such a system is described as a number of individual processes which simultaneously enact a program very much like a computer program. (Some features of the SOL language are directly applicable to programming languages for parallel computers, as well as for simulation.) Once a system has been described in the language, the program can be translated by the SOL compiler into an interpretive code, and the execution of this code produces statistical information about the model. A detailed example of a SOL model for a multiple on-line console system is exhibited, indicating the notational simplicity and intuitive nature of the language.**

SIMULATION by computer is one of the most important tools available to scientists and engineers who are studying complex systems. The first computer programs of this type were especially designed to simulate some particular model; but afterwards the authors of several of these programs abstracted the essential features of their program organization and prepared *general-purpose* simulation programs. The most extensively used general-purpose programs of this type have apparently been the SIMSCRIPT compiler of Markowitz, Hauser, and Karr [1], and the GPSS (General-Purpose Systems Simulator) routines of Gordon [2]–[4].

Although SIMSCRIPT and GPSS are both general-purpose simulation programs, they are built around quite different concepts because of their independent evolution, and so they bear little resemblance to each other. SOL (Simulation-Oriented Language) is another general-purpose simulation routine, in which we have attempted to incorporate the best features of the other languages. After a careful study of SIMSCRIPT and GPSS, and after having implemented a version of GPSS for another computer, we found that it would be possible to generalize the characteristics of the former programs, while at the same time the language became simpler and more convenient for the preparation of models. This simplification was achieved by extracting the essential characteristics of GPSS and recasting them into a symbolic language such as SIMSCRIPT. There are, of course, a great many ways in which this can be done, and we are not sure that the compromises we have chosen have been optimal; but a year of experience with the SOL language, after applying it to a number of problems of different kinds, indicates that SOL is a quite powerful and flexible way to describe systems for simulation. We also found that the increased generality available in SOL was actually simpler to implement into a computer program than the previous routines were.

Manuscript received January 3, 1964.
D. E. Knuth is with the California Institute of Technology, Pasadena, Calif.
J. L. McNeley is with the Burroughs Corporation, Pasadena, Calif.

A complex system can be represented as a number of individual processes, each of which follows a *program* very much like a computer program. For example, if we were simulating traffic in a network of streets, we might have one program describing a typical automobile (or perhaps two programs, one which describes all of the women drivers and one which describes all of the men), another program which represents the action of traffic signals, and possibly some other programs representing pedestrians, etc. Each program depends not only on quantities which are specified in advance, but also on *random* quantities which describe a probabilistic behavior; thus, we can specify the probability that a driver will turn left, the probability that he will switch lanes, the distribution of speeds, etc. Although each program represents only a single entity (such as a single automobile), there can be many entities each carrying out the same program, each at its own place in the program.

Because of these considerations, SOL is a language which is in many respects very much like a problem-oriented language such as ALGOL or FORTRAN. There are three major points of difference between SOL and conventional compiler languages. SOL provides

1) mechanisms for parallel computation,
2) a convenient notation for random elements within arithmetic expressions,
3) automatic means of gathering statistics about the elements involved.

On the other hand, many of the features of problem-oriented languages do not appear in SOL, not because they are incompatible with it, but rather because they introduce more complication into this scheme than seems to be of practical value for simulation processes.

A program written in the SOL language is punched onto cards and it is then compiled by the SOL *compiler* into an interpretive pseudocode. The SOL *interpreter* is another machine program, which executes this pseudocode and produces the results. (The SOL system has been implemented for the B5000 computer, but at the present time it is being used only for research within the Burroughs Corporation, and it is not currently available for distribution.)

A self-contained, complete description of SOL ap-

pears in another paper [5]. The definition there is rather terse since it is intended primarily as a reference description; we will introduce the language here by means of an example, discussing the significance of each statement in an intuitive fashion.

Example: Communication with Remote Terminals

The following example has been chosen not only to illustrate most of the features of SOL, but also because it is a practical application in which SOL has been used to evaluate the design of an actual system of some complexity.

Consider the configuration shown in Fig. 1. This represents one of four similar groups of devices which all share the processor shown at the right. The "TU's" are terminal units which may be thought of as inquiry stations or typewriters. There are three groups of typewriters, with three in the first group (TU[1], TU[3], TU[5]), two in the second group (TU[2], TU[4]) and only one in the third (TU[6]). These groups are located many miles from each other and from the central processor. People come in at the rate of about five or six per minute to use each typewriter, and they wait in the appropriate queue until the typewriter is free.

These people will send one of three kinds of messages.

Message	Frequency	Compute time	Number of Response Words
A	20 per cent	250 msec	3
B	50 per cent	300 msec	4
C	30 per cent	400 msec	5

Each message type has a different frequency and requires a different amount of central processor time.

Communication between the typewriters and the processor is handled by *site buffers* SB[1], SB[2], SB[3], one at each remote site, and by two *processor buffers* PBU's, which receive the information and transmit it to the computer. These processor buffers sequentially scan TU[1], TU[2], $\cdots$, TU[6], TU[1], $\cdots$ until locating a typewriter ready to transmit information; this scanning is done by sending control pulses to all lines, then receiving a "positive" response from the SB if the appropriate TU is ready. Then a message is transferred from SB to the PBU and from there to the processor; after computing the answer, the processor refills the PBU, and the appropriate number of words is sent back to the SB and is typed on the TU (one word at a time). Further details will be given as we discuss the program.

We will compose three programs.

1) A program which describes the action of each person who uses the remote typewriters.
2) A program which describes the action of each of the two PBU's.
3) A program which simulates the action of the other six PBU's, which share the central processor with the configuration shown in Fig. 1.

Fig. 2 shows these three programs together with the control information, as a complete SOL model.

The independent quantities which enact the programs as the simulation proceeds are called *transactions*. (Much of the terminology used in SOL is taken from Gordon's simulator [2]–[4].) As simulation begins, there are only three transactions: one for each of the programs 1), 2), 3). Therefore, these programs describe not only the action of the quantities mentioned above, they also describe the creation and dissolution of new transactions.

Each transaction contains *local variables* which have values that can be referred to only by that transaction. There are also *global variables*, and some other types of global quantities, which can be referred to by all transactions. Thus, transactions can interact with each other by setting and testing global quantities. Only one "copy" of each global variable is present in the system, but there are in general many copies of each local variable (one for each transaction).

Program 1), which represents the people using the typewriters, might begin as follows:

```
process USERS;
begin integer Q, START TIME, MESSAGE TYPE;
new transaction to START; new transaction to START;
ORIGIN: new transaction to START; wait 0:5000; go to
    ORIGIN;
START:
```

The first line merely identifies a *process* (*i.e.*, a program) with the name "USERS." The language resembles ALGOL, and we distinguish control words by putting them in bold-face type. The second line states that there are three local variables in these transactions, having the names Q, START TIME and MESSAGE TYPE. The statement "**new transaction to** START" describes the creation of a new transaction whose local variables have the same values as the local variables of the parent transaction (in this case zero, since all local variables are automatically set to zero at the beginning of a process), and this new transaction begins executing the program at the statement labeled START. The statement "**wait** 0:5000" means an amount of simulated time, chosen randomly from 0 to 5000, is to elapse before the next statement is executed. In general, the statement "**wait** E," where E is some expression, means that E units of time are to pass before excuting the next statement. The expression $E_1:E_2$ always denotes a random integer chosen between E_1 and E_2, and therefore "**wait** 0:5000" has the meaning stated above. A unit of time in this case represents 1 msec in the simulated model.

The reader should now reread the above sequence of coding before proceeding further. The essential action it describes is that three transactions will begin executing the program beginning at the statement called START, and thereafter a new transaction (*i.e.*, a new user enter-

Appendix III

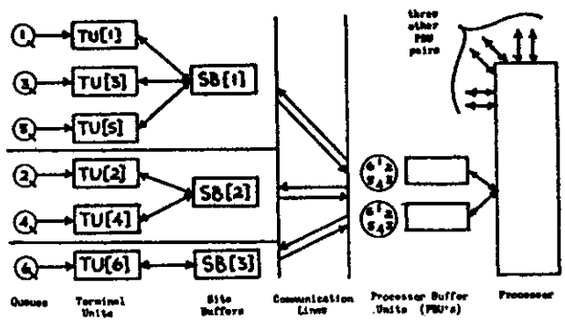

Fig. 1—Multiple console on-line communication system.

```
begin
facility TU[6], SB[3], LINE, COMPUTER;
store 10 QUEUE[6];
integer TUSTATE[6], SBNUMBER[6], TUMESSAGE[6];
table (2000 step 500 until 15000) TABLE[6];
process MASTER CONTROL;
begin SBNUMBER[1]←1; SBNUMBER[2]←2;
      SBNUMBER[3]←1; SBNUMBER[4]←2;
      SBNUMBER[5]←1; SBNUMBER[6]←3;
wait 60×60×1000; stop end;
process USERS;
begin integer Q, START TIME, MESSAGE TYPE;
new transaction to START; new transaction to START;
ORIGIN: new transaction to START; wait 0:5000; go to
  ORIGIN;
START: Q←1:6; enter QUEUE[Q];
MESSAGE TYPE←(1,1,2,2,2,2,2,3,3,3);
seize TU[Q];
TUMESSAGE[Q]←MESSAGE TYPE;
wait 6000:8000;
START TIME←time;
output #TU#, Q, #SENDS MESSAGE#, MESSAGE TYPE,
  #AT TIME#, time;
TUSTATE[Q]←1;
wait until TUSTATE[Q]=0;
release TU[Q]; leave QUEUE[Q];
tabulate (time−START TIME) in TABLE[Q];
output #TU#, Q, #RECEIVES REPLY AT TIME#, time;
cancel end;
process PBU; begin integer S, T, WORDS;
new transaction to SCAN; T←3;
SCAN: T←T+1; if T>6 then T←1; wait 1;
S←SBNUMBER[T];
seize LINE;
wait 5; if SB[S] busy then (wait 80; release LINE; go to
  SCAN);
seize SB[S]; wait 15; if TUSTATE[T]≠1 then
(wait 65; release LINE; release SB[S]; go to SCAN);
wait 225; SEND: wait 170; if pr(0.02) then (wait 20; go to
  SEND);
new transaction to COMPUTATION; wait 20; release SB[S];
release LINE; TUSTATE[T]←2; cancel;
COMPUTATION: seize COMPUTER; WORDS←TUMESSAGE[T]
  +2;
wait (if WORDS=3 then 250 else if WORDS=4 then 300
  else 400);
release COMPUTER;
OUTPUT: wait 1; seize LINE; wait 5;
if SB[S] busy then (wait 80; release LINE; go to OUTPUT);
seize SB[S]; wait 75;
RECEIVE: wait 80; if pr(0.01) then (wait 20; go to
  RECEIVE);
release LINE;
WORDS←WORDS−1;
if WORDS=0 then new transaction to SCAN;
wait 325; release SB[S]; wait 170;
if WORDS>0 then go to OUTPUT;
TUSTATE[T]←0; cancel end;
process OTHER PBUS;
begin integer I; I←6;
CREATE: new transaction to COMPUTE;
I←I−1; if I>0 then go to CREATE; cancel;
COMPUTE: wait 3200:5000; seize COMPUTER;
wait (250, 250, 300, 300, 300, 300, 300, 400, 400, 400);
release COMPUTER; go to COMPUTE end;
end.
```

Fig. 2—Complete SOL program for the on-line system.

Appendix III

ing the system) will be created at intervals of about 2.5 sec. We have started the system with three transactions so that it will not take it very long to arrive at a more or less stable condition.

The program now proceeds as follows:

START: Q←1:6; enter QUEUE[Q];

The statement "Q←1:6" means that local variable Q is set to a random number between 1 and 6; thus the user is assigned to one of the six typewriters. The "enter" statement refers to one of six global quantities, QUEUE[1], · · · , QUEUE[6]. At the conclusion of the simulation, data will be reported giving the average number of people in each queue at a given time, and also the maximum number.

MESSAGE TYPE←(1,1,2,2,2,2,2,3,3,3);

The expression $(E_1, E_2, \cdots, E_n)$ denotes a random choice selected from among the n expressions. Therefore, the given statement means that the local variable MESSAGE TYPE receives the value 1 with probability 20 per cent, 2 with probability 50 per cent and 3 with probability 30 per cent; this represents the choice of message A, B or C as stated earlier.

seize TU[Q];

This statement refers to one of the global quantities TU[1], · · · , TU[6], which are classified as *facilities*. A facility is *seized* by one transaction, and then it cannot be seized by another transaction until it has been *released* by the former transaction. Therefore, if transaction X comes to a seize statement, where the corresponding facility is *busy* (*i.e.*, has been seized by transaction Y), transaction X stops executing its program until transaction Y releases the facility. If several transactions are waiting for this event, they are processed in a first-come-first-served fashion.

Thus, the statement "seize TU[Q]" expresses the situation that the user takes control of typewriter number Q, after possibly waiting in line for it to become available.

TUMESSAGE[Q]←MESSAGE TYPE;

This statement says that the global variable TUMESSAGE[Q] is set to indicate the type of message. This global variable is used to communicate with the PBU process which is described below.

wait 6000:8000;

This statement simulates the time of 6 to 8 sec, taken by the man to type his request on the terminal unit.

START TIME←time;

We now set the local variable START TIME equal to "time," the current value of the simulated clock.

output #TU#, Q, #SENDS MESSAGE#, MESSAGE TYPE, #AT TIME#, time;

This statement causes the printing of a line during the simulation, having the form "TU 3 SENDS MESSAGE 2 AT TIME 12610." The "#" symbols indicate a string inserted into the output.

TUSTATE[Q]←1;

Another global variable TUSTATE[Q] is now set to 1 to indicate that the typed message is ready to send. TUSTATE[Q] has three possible settings.

TUSTATE = 0 means the TU is free.
TUSTATE = 1 means the message has been typed.
TUSTATE = 2 means the answer message may be typed.

The next statement

wait until TUSTATE[Q] = 0;

means the transaction is to stop at this point until TUSTATE[Q] has been set to zero (by some other transaction). This indicates that we are to wait until the answer message has been fully received. When that occurs, the transaction finishes its work as follows:

release TU[Q]; leave QUEUE[Q];
tabulate (time − START TIME) in TABLE[Q];

The latter statement is used for statistical data; TABLE[Q] is a global quantity which receives "readings" by means of "tabulate" statements. At the end of simulation, this table is printed out giving the mean, the standard deviation and a histogram of the data it has received.

output #TU#, Q, #RECEIVES REPLY AT TIME#, time;
cancel end;

The last statement, "cancel," causes the disappearance of the transaction, and the word "end" indicates the end of the program for this process.

Program 2), which runs simultaneously with 1) and 3), describes the action of the PBU's.

process PBU; begin integer S, T, WORDS;
new transaction to SCAN; T←3;
SCAN:

We have three local variables, S, T and WORDS. At the beginning, two transactions (representing the two PBU's) start at SCAN, one with its variable T = 0, the other with T = 3.

SCAN: T←T+1; if T>6 then T←1; wait 1;

These statements represent the cyclic scanning process which we assume takes 1 msec. The variable T represents the number of the TU which the PBU will be referencing.

S←SBNUMBER[T];

"SBNUMBER" is a table of constants, which is used to tell which SB corresponds to the TU scanned.

seize LINE;

Appendix III

We now seize the facility LINE, which represents the long-distance communication lines. (If the other PBU has seized LINE already, we must wait until it has been released.)

```
wait 5; if SB[s] busy then
                    (wait 80; release LINE; go to SCAN);
```

We wait 5 msec for a control signal to propagate to the SB unit. Here SB[s] is a facility; if it is busy (*i.e.*, has been seized by the other PBU) we wait 80 msec more, receiving no signal back, so we release the line and return to scan the next TU.

```
seize SB[s]; wait 15; if TUSTATE[T]≠1 then
(wait 65; release LINE; release SB[s]; go to SCAN);
```

If SB[s] received the control signal, it is brought under the control of this PBU. Fifteen milliseconds later, the number T has been transmitted across the line, and it takes 65 msec for the SB to determine if TU[T] is ready to transmit or not. If not, we release the SB and the line, and scan again.

```
wait 225; SEND: wait 170; if pr(0.02) then
                                (wait 20; go to SEND);
```

It takes 225 msec for the SB to get ready to transmit the message and to send a warning signal across the line to the PBU. Then 170 msec are required to send the input message. The construction "if pr(0.02)" means "2 per cent of the time," and so this statement indicates that, with probability 0.02, a parity error in the transmission is detected; in such a case, we send back a signal calling for retransmission of the message.

```
new transaction to COMPUTATION; wait 20; release SB[s];
release LINE; TUSTATE[T]←2; cancel;
```

At this point two parallel processes take place. As the PBU tries to send the message to the computer, it also sends a "message received" signal across the lines to the SB, and, 20 msec later, the SB and the lines are released. The TUSTATE is adjusted, and then this portion of the transaction is cancelled.

```
COMPUTATION: seize COMPUTER;
  WORDS←TUMESSAGE[T]+2;
wait (if WORDS=3 then 250 else
  if WORDS
  =4 then 300 else 400);
release COMPUTER;
```

Here we send the message to the computer facility, possibly waiting for it to become available. The local variable WORDS is set to the number of words output for the current message, and we also wait the appropriate amount of computer time. At this point, the output message has been created by the computer, and it has been sent back to the PBU. The final job is to output this message, one word at a time:

```
OUTPUT: wait 1; seize LINE; wait 5;
if SB[s] busy then (wait 80; release LINE; go to OUTPUT);
```

A control word is sent out to interrogate the SB, as in the case of input above.

```
seize SB[s]; wait 75;
RECEIVE: wait 80; if pr(0.01) then
  (wait 20; go to RECEIVE);
release LINE;
```

We have output one word to the SB; there was probability 1 per cent that a transmission error was detected.

```
WORDS←WORDS−1;
if WORDS=0 then new transaction to SCAN;
wait 325; release SB[s]; wait 170;
```

After the last word has been transmitted, a parallel activity starts with another scan. It takes 325 msec for the SB to send the word to the typewriter, and another 170 msec are required for the typewriter to finish its typing.

```
if WORDS>0 then go to OUTPUT;
TUSTATE[T]←0; cancel end;
```

When the output has all been typed, TUSTATE is reset to zero (thus activating the USER transaction) and this parallel branch of the program disappears.

Program 3) is used to describe the traffic which takes place at the computer, by creating six simulated PBU's as follows:

```
process OTHER PBUS;
begin integer I; I←6;
CREATE: new transaction to COMPUTE;
I←I−1; if I>0 then go to CREATE; cancel;
COMPUTE: wait 3200:5000; seize COMPUTER;
wait (250,250,300,300,300,300,300,400,400,400);
release COMPUTER; go to COMPUTE end;
```

Our example program is now almost complete. We precede the three processes given above by the following code, which declares the global quantities. There is also a fourth process which accomplishes the initialization and which stops the simulation after 1 hour of simulated time.

```
facility TU[6], SB[3], LINE, COMPUTER;
store 10 QUEUE[6];
integer TUSTATE[6], SBNUMBER[6], TUMESSAGE[6];
table (2000 step 500 until 15000) TABLE [6];
process MASTER CONTROL;
begin SBNUMBER[1]←1; SBNUMBER[2]←2;
      SBNUMBER[3]←1; SBNUMBER[4]←2;
      SBNUMBER[5]←1; SBNUMBER[6]←3;
wait 60×60×1000; stop end;
```

Remarks

We have purposely chosen a rather complex example to show how SOL can be used to solve an actual problem of practical importance, and to show in what a natural manner the system can be described in the language.

Fig. 3 is a sample of some of the output resulting from the program of the preceding section.

```
TU          6    SENDS MESSAGE          1     AT TIME        6586
TU          4    SENDS MESSAGE          1     AT TIME        7152
TU          5    SENDS MESSAGE          2     AT TIME        7295
      TU          6     RECEIVES REPLY AT TIME        9973
      TU          4     RECEIVES REPLY AT TIME       10305
      TU          5     RECEIVES REPLY AT TIME       13353
TU          6    SENDS MESSAGE          3     AT TIME       16908
TU          2    SENDS MESSAGE          2     AT TIME       17476
TU          5    SENDS MESSAGE          1     AT TIME       19405
      TU          6     RECEIVES REPLY AT TIME       21166
      TU          2     RECEIVES REPLY AT TIME       21412
TU          3    SENDS MESSAGE          2     AT TIME       21646
      TU          5     RECEIVES REPLY AT TIME       24229
TU          1    SENDS MESSAGE          2     AT TIME       25424
      TU          3     RECEIVES REPLY AT TIME       27959
TU          4    SENDS MESSAGE          1     AT TIME       30442
TU          5    SENDS MESSAGE          2     AT TIME       31409
      TU          1     RECEIVES REPLY AT TIME       31609
      TU          4     RECEIVES REPLY AT TIME       33278
TU          3    SENDS MESSAGE          3     AT TIME       34067
TU          2    SENDS MESSAGE          3     AT TIME       34478
      TU          5     RECEIVES REPLY AT TIME       35046
      TU          2     RECEIVES REPLY AT TIME       38958
      TU          3     RECEIVES REPLY AT TIME       39376
TU          1    SENDS MESSAGE          1     AT TIME       40472

CLOCK TIME AT END OF SIMULATION WAS   3600000

NUMBER OF TIMES LABELS WERE ENCOUNTERED

LABEL          =       COUNT        LABEL          =       COUNT        LABEL          =       COUN
      ORIGIN   =        1455              START    =        1457             SCAN     =        1730
        SEND   =        1477        COMPUTATION    =        1446           OUTPUT     =         802
     RECEIVE   =        5990             CREATE    =           6          COMPUTE     =         476

                    NAME OF FACILITY     FRACTION OF TIME IN USE

                          TU[001]                0.8318
                          TU[002]                0.8055
                          TU[003]                0.7887
                          TU[004]                0.8085
                          TU[005]                0.8302
                          TU[006]                0.7585
                          SB[001]                0.6051
                          SB[002]                0.4221
                          SB[003]                0.2120
                             LINE                0.8649
                         COMPUTER                0.5509
```

NAME OF STORE	CAPACITY	MAXIMUM USED	AVERAGE OCCUPANCY	AVERAGE UTILIZATION
QUEUE[001]	10	10	2.5272	0.2527
QUEUE[002]	10	10	2.4255	0.2426
QUEUE[003]	10	10	2.3835	0.2384
QUEUE[004]	10	7	1.7696	0.1770
QUEUE[005]	10	8	2.1844	0.2184
QUEUE[006]	10	5	1.4971	0.1497

TABLE NAME IS TABLE[003]

NUMBER OF TABLE ENTRIES 235 SUM OF ALL ENTRY VALUES 1201076

MEAN OF TABLE 5110.9617 STANDARD DEVIATION 1441.51124

UPPER LIMIT	NUMBER	PER CENT	CUMULATIVE	MULTIPLE OF MEAN
2000	0	0.00	0.00	0.3913
2500	0	0.00	0.00	0.4891
3000	5	2.13	2.13	0.5870
3500	14	5.96	8.09	0.6848
4000	36	15.32	23.40	0.7826
4500	32	13.62	37.02	0.8805
5000	46	19.57	56.60	0.9783
5500	23	9.79	66.38	1.0761
6000	25	10.64	77.02	1.1739
6500	18	7.66	84.68	1.2718
7000	12	5.11	89.79	1.3696
7500	10	4.26	94.04	1.4674
8000	5	2.13	96.17	1.5653
8500	1	0.43	96.60	1.6631
9000	4	1.70	98.30	1.7609
9500	1	0.43	98.72	1.8587
10000	1	0.43	99.15	1.9566
10500	1	0.43	99.57	2.0544
11000	0	0.00	99.57	2.1522
11500	1	0.43	100.00	2.2501
12000	0	0.00	100.00	2.3479
12500	0	0.00	100.00	2.4457
13000	0	0.00	100.00	2.5436
13500	0	0.00	100.00	2.6414
14000	0	0.00	100.00	2.7392
14500	0	0.00	100.00	2.8370
15000	0	0.00	100.00	2.9349

Fig. 3 (*pp. 406–407*)—Samples of the output obtained.

Appendix III

IEEE TRANSACTIONS ON ELECTRONIC COMPUTERS

The ideas used in SOL for creating and canceling transactions have applications in the design of languages for highly parallel computers.

The techniques which are used in the implementation of SOL will be the subject of another paper. It should be indicated here, however, that the implementation gives a rather efficient program because separate lists are kept for transactions which are waiting for different reasons. Those which are waiting for time to pass are kept sorted on the required time. Those which are waiting for a condition such as "wait until A = 0," for some global variable A, are kept in a list associated with A; this list is interrogated only when the value of A has been changed.

The SOL system has proved to be especially advantageous for simulating computer systems since "typical programs," which we assume are to be run on the simulated computers, are easily coded in SOL's language.

ACKNOWLEDGMENT

The authors wish to express their appreciation to J. Merner for many helpful suggestions.

REFERENCES

[1] H. M. Markowitz, B. Hauser, and H. W. Karr, "SIMSCRIPT—A Simulation Programming Language," Prentice-Hall, Inc., Englewood Cliffs, N. J.; 1963.
[2] G. Gordon, "A general purpose systems simulation program," *Proc. Eastern Joint Computers Conf.*, pp. 87–104; December, 1961.
[3] ——, "A general purpose systems simulation program," *IBM Systems J.*, vol. 1, pp. 18–32; September, 1962.
[4] "Reference Manual, General Purpose Systems Simulator II," IBM Corp., White Plains, N. Y.; 1963.
[5] D. E. Knuth and J. L. McNeley, "A formal definition of SOL," this issue, page 409.
[6] M. R. Lackner, "Toward a general simulation capability," *Proc. Spring Joint Computer Conference*, pp. 1–14; May, 1962.

Appendix III

A Formal Definition of SOL

D. E. KNUTH AND J. L. McNELEY

Summary—**This paper gives a formal definition of SOL, a general-purpose algorithmic language useful for describing and simulating complex systems. SOL is described using meta-linguistic formulas as used in the definition of ALGOL 60. The principal differences between SOL and problem-oriented languages such as ALGOL or FORTRAN is that SOL includes capabilities for expressing parallel computation, convenient notations for embedding random quantities within arithmetic expressions and automatic means for gathering statistics about the elements involved. SOL differs from other simulation languages such as SIMSCRIPT primarily in simplicity of use and in readability since it is capable of describing models without including computer-oriented characteristics.**

I. General Description

SOL IS an algorithmic language used to construct models of general systems for simulation in a readable form. The model builder describes his model in terms of processes whose number and detail are completely arbitrary and definable within the constraints of the language elements. A SOL model consists of a number of statements and declarations which have a character similar to that found in programming languages such as ALGOL.

The model is not built to be executed in a sequential fashion as ordinary programming languages require. Rather, the processes are written and executed as if all were running in parallel. Control between processes is maintained by the interaction of *global* entities and by control and communication instructions within the different processes. At the initiation of the simulation all processes are begun simultaneously.

Variables declared within a process are called *local* variables. Within a given process it is possible to have several actions going on at once; therefore, we may think of several objects on which the action takes place each in its own place in the process at any given time. These objects will be referred to as *transactions*. A set of local variables corresponding in number to those declared in the process is "carried with" each transaction of that process. Transactions situated within one process may not refer to the local variables of another process nor to the local variables of another transaction in the same process.

Global quantities are of three major types: global variables, facilities and stores. *Global variables* can be referenced or changed by any transaction from any process in the system, and the variable possesses only one value at any given time.

Manuscript received January 3, 1964.
D. E. Knuth is with the California Institute of Technology, Pasadena, Calif.
J. L. McNeley is with the Burroughs Corporation, Pasadena, Calif.

A *facility* is a global element which can be controlled by only one transaction at a time. Associated with each request for the facility is a "control strength," and if a requesting transaction has a higher strength than the transaction controlling the facility, an interrupt will occur. Interrupts may be nested to any depth. If the requesting transaction is not of greater strength than the controlling transaction, then the requesting transaction stops and waits for the facility until the controlling transaction releases its control. When a transaction is interrupted, it cannot advance to any other position in its program until it regains control of the facility.

Stores are space-shared rather than time-shared global elements, and they are assigned a specific storage capacity. As long as there is sufficient storage to accommodate the requesting transaction the request for space is satisfied; otherwise, the transaction waits until the space it is requesting becomes available. In this sense, a facility may be regarded as a store which has a capacity of one unit only, except for the fact that no interrupt capability is provided for stores.

Simulated time passes in discrete units indicated in "wait statments." The model builder requires the transactions to wait a proper number of time units at the appropriate places in the processes, and this specifies the time element. The interpretation of the physical significance of a unit of time is immaterial in the SOL language; if all time interval specifications are multiplied by a factor of ten it will not decrease the speed by which the model is simulated.

Control within or between processes is also introduced into the simulation by allowing a transaction to wait until a global variable or expression obtains a certain value. A transaction may also be forced to wait until a space- or time-shared element attains a certain status.

Output statements which display the progress of the simulation may be inserted at will in the model. Special types of statistics are automatically available, such as the per cent of utilization of a facility, the average and maximum number of elements in a store at a given moment, etc. Another type of global quantity, called a *table*, is introduced to record statistical information about desired data. The mean, the standard deviation and a histogram are provided for all data recorded in a table.

Processes initiate parallelism within themselves by using a duplication operation. The transaction makes an exact copy of itself and sends the copy to a specified location in the process while the original continues in sequence. A transaction is taken out of the system when it executes a "cancel" statement.

Appendix III

Other operations available in SOL are similar to those of existing algorithmic languages, but these portions of the language are at the present time less powerful than the features available in a large scale programming language.

A detailed example of a complete SOL model appears in a companion paper in this issue [2].

II. Syntax and Semantics of SOL

We will define the syntax of SOL using meta-linguistic formulas as given in the definition of ALGOL 60 [1]. Certain things which have been carefully defined in ALGOL 60 will not be redefined here but will merely be stated to have the same interpretation as given by ALGOL. We will use the abbreviation *⟨A⟩* to mean "a list of ⟨A⟩," *i.e.*,

⟨A⟩::=⟨A⟩|*⟨A⟩*, ⟨A⟩

Comments may be written in the form **"comment** ⟨string without semicolons⟩;" as in ALGOL 60.

A. Identifiers and Constants

⟨letter⟩::=A|B|C|D| · · · |Z
⟨digit⟩::=0|1|2|3| · · · |9
⟨number⟩::=⟨constant⟩|⟨decimal constant⟩
⟨constant⟩::=*⟨digit⟩*
⟨decimal constant⟩::=⟨constant⟩.⟨constant⟩
⟨identifier⟩::=⟨letter⟩|⟨identifier⟩⟨letter⟩|
⟨identifier⟩⟨digit⟩

Identifiers are used as the names of variables, statistical tables, stores, facilities, processes, procedures and statements. The same identifier can be used for only *one* purpose in a program. Constants are used to represent integer numbers. Decimal constants represent real numbers. Identifiers must be declared before they are used elsewhere.

B. Declarations

⟨declared item⟩::=⟨identifier⟩|⟨identifier⟩[⟨constant⟩]
⟨variable declaration⟩::=**integer** *⟨declared item⟩*|
real* ⟨declared item⟩*
⟨facility declaration⟩::=**facility** *⟨declared item⟩*
⟨store declaration⟩::=**store** *⟨constant⟩⟨declared item⟩*
⟨table declaration⟩::=**table** *(⟨number⟩**step**⟨number⟩
until ⟨number⟩)⟨declared item⟩*
⟨monitor declaration⟩::=**monitor** *⟨identifier⟩*

If the declared item is simply an identifier, it means that a single item of that name is being declared. The other form, *e.g.*, A[10], means 10 similar items called A[1], A[2], · · · , A[10] are being declared.

The variable declaration is used to specify variables (either local or global, depending on where the declaration appears). All variables are initially set to zero when declared. "Integer" variables differ from "real" variables in that when a value is assigned to them it is rounded to the nearest integer.

When a facility is declared, it is initially "not busy"; at the end of the simulation run, statistics are reported giving the per cent of time each facility was in use.

A store declaration gives the capacity of each store (the number preceding the identifier). At the end of the simulation run statistics are given on the average and the maximum number of items occupying the store (as a function of time). Stores are empty when first declared.

A "table" is used to gather detailed statistical information of any desired type; readings are tabulated and afterwards the mean, the standard deviation, histogram distribution, etc., are output. The constants preceding the table name give the starting point for histogram intervals, the increment between intervals and the highest value.

A monitor declaration names items which already have been declared, with the understanding that these identifiers are to be "monitored." This means that whenever a change in the state of the corresponding quantity is detected, a line will be printed giving the details. This capability is especially useful when checking out a model, and it can also be used to advantage for output during a regular simulation run.

C. Expressions and Relations

⟨name⟩::=⟨identifier⟩|⟨identifier⟩[⟨expression⟩]

By ⟨variable name⟩, ⟨facility name⟩, etc., we will mean that the identifier in the name has appeared in a ⟨variable declaration⟩, ⟨facility declaration⟩, etc., respectively.

⟨primary⟩::=⟨variable name⟩|⟨store name⟩|
⟨constant⟩|⟨decimal constant⟩|**time**|
(*⟨expression⟩*)|**abs**(⟨expression⟩)|
max(*⟨expression⟩*)|**min**(*⟨expression⟩*)|
normal(⟨expression⟩, ⟨expression⟩)|
exponential(⟨expression⟩)|**poisson**(⟨expression⟩)|
geometric(⟨expression⟩)|**random**
⟨term⟩::=⟨primary⟩|⟨term⟩×⟨primary⟩|
⟨term⟩÷⟨primary⟩|⟨term⟩/⟨primary⟩|
⟨term⟩**mod**⟨primary⟩
⟨sum⟩::=⟨term⟩|+⟨term⟩|−⟨term⟩|⟨sum⟩+⟨term⟩|
⟨sum⟩−⟨term⟩
⟨unconditional expression⟩::=⟨sum⟩|⟨sum⟩:⟨sum⟩
⟨expression⟩::=⟨unconditional expression⟩|
if ⟨relation⟩ **then** ⟨expression⟩ **else** ⟨expression⟩

The meaning of the arithmetical operations inside expressions is identical to the meaning in ALGOL 60.

The new elements here are "*a* **mod** *b*," the positive remainder obtained upon dividing *a* by *b*; "max(e_1, · · · , e_n)" and "min(e_1, · · · , e_n)," which denote the maximum and minimum values, respectively, of the n expressions; and there are also notations for expressing random values. The expression "(e_1, · · · , e_n)" indicates that a random selection is made from among the n expressions with equal probability of choosing any

expression. The expressions normal(M, S), poisson(M), geometric(M) and exponential(M) indicate random values with special distributions which occur frequently in applications. A random number drawn from the normal distribution with mean M and standard deviation S is denoted by normal(M, S) and is a real (not necessarily integer) value. A number drawn from the exponential distribution with mean M is denoted by exponential(M) and is also of type real. The poisson distribution signified by poisson(M), on the other hand, yields only integer values; the probability that poisson(M) = n is $(e^{-M}M^n/n!)$. The geometric distribution with mean M, denoted by geometric(M), also yields integer values, where the probability that geometric(M) = n is $1/M(1-1/M)^{n-1}$. The symbol random denotes a random real number between 0 and 1 having uniform distribution. Finally, we have the notation $e_1{:}e_2$, which denotes a random integer between the limits e_1 and e_2; more formally

$$e_1{:}e_2 = \begin{cases} 0, & e_1 > e_2 \\ (e_1, e_1+1, \cdots, e_2) & e_1 \leqq e_2. \end{cases}$$

The normal, exponential, poisson and geometric distributions are mathematically expressible in terms of random as follows:

$$\text{normal(M,S)} = S \times \sqrt{-2 \ln (\text{random})} \times \sin (2\pi \, \text{random}) + M$$

$$\text{exponential(M)} = -M \ln (\text{random})$$

$$\text{poisson(M)} = n \text{ if } e^{-M}\left(1 + M + \frac{M^2}{2!} + \cdots + \frac{M^{n-1}}{(n-1)!}\right) \leqq \text{random} < e^{-M}\left(1 + M + \cdots + \frac{M^n}{n!}\right)$$

$$\text{geometric(M)} = \left[1 + \ln (\text{random}) / \ln \left(1 - \frac{1}{M}\right)\right].$$

(The poisson distribution should not be used for values of M greater than 10.) As examples of the use of these distributions, consider a population of customers coming to a market with an average of one customer every M minutes. The distribution of waiting time between successive arrivals is exponential(M). On the other hand, if an average of M customers come in per hour, the distribution of the actual number of customers arriving in a given hour is poisson(M). If an individual performs an experiment repeatedly with a chance of success, 1/M on each independent trial, the number of trials needed until he first succeeds is geometric(M).

The special symbol "time" indicates the current time; intially, time is zero. The value of a store name is the current number of occupants of the store.

⟨relational operator⟩::= = | ≠ | < | ≦ | > | ≧
⟨relation primary⟩::= ⟨unconditional expression⟩
⟨relational operator⟩⟨unconditional expression⟩ |
⟨facility name⟩ **busy** | ⟨facility name⟩ **not busy** |
⟨store name⟩ **full** | ⟨store name⟩ **not full** |
⟨store name⟩ **empty** | ⟨store name⟩ **not empty** |
pr(⟨expression⟩) | (⟨relation⟩)
⟨relation⟩::= ⟨relation primary⟩ |
⟨relation primary⟩∨⟨relation primary⟩ |
⟨relation primary⟩∧⟨relation primary⟩ |
¬⟨relation primary⟩

These relations have obvious meanings except for the construction "pr(e)" which stands for a random condition which is true with probability e. (Here e must be less than or equal to 1.) Thus we might say

if pr(0.12) **then** (12 per cent of the time)
else (88 per cent of the time).

III. Statements

A. *Processes*

As this simulator operates, any number of processes written in the language may be in use at once. We may think of several objects, each in its own place in the process at any given time. These objects are referred to as *transactions*. In this section, we describe the various manipulations that transactions can perform in the language.

⟨process description⟩::= **process** ⟨identifier⟩;
⟨statement⟩ |
process ⟨identifier⟩; **begin**
⟨process declaration list⟩; ⟨statement list⟩ **end**
⟨process declaration⟩::= ⟨variable declaration⟩ |
⟨procedure declaration⟩ | ⟨monitor declaration⟩
⟨process declaration list⟩::= ⟨process declaration⟩ |
⟨process declaration list⟩; ⟨process declaration⟩

There are two kinds of variables, *global* variables (not declared in a process) and *local* variables (those which are declared in a process). All transactions can refer to the global variables, and a global variable has only one value at any given time. But a local variable is "carried with" each transaction within a given process, and there is in general, a different value for a local variable depending on which transaction is using it. Transactions situated within one process may not refer to the local variables of another process, nor can the local variables of one transaction within a process be reached directly by other transactions in that same process. Communication between processes is accomplished solely with the help of global quantities.

B. *Labels*

A statement may be named by any identifier as follows:

⟨statement⟩::= ⟨unlabeled statement⟩ |
⟨identifier⟩:⟨statement⟩

By the designation ⟨label⟩ we will mean the name of a statement.

C. Creation of Transactions

At the beginning of simulation, there is one transaction present for each process described. Each of these initial transactions starts at time zero and is positioned at the beginning of the process. More transactions may be created by using "start statements."

⟨start statement⟩::=**new transaction to** ⟨label⟩

This statement, when executed, creates a new transaction (whose local variables are the same in number and value as those of the transaction which created it). The new transaction begins executing the program at ⟨label⟩ while the original transaction continues in sequence. New transactions are also created by input statements (Section III-T).

D. Disappearance of Transactions

Transactions "die" when they execute a cancel statement.

⟨cancel statement⟩::=**cancel**

An implied cancel statement is at the end of every process, so cancel statements need not always be explicitly written.

E. Replacement Statements

⟨replacement statement⟩::=⟨variable name⟩ ←⟨expression⟩

This replaces the value of the variable by the value of the expression. The variable may be global or local, but not the name of a store. If the variable is an integer variable, the expression is rounded.

F. Priority

Time is measured in discrete units, so it may happen that by coincidence two transactions want to do something at precisely the same time. They may be in conflict, *e.g.*, they may both want to seize a facility, or to change the value of the same global variable or one may want to change it while the other is using its value. Actually, in such cases of conflict, the simulator does choose a specific order for execution; no two things actually happen at the same instant, as we deal more properly with *infinitesimal* units of time between the discrete units. The choice of order is fairly arbitrary except when a difference of priority is specified; in that case, the transaction with *higher* priority will be acted on first. Each transaction has a priority, which is initially zero; priority is changed by the statement

PRIORITY←⟨expression⟩.

The declaration "**integer** PRIORITY" is implied at the beginning of each process, *i.e.*, PRIORITY is treated as a local variable. In the present implementation of SOL, the priority must be between 0 and 63. The effect of priority is spelled out further in Section IV.

G. Wait Statements

⟨wait statement⟩::=**wait** ⟨expression⟩

The expression is rounded to the nearest integer, and then this statement advances "**time**" by max(0, ⟨expression⟩), as far as this transaction is concerned. All time delays in a simulated process are, in the last analysis, specified by using wait statements.

H. Wait-Until Statements

⟨wait-until statement⟩::=**wait until** ⟨relation⟩

This causes the transaction to freeze at this point until the relation becomes true (because of action by other transactions). The relation must not involve expressions which have a random value; *e.g.*, it is not legal to write "**wait until** pr(10)" or "**wait until** A[1:4]=0," etc.

I. Enter Statements

⟨enter statement⟩::=**enter** ⟨store name⟩|
enter ⟨store name⟩, ⟨expression⟩

The first form is an abbreviation for "**enter** ⟨store name⟩, 1." The value of the expression, rounded to the nearest integer, gives the number of units requested of the store. The transaction will remain at this statement until that number of units is available and until all other transactions of greater or equal priority which have been waiting for storage space have been serviced.

J. Leave Statement

⟨leave statement⟩::=**leave** ⟨store name⟩|
leave ⟨store name⟩, ⟨expression⟩

The first form is an abbreviation for "**leave** ⟨store name⟩, 1." This statement returns the number of units equivalent to the value of the (rounded) expression.

K. Seize Statements

⟨seize statement⟩::=**seize** ⟨facility name⟩|
seize ⟨facility name⟩, ⟨expression⟩

The first form is equivalent to "**seize** ⟨facility name⟩, 0." This statement is usually rather simple, but there are situations when complications arise. If the facility is not busy when this statement occurs, then it becomes busy at this point and remains busy until later released by this transaction. (Note: If this transaction creates another transaction by means of a start statement, the new transaction does not control the facility.)

The expression appearing above represents the "control strength" which is normally zero. Allowance is made, however, for one transaction to interrupt another. If the facility is busy when the seize statement occurs, let E_1 be the control strength with which the facility was seized and let E_2 be the control strength of this seize statement. If $E_2 \leq E_1$, the transaction waits until the facility is not busy. If $E_2 > E_1$, however, *interrupt* occurs. The transaction T_1 which had control of

Appendix III

the facility is stopped wherever it was in its program, and the present transaction T_2 seizes the facility. When T_2 releases the facility, the following occurs:

1) If T_1 was executing a wait statement when interrupted, the time of wait is increased by the time which passed during the interrupt.
2) There may be several transactions not waiting to seize this facility. If any of these has a higher control strength than E_1, then T_1 is interrupted again. The transaction which interrupts is chosen by the normal rules for deciding who obtains control of a facility upon release, as described in the next section.

The control strength in the present implementation of SOL must be an integer between 0 and 4095. This allows interrupts to be nested up to 4095 deep.

L. Release Statements

⟨release statement⟩::=**release** ⟨facility name⟩

This statement is permitted only when the transaction is actually controlling the facility because of a previous seizure. When the facility is released, there may be several other transactions waiting because of seize statements. In this case, the one which gets control of the facility next is chosen by a consideration of the following three quantities in order:

1) highest control strength,
2) highest PRIORITY,
3) first to request the facility.

M. Go To Statements

⟨go to statement⟩::=**go to** ⟨label⟩|
go to (*⟨label⟩*), ⟨expression⟩

This statement is used to transfer to another point in the program; statements are usually executed sequentially. In the second form, the expression is used to select which statement to transfer to; if there are n labels, the expression, when rounded to the nearest integer, must have a value between 0 and n. Zero means continue in sequence, 1 means go to the first statement mentioned, and so on.

N. Compound Statements

Several statements may be combined into one, as follows:

⟨statement list⟩::=⟨statement⟩|⟨statement list⟩;
⟨statement⟩
⟨compound statement⟩::=**begin** ⟨statement list⟩ **end**|
(⟨statement list⟩)

O. Conditional Statements

⟨conditional⟩::=**if** ⟨relation⟩ **then** ⟨unconditional statement⟩|
if ⟨relation⟩ **then** ⟨unconditional statement⟩ **else** ⟨statement⟩

The meaning is the same as in ALGOL; testing of the relation requires no simulated time.

P. Tabulate Statements

⟨tabulate statement⟩::=**tabulate** ⟨expression⟩ **in** ⟨table name⟩

The value of the expression is recorded as a statistical observation in the table specified.

Q. Output Statements

⟨carriage control⟩::=⟨empty⟩|**page**|**line**|**double**
⟨string⟩::=⟨any sequence of characters excluding "#"⟩
⟨output list item⟩::=#⟨string⟩#|⟨expression⟩|
⟨store name⟩|⟨table name⟩|⟨facility name⟩
⟨output statement⟩::=**output** *⟨carriage control⟩
⟨output list item⟩*

Output occurs for all items listed, in turn, after doing the appropriate carriage control positioning. The output for a string is the string itself. An output for an expression is the value. For a store, table or facility, the appropriate statistical information is output. At the conclusion of an output statement, the final line is printed out.

R. Stop Statements

⟨stop statement⟩::=**stop**

A stop statement causes simulation to terminate immediately, and all transactions cease. The statistics for all stores, tables and facilities are output as in the output statement, as well as the final time, the number of times each labeled statement was referenced and the number of transactions which appeared in each process.

S. Procedures

⟨procedure declaration⟩::=**procedure** ⟨identifier⟩;
⟨statement⟩
⟨procedure statement⟩::=⟨procedure name⟩

A procedure is simply a subroutine used to save coding. Parameters are not allowed, but their effect can be achieved by setting local variables in the transactions before calling the procedure. There are local procedures and global procedures (the latter are declared outside of a process). Global procedures cannot refer to local variables. A go to statement may not lead out of a procedure body. Procedures may be used recursively.

T. Transaction Input-Output

⟨transaction read statement⟩::=**read** ⟨constant⟩ **to** ⟨label⟩
⟨transaction write statement⟩::=**write** ⟨constant⟩

The read statement inputs a set of values of local variables for a transaction of the same type as the one executing the read statement; this set of values is used in the creation of a new transaction which begins exe-

Appendix III

IEEE TRANSACTIONS ON ELECTRONIC COMPUTERS

cuting the program at the statement mentioned. The write statement writes the current values of the local variables of the transaction onto the unit specified and does not cancel the present transaction. The constant in each refers to a tape or card unit number. The same tape should not be used for both input and output in the same simulation run.

U. Summary of Statements

⟨unlabeled statement⟩: : = ⟨unconditional statement⟩|
⟨conditional⟩
⟨unconditional statement⟩: : = ⟨start statement⟩|
⟨cancel statement⟩|
⟨replacement statement⟩| ⟨wait statement⟩|
⟨wait-until statement⟩| ⟨enter statement⟩|
⟨leave statement⟩| ⟨seize statement⟩|
⟨release statement⟩| ⟨go to statement⟩|
⟨compound statement⟩| ⟨output statement⟩|
⟨tabulate statement⟩| ⟨stop statement⟩|
⟨transaction read statement⟩| ⟨procedure statement⟩|
⟨transaction write statement⟩| ⟨empty⟩

IV. The Model as a Whole

⟨model⟩: : = **begin** ⟨global declaration list⟩; ⟨process list⟩ **end.**
⟨declaration⟩: : = ⟨variable declaration⟩|
⟨facility declaration⟩|
⟨store declaration⟩| ⟨table declaration⟩|
⟨monitor declaration⟩| ⟨procedure declaration⟩
⟨global declaration list⟩: : = ⟨declaration⟩|
⟨global declaration list⟩; ⟨declaration⟩
⟨process list⟩: : = ⟨process sdescription⟩|
⟨process list⟩; ⟨process description⟩

Initially all variables are zero, all facilities are "not busy," all stores are "empty," the time is zero, one transaction appears for each process described and the simulator is in the "choice state."

When the simulator is in "choice state," each transaction is either positioned at a wait statement, a wait-until statement, a seize or enter statement or else it has just been created. (We will dispense with the latter case by assuming a "wait 0" statement has been inserted just before the present position when a new transaction is created.) If there are no transactions which can move at this time, the time is advanced to the earliest completion time for a wait statement. Now, from the set of transactions able to move, that one is selected which has the highest PRIORITY, and in case of ties, which has been waiting the longest. (If there is still a tie, an arbitrary choice is made.) The selected transaction is activated, and it continues to execute its statements until encountering a cancel or stop statement, a priority assignment statement, a wait statement, a wait-until statement with a false relation or a seize or enter statement which cannot take place at that time. We examine all other transactions which are stopped because of a wait-until statement involving global quantities changed by the present transaction. If the corresponding relation is now true, these transactions become free to move at the current time. Then we have once again reached "choice state." Note that all release statements which are passed during the time the selected transaction was moving are processed immediately in such a way that the facility becomes not busy only if no other transaction were interrupted or were waiting to seize it; if other transactions *are* in the latter category, the choice of successor and the transfer of control described in Section III-L takes place immediately as the release statement is executed. Therefore, it is conceivable that the statement "wait until FAC not busy" may never be passed if other transactions are always ready to seize the facility FAC. Similar remarks apply to the leave statements.

Since this paper was written, a few additions have been made to the SOL language, including "synchronous" variables and some additional diagnostic capabilities.

References

[1] "Revised report on the algorithmic language ALGOL 60," *Comm. ACM*, pp. 1–17; January 6, 1963.
[2] D. E. Knuth and J. L. McNeley, "SOL—A symbolic language for general-purpose systems simulation," this issue, page 401.

ERRATA for SOL-20

Since the publication of the SOL-20 manual, changes have been made to improve the SOL system as implemented. The major changes were: (1) the addition of procedures which may contain SOL statements, (2) changing TIME to type real instead of integer, (3) increasing the run time speed of the system, and (4) eliminating the need to recompile the SOL system for every SOL program. The system is now callable from a job card under the name 'SOL'.

As a consequence of the aforementioned improvements, some of the documentation is incorrect. This errata explains all the changes and additions to the manual.

NOTE: In this errata, positive line numbers (e.g. 7, 11, etc.) indicate lines from the top of the page; negative numbers (e.g. -10, -7, etc.) indicate lines from the bottom.

Page 2.1

line -10:

<table declaration> ::= real array <identifier>

line -7:

of the table plus 1 while the...

line -6:

...the same identifier. This number is 1 if there is only one table.

line -1:

line 9: real array TAB[1:2,0:28,1:6]; | tables

Page 2.3

line 7:

is a global variable. It is of type real, not integer.

line 11:

real procedure RAND(N,E); value N; integer N;

line 12:

real array E;

line -11:

poisson (M) = 0 if random < e^{-M}

Page 2.4

lines 2,3:

If the result is to be of type integer then COLON is used.

integer procedure COLON (E1,E2); value E1,E2; integer E1, E2;

otherwise

real procedure UNIFORM (E1,E2); value E1,E2; real E1,E2;

line 5:

line 36: Q ← COLON(1,6);

Page 2.5

line 2:

Processes are declared by a switch statement

line 4:

switch < switch identifier > := *<process identifier>*-

Page 2.5 (cont)

line 18:

...name of transactions appearing in a new transaction statement must be

Page 2.6

line 15:

$2^{15} - 1$.

line -10:

procedure WAIT(WAIT.TIME); value WAIT.TIME; real WAIT.TIME;

Page 2.8

line 5:

real EXPRESSION: integer Q; real array TABLE;

line -12:

procedure STOP (<label>);

line -7:

<label> and occurs...

line -4:

line 25: STOP(RESULTS);

Page 2.9

line -15:

real array TABLE; string NAME;

lines -6, 5:

A SOL-procedure is defined to be exactly the same as an ALGOL-20 procedure, except it contains a wait, wait until, seize, enter, or cancel statement. The SOL-procedure is thus constructed the same as Algol procedure. However, there are some restrictions on the structure and usage of procedures containing the aforementioned SOL statements.

1. No own variables may be declared as local variables in the SOL-procedure body.

2. No arrays may be declared as local variables in the SOL-procedure body. However, arrays may be passed as parameters.

3. Local variables are undefined in the usual Algol-20 sense, i.e., upon entering the procedure, their values are undefined.

4. SOL-procedures may contain nested blocks which in turn may contain SOL code. No arrays may be declared in the block.

5. A SOL-procedure may not be used as an input parameter to a procedure, SOL or Algol. Also, a SOL-procedure cannot be called from any procedure. In general then, a SOL-procedure may not be nested inside another procedure.

6. for statements with SOL statements in their scopes must not have multiple for lists.

7. A go to statement may lead out of a procedure body, but must be into the transaction that called the procedure.

8. A NEW.TRANSACTION or a READIN statement cannot occur in the procedure body.

9. SOL-procedures and Algol procedures are non-recursive.

10. SOL-procedures may have input parameters.

Page 2.10

lines 13-21:

The SOL system is written in ALGOL and has been precompiled. Therefore, the SOL program's first card must have an 'AL' in the language field and the program must provide the matching 'END' (Since the SOL system is an ALGOL program, it starts with a BEGIN). The system is called from a job card using the mnemonic SOL. The integer array for process priorities, the switch statement for processes and all facilities, stores, and global SOL variables must be declared. Facilities and stores that are simple variables must be declared as real variables.

All labels used as parameters to the procedures NEW.TRANSACTION, STOP, and READIN must be declared...

line -9:

... Also the vector for process priorities must be initialized.

line -7:

START.SIMULATION(<simple variable>,<array identifier>,<switch identifier>);

The <simple variable> is the last global scalar declared in the program before the procedure call START.SIMULATION. If procedure declarations immediately precede this call, then some scalar must be declared between the last declaration and the call (The order of array and scalar declarations is not relevant). The <array identifier> is the vector declared previously that will contain the priorities for the processes. The <switch identifier> is the identifier of the switch statement previously declared that lists the processes.

Page 4.1

Add to the lists of errors:

Error 1: PROCEDURE CALL NESTED INSIDE A PROCEDURE

A SOL-procedure is called from another procedure or SOL-procedure.

The following errors have been changed:

Error 2, Error 3, Error 4, Error 5: TABLE SPACE EXCEEDED

The simulation has exhausted the space allocated by the system for tables.

Error 8: TRANSACTION INTERRUPTING ITSELF

A transaction presently controlling a facility is reseizing the facility with a higher control strength.

Page 4.2

lines -2, 1:

the outermost block of the SOL program:

RUN.ERROR(<label>,'TIMR');

where <label> is the start of the procedure calls for outputting the statistical results of the simulation.

Page 5.1

The following identifiers are no longer reserved:

MOVE, MT.TAB, PROCESSES, PUSH, RESULTS, RETURN

Page 5.1 (cont)

The following identifiers are now reserved:

B.LINK1, B.LINK2, COLON, HH2, T.CNT, UMARK, X1, X2

Page I.1

line 10:

real array *<identifier>[1:2

line 23:

UNIFORM....; COLON(<number>, <number>);

Page I.2

line 4:

switch <switch identifier> := *<process

line 6:

integer array <array identifier>[1:<number>];

line -10:

STOP(<label>);

add: procedure <identifier> <procedure statement> Same as in Algol. For SOL-procedure restrictions, see SOL.2.9, section 5 on procedures.

Add page II.0

A few critical features of the test program are worthy of mention.

- the program does not start with a BEGIN, this is provided by the system.
- the program provides the matching END.
- the first card of the program must have an 'AL' in the language field.

Additional statistical information is now automatically provided by the system.

1. facilities:
   - utilization: the number of times the facility was seized, not including interrupts, over the simulation run.
   - average utilization: total time the facility was busy divided by the utilization.

2. stores:

   - number of entries: the total number of units entered in the store.

3. tables:

   - deviation from mean: the upper limit of the histogram interval minus the mean, all divided by the standard deviation.
   - all entries that exceed the last histogram limit are indicated by the word OVERFLOW printed in the limit field. Anything less than the first limit goes into the first cell.

The average occupancy of a store is calculated as follows: Let

L = length of time for which the store remained at its current value.

U = total unit-occupancy = $\Sigma L \cdot$ (storage contents)

Then average occupancy = U/ total time of run

Hence, this figure is a weighted average of the number of transactions in the store. Average utilization is then average occupancy divided by the capacity.

Page II.1

The following 'program' will correct the test program.

```
ALTER 1 TO 2;
            REAL LINE, COMPUTER;  | FACILITIES
ALTER 5 TO 6;
            REAL ARRAY TU[1:6], SB[1:3];  | FACILITIES
ALTER 9;
            REAL ARRAY TAB[1:2,0:28,1:6];  | TABLES
INSERT AFTER 11;
            LABEL RESULTS;
ALTER 19;
            START.SIMULATION(I,PROC.PRIO,PROCESSES);
ALTER 25;
            WAIT(15*60*1000); STOP(RESULTS);  | STOP SIMULATION
ALTER 28 TO 29;
            INTEGER Q, MESSAGE.TYPE; REAL START.TIME;
            REAL ARRAY TYPE[1:10];
ALTER 36;
            START:  Q ← COLON(1,6); ENTER(QUEUE[Q],1);
```

```
ALTER 89 TO 90;
              INTEGER I; REAL ARRAY WAITS[1:10];
              WAITS[1] ← WAITS[2] ← 250;
```