1965

SOL-20

Gilbert J. Hansen
Carnegie Mellon University

Follow this and additional works at: http://repository.cmu.edu/compsci

This Technical Report is brought to you for free and open access by the School of Computer Science at Research Showcase @ CMU. It has been accepted for inclusion in Computer Science Department by an authorized administrator of Research Showcase @ CMU. For more information, please contact research-showcase@andrew.cmu.edu.
NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.
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.
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>

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

\[ \text{STORE}( \text{store identifier}, \text{constant} ); \]

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

\[ \text{TABLE}( \text{number}, \text{table identifier}, \text{number}, \text{number}, \text{number} ); \]

The first \( \text{number} \) represents the table number, i.e., the desired table specified by the table identifier. The other three \( \text{number} \)'s give the starting point for histogram intervals, the increment between intervals, and the highest value, respectively.

Example:

```sol
lines 13-16: for \( I \leftarrow 1 \) step 1 until 6 do begin
    \text{STORE}( \text{QUEUE} [I], 10); \text{declare stores}
    \text{TABLE}(1, \text{TAB}, 2000, 500, 15000); \text{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. \( e_1 \leq e_2 \):

integer procedure \textsc{uniform}(e_1, e_2); value \( e_1, e_2 \);

Examples:
- line 36: \( q \leftarrow \textsc{uniform}(1,6) \);
- line 40: \( \text{wait} \textsc{uniform}(6000,8000) \);

2. relations

a. \langle \text{facility name} \rangle \text{ busy}, \langle \text{facility name} \rangle \text{ not busy}:

boolean procedure \text{busy}(\text{facility}); \ real \text{facility};

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

Example:
- line 59: \textbf{if} \text{busy}(\text{SB3}) \textbf{then begin}

b. \langle \text{store name} \rangle \text{ full}, \langle \text{store name} \rangle \text{ not full}:

boolean procedure \text{full}(\text{store}); \ real \text{store};

If the facility is full, the value of the function is \textbf{true}, otherwise it is \textbf{false}.

c. \langle \text{store name} \rangle \text{ empty}, \langle \text{store name} \rangle \text{ not empty}:

boolean procedure \text{empty}(\text{store}); \ real \text{store};

If the store is empty, the value of the function is \textbf{true}, otherwise it is \textbf{false}.

d. \text{pr}(\langle \text{expression} \rangle ):

boolean procedure \text{pr}(\text{prob}); \ value \text{prob}; \ real \text{prob};

The value of the function is \textbf{true} with probability given by the value of the variable \text{prob}.

Example:
- line 65: \textbf{if} \text{pr}(0.02) \textbf{then begin}

\textsc{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.:

\[ \langle \text{process identifier} \rangle : \text{begin} \langle \text{process declaration list} \rangle ; \langle \text{statement list} \rangle \text{end}; \]
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.,


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

\[ \text{PRIORIY} \leftarrow \langle \text{expression} \rangle; \]

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.PRI01 :nj 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

```plaintext
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

```plaintext
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 \([0] = 0\));
```
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 (\x{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

```plaintext
procedure TABULATE(EXPR, 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,'TU1',20,'RECEIVES REPLY AT TIME',18,79,18,E>);
```

R. Stop Statements

```plaintext
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

```plaintext
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 \( \neq 0 \), 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 \( I < 6 \) step 1 until 6 do
PRT.FAC(I, TU[I], 'U');
line 106: PRT.FAC(0, LINE, 'LINE');

2. statistics for stores

```algl20
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 \( I < 6 \) step 1 until 6 do
PRT.ST(I, QUEUE[I], 'QUE');

3. statistics for tables

```algl20
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 \( I < 6 \) 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.

```algol
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.PRI0 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.PRI0 must be initialized.

The simulation is started by the procedure call:

```algol
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.
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.

<table>
<thead>
<tr>
<th>Trace switch</th>
<th>Traces</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRACE[1]</td>
<td>everything</td>
</tr>
<tr>
<td>TRACE[2]</td>
<td>new transaction statements</td>
</tr>
<tr>
<td>TRACE[3]</td>
<td>cancel statements</td>
</tr>
<tr>
<td>TRACE[4]</td>
<td>seize statements</td>
</tr>
<tr>
<td>TRACE[5]</td>
<td>release statements</td>
</tr>
<tr>
<td>TRACE[6]</td>
<td>enter statements</td>
</tr>
<tr>
<td>TRACE[7]</td>
<td>leave statements</td>
</tr>
<tr>
<td>TRACE[8]</td>
<td>wait statements</td>
</tr>
<tr>
<td>TRACE[9]</td>
<td>wait until statements</td>
</tr>
</tbody>
</table>

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.
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.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Normal</th>
<th>Slt</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUSY</td>
<td>NORMAL</td>
<td>SLT</td>
</tr>
<tr>
<td>CANCEL</td>
<td>POISSON</td>
<td>SRT</td>
</tr>
<tr>
<td>CTD</td>
<td>POP</td>
<td>S.TAB</td>
</tr>
<tr>
<td>EC</td>
<td>PR</td>
<td>START.SIMULATION</td>
</tr>
<tr>
<td>EMPTY</td>
<td>PRIORITY</td>
<td>STOP</td>
</tr>
<tr>
<td>ENTER</td>
<td>PROCESSES</td>
<td>STORE</td>
</tr>
<tr>
<td>ERROR</td>
<td>PROC.PRIOR</td>
<td>TABLE</td>
</tr>
<tr>
<td>E.TAB</td>
<td>PRT.FAC</td>
<td>TABULATE</td>
</tr>
<tr>
<td>EXPONENTIAL</td>
<td>PRT.ST</td>
<td>TC</td>
</tr>
<tr>
<td>FULL</td>
<td>PRT.TAB</td>
<td>TIME</td>
</tr>
<tr>
<td>GEOMETRIC</td>
<td>PUSH</td>
<td>TRACE</td>
</tr>
<tr>
<td>HZ</td>
<td>RAND</td>
<td>TR.OS</td>
</tr>
<tr>
<td>LBIT</td>
<td>RANDOM</td>
<td>T.TAB</td>
</tr>
<tr>
<td>LEAVE</td>
<td>READIN</td>
<td>UBIT</td>
</tr>
<tr>
<td>MASTER.CONTROL</td>
<td>RELEASE</td>
<td>UNIFORM</td>
</tr>
<tr>
<td>MAXAD</td>
<td>RESULTS</td>
<td>WAIT</td>
</tr>
<tr>
<td>MOVE</td>
<td>RETURN</td>
<td>WAIT.UNTIL</td>
</tr>
<tr>
<td>MTC</td>
<td>RUN.ERROR</td>
<td>WC</td>
</tr>
<tr>
<td>MT.TAB</td>
<td>SC</td>
<td>W.TAB</td>
</tr>
<tr>
<td>NEW.TRANSACTION</td>
<td>SCAL</td>
<td>WUC</td>
</tr>
<tr>
<td></td>
<td>SEIZE</td>
<td>WU.TAB</td>
</tr>
</tbody>
</table>
Appendix I
Summary of SOL-20 Differences

I. Declarations

SOL
- store *<constant><declared item>*
- table *(<number>)<step(<number>) until <declared item>*
- facility *(<declared item>)*

SOL-20
- real *(<identifier>)*
- real array *(<identifier>)*
- STORE(<declared item>, <constant>);
- half array *(<identifier>[[1:2, 0:<number>],1:<number>]*
- TABLE(<table number>, <table identifier>, <number>, <number>, <number>);
- real *(<identifier>)*
- real array *(<identifier>)*

II. Expressions

SOL
- time
  - (e_1, e_2, ..., e_n)
  - e_1: e_2
  - normal(<expression>, <expression>)
  - exponential(<expression>)
  - poisson(<expression>)
  - geometric(<expression>)
- random

SOL-20
- TIME
  - RAND(<number>, <array identifier>);
  - UNIFORM(<number>, <number>);
  - NORMAL(<expression>, <expression>);
  - EXPONENTIAL(<expression>);
  - POISSON(<expression>);
  - GEOMETRIC(<expression>);
  - RANDOM(<expression>, <expression>, 0);

III. Relations

SOL
- (facility name)busy
- (facility name)not busy
- (store name)full
- (store name)not full
- (store name)empty
- (store name)not empty
- pr(<expression>)

SOL-20
- BUSY(<facility name>);
- FULL(<store name>);
- EMPTY(<store name>);
- PR(<expression>);
Appendix I

IV. Statements

**SOL**

```plaintext
process<identifier>; begin
  <process declaration list>
end;

new transaction to label
  cancel;
  wait<expression>
  wait until<relation>
  enter<store name>
  leave<store name>
  seize<facility name>
  release<facility name>
  go to (*<label>),<expression>
  tabulate<expression> in <table name>
output statement
stop
read<constant> to <label>
write<constant>
statistical output(automatic)
```

**SOL-20**

```plaintext
switch PROCESSES := *)process
  Identifier*;
variables PROC.PRIO [j:<number] :
  integer array;
proc<identifier>; begin
  <process declaration list>
end;

NEW.TRANSACTION(<label>);
  CANCEL;
  WAIT(<expression>);
  WAIT.UNTIL(<relation>)(parameterless boolean procedure);
  ENTER(<store name>,<expression>);
  LEAVE(<store name>,<expression>);
  SEIZE(<facility name>,<expression>);
  RELEASE(<facility name>);
  use a switch statement
  TABULATE(<expression>,<number>, <table identifier>);
  NAME and PRINT statements
  STOP;
  READIN(<procedure identifier>, <label>);
  NAME and PRINT statements
  PRT.FAC(<number>,<facility identifier>,<string>);
  PRT.ST(<number>,<store identifier>,<string>);
  PRT.TAB(<number>,<table identifier>,<string>);
```
BEGIN
REAL LINE, COMPUTER; | FACILITIES
SWITCH PROCESSES := CONTROL, USERS, PBUS, OTHER, PBUS; | DECLARE PROCESSES
INTEGER ARRAY PROC.PRI[1:4]; | PRIORITY TABLE FOR PROCESSES

FOR I = 1 STEP 1 UNTIL 6 DO BEGIN
STDRE(TU[I].SUB[0:3]); | DECLARE STORES
TABLE(I. TA3[0:6], TAB[0:2]); | TABLES

LABEL START: SCAN, COMPUTATION, COMPUTER;
DECLARE TRANSACTIONS
RUN.ERROR(RESULTS.TIMR.);
FOR I = 1 STEP 1 UNTIL 6 DO
PROC.PRI[I] := 0; | SET PRIORITIES FOR PROCESSES
START.SIMULATION;
CONTROL: BEGIN
INTEGER I;
DUMMY DECLARATION TO MAKE PROCESS A BLOCK
WAIT(15*60*1000); STOP; | STOP SIMULATION
END;

USERS: BEGIN
INTEGER Q = START.TIME, MESSAGE.TYPE;
HALF ARRAY TYPE[1:10];
FOR I = 3 STEP 1 UNTIL 7 DO TYPE[I] := 2;
NEW.TRANSACTION(START); NEW.TRANSACTION(START); ORIGIN; NEW.TRANSACTION(START);
GO TO ORIGIN;
START: Q = UNIFORM(1, 6); ENTER(TU[Q].Q1);
MESSAGE.TYPE := RAND(10, TYPE);
SEIZE(TU[Q].QO);
TMESSAGE[Q] := MESSAGE.TYPE;
WAIT(UNIFORM(6000, 8000));
START.TIME := TIME;
NAME(Q, MESSAGE.TYPE, TIME);
PRINT(108, TU[Q].Q1, 2D, 10, 'SEND MESSAGE', 2D, 10, 'AT TIME', 10, 7D, 'END');
ABULATE((TIME - START.TIME), Q, TAB);
NAME(Q, TIME);

BEGIN

INTEGER S,T,WORDS;
NEW TRANSACTION(SCAN); T = 3;
SCAN: T = T + 1; IF T > 6 THEN T = 1; WAIT(1);
S = SBSTRACT[T];
SEIZE(LINE,0);
WAIT(S); IF BUSY(SB[S]) THEN BEGIN
WAIT(80); RELEASE(LINE); GO TO SCAN END;
SEIZE(SB[S],0); WAIT(15); IF TUSTATE[T] = 1 THEN BEGIN
WAIT(55); RELEASE(LINE); RELEASE(SB[S]);
GO TO SCAN END;
WAIT(225);
SEND: WAIT(170); IF PR(0.02) THEN BEGIN
WAIT(20); GO TO SEND END;
NEW TRANSACTION(COMPUTATION); WAIT(20); RELEASE(SB[S]);
RELEASE(LINE); TUSTATE[T] = 2;
CANCEL;

COMPUTATION: SEIZE(COMPUTER,0); WORDS = TUNMESSAGE[T] + 2;
WAIT(IF WORDS = 3 THEN 250 ELSE IF WORDS = 4 THEN 300 ELSE 400);
RELEASE(COMPUTER);
OUTPT: WAIT(1); SEIZE(LINE,0); WAIT(S);
IF BUSY(SB[S]) THEN BEGIN
WAIT(80); RELEASE(LINE); GO TO OUTPT END;
SEIZE(SB[S],0); WAIT(75); RECEIVE: WAIT(80); IF PR(0.01) THEN BEGIN
WAIT(20); GO TO RECEIVE END;
RELEASE(LINE);
WORDS = WORDS - 1;
IF WORDS = 0 THEN NEW TRANSACTION(SCAN); WAIT(325); RELEASE(SB[S]); WAIT(170);
IF WORDS > 0 THEN GO TO OUTPT;
TUSTATE[T] = 0;
CANCEL;
END;

OTHER PBUS: BEGIN
INTEGER I; HALF ARRAY WAITS[1:10];
FOR I = 3 STEP 1 UNTIL 7 DO WAIT[I] = 300;
I = 6;
CREATE: NEW TRANSACTION(COMPUTE);
I = I - 1; IF I > 0 THEN GO TO CREATE; CANCEL;
COMPUTE: WAIT(UNIFORM(3200,5000)); SEIZE(COMPUTER,0);
WAIT(RANDOM(10,WAITS));
RELEASE(COMPUTER); GO TO COMPUTE;
RELEASE(COMPUTER); GO TO COMPUTE;
END;

RESULTS:
CO PRINT FRACTION OF TIME USED BY FACILITIES
FOR I = 1 STEP 1 UNTIL 6 DO
Appendix II

PRT.FAC(1, TU[I]* TU');
FOR I + 1 STEP 1 UNTIL 3 DO
  PRT.FAC(1, SB[I]* SB');
  PRT.FAC(0, LINE*'LINE');
  PRT.FAC(0, COMPUTER*COMPI);
END;

CO PRINT INFORMATION ON STORES
FOR I + 1 STEP 1 UNTIL 6 DO
  PRT.ST(1, QUEUE[I]* QUEUE');
END;

CO PRINT STATISTICS ON TABLES
FOR I + 1 STEP 1 UNTIL 6 DO
  PRT.TAB(1, TAB* TAB');
END;
TU 2 SEND MESSAGE 1 AT TIME 7752
TJ 4 SEND MESSAGE 2 AT TIME 7784
TU 3 SEND MESSAGE 2 AT TIME 10302
TU 2 RECEIVES REPLY AT TIME 10592
TJ 5 SEND MESSAGE 2 AT TIME 14052
TU 3 RECEIVES REPLY AT TIME 14118
TU 4 RECEIVES REPLY AT TIME 15442
TU 1 RECEIVES REPLY AT TIME 1844
TJ 4 SEND MESSAGE 2 AT TIME 19792
TU 5 SEND MESSAGE 3 AT TIME 23988
TU 5 RECEIVES REPLY AT TIME 23999
TU 4 RECEIVES REPLY AT TIME 2570
TJ 1 SEND MESSAGE 2 AT TIME 25935
TU 2 SEND MESSAGE 1 AT TIME 26679
TU 6 RECEIVES REPLY AT TIME 2800
TU 1 RECEIVES REPLY AT TIME 3030
TU 2 RECEIVES REPLY AT TIME 3059
TJ 3 SEND MESSAGE 3 AT TIME 31488
TU 4 SEND MESSAGE 3 AT TIME 31842
TJ 5 SEND MESSAGE 3 AT TIME 32952
TU 3 RECEIVES REPLY AT TIME 3514
TU 4 RECEIVES REPLY AT TIME 3650
TJ 1 SEND MESSAGE 2 AT TIME 36767
TU 5 RECEIVES REPLY AT TIME 3946
TJ 2 SEND MESSAGE 2 AT TIME 41957
TJ 3 SEND MESSAGE 1 AT TIME 42977
TJ 4 SEND MESSAGE 3 AT TIME 43320
TU 1 RECEIVES REPLY AT TIME 4419
TU 2 RECEIVES REPLY AT TIME 4644
TU 5 SEND MESSAGE 3 AT TIME 46994
TU 3 RECEIVES REPLY AT TIME 4812
TU 4 RECEIVES REPLY AT TIME 4942
TJ 6 SEND MESSAGE 3 AT TIME 49811
TJ 1 SEND MESSAGE 2 AT TIME 52131
TU 5 RECEIVES REPLY AT TIME 5385
TU 6 RECEIVES REPLY AT TIME 5401
TJ 2 SEND MESSAGE 2 AT TIME 54502
TJ 3 SEND MESSAGE 1 AT TIME 54982
TJ 4 SEND MESSAGE 3 AT TIME 55320
TU 1 RECEIVES REPLY AT TIME 5719
TU 2 RECEIVES REPLY AT TIME 5813
TJ 5 SEND MESSAGE 3 AT TIME 5033
TU 3 RECEIVES REPLY AT TIME 6044
TU 4 RECEIVES REPLY AT TIME 6068
TU 5 RECEIVES REPLY AT TIME 6579
TU 6 SEND MESSAGE 1 AT TIME 55013
TU 2 SEND MESSAGE 2 AT TIME 66100
TU 3 SEND MESSAGE 3 AT TIME 67319
TJ 4 SEND MESSAGE 2 AT TIME 68017
TU 6 RECEIVES REPLY AT TIME 6944
TU 2 RECEIVES REPLY AT TIME 7029
TU 5 SEND MESSAGE 3 AT TIME 72209
TU 3 RECEIVES REPLY AT TIME 7315
<table>
<thead>
<tr>
<th>NAME OF FACILITY</th>
<th>FRACTION OF TIME IN USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>TUI(01)</td>
<td>0.6645</td>
</tr>
<tr>
<td>TUI(02)</td>
<td>0.7002</td>
</tr>
<tr>
<td>TUI(03)</td>
<td>0.8633</td>
</tr>
<tr>
<td>TUI(04)</td>
<td>0.8137</td>
</tr>
<tr>
<td>TUI(05)</td>
<td>0.7230</td>
</tr>
<tr>
<td>TUI(06)</td>
<td>0.8529</td>
</tr>
<tr>
<td>SB(01)</td>
<td>0.5843</td>
</tr>
<tr>
<td>SB(02)</td>
<td>0.4566</td>
</tr>
<tr>
<td>SB(03)</td>
<td>0.2327</td>
</tr>
<tr>
<td>LINE</td>
<td>0.8761</td>
</tr>
<tr>
<td>COMP</td>
<td>0.5168</td>
</tr>
<tr>
<td>NAME OF STORE</td>
<td>CAPACITY</td>
</tr>
<tr>
<td>--------------</td>
<td>----------</td>
</tr>
<tr>
<td>QUE(u01)</td>
<td>10</td>
</tr>
<tr>
<td>QUE(u02)</td>
<td>10</td>
</tr>
<tr>
<td>QUE(u03)</td>
<td>10</td>
</tr>
<tr>
<td>QUE(u04)</td>
<td>10</td>
</tr>
<tr>
<td>QUE(u05)</td>
<td>10</td>
</tr>
<tr>
<td>QUE(u06)</td>
<td>10</td>
</tr>
<tr>
<td>UPPER LIMIT</td>
<td>NUMBER</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td>2000.0</td>
<td>0</td>
</tr>
<tr>
<td>2500.0</td>
<td>0</td>
</tr>
<tr>
<td>3000.0</td>
<td>0</td>
</tr>
<tr>
<td>3500.0</td>
<td>4</td>
</tr>
<tr>
<td>4000.0</td>
<td>8</td>
</tr>
<tr>
<td>4500.0</td>
<td>10</td>
</tr>
<tr>
<td>5000.0</td>
<td>6</td>
</tr>
<tr>
<td>5500.0</td>
<td>7</td>
</tr>
<tr>
<td>6000.0</td>
<td>1</td>
</tr>
<tr>
<td>6500.0</td>
<td>1</td>
</tr>
<tr>
<td>7000.0</td>
<td>2</td>
</tr>
<tr>
<td>7500.0</td>
<td>4</td>
</tr>
<tr>
<td>8000.0</td>
<td>1</td>
</tr>
<tr>
<td>8500.0</td>
<td>1</td>
</tr>
<tr>
<td>9000.0</td>
<td>1</td>
</tr>
<tr>
<td>9500.0</td>
<td>1</td>
</tr>
<tr>
<td>10000.0</td>
<td>0</td>
</tr>
<tr>
<td>10500.0</td>
<td>0</td>
</tr>
<tr>
<td>11000.0</td>
<td>0</td>
</tr>
<tr>
<td>11500.0</td>
<td>0</td>
</tr>
<tr>
<td>12000.0</td>
<td>0</td>
</tr>
<tr>
<td>12500.0</td>
<td>0</td>
</tr>
<tr>
<td>13000.0</td>
<td>0</td>
</tr>
<tr>
<td>13500.0</td>
<td>0</td>
</tr>
<tr>
<td>14000.0</td>
<td>0</td>
</tr>
<tr>
<td>14500.0</td>
<td>0</td>
</tr>
<tr>
<td>15000.0</td>
<td>0</td>
</tr>
<tr>
<td>UPPER LIMIT</td>
<td>NUMBER</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td>2000.0</td>
<td>0</td>
</tr>
<tr>
<td>2500.0</td>
<td>0</td>
</tr>
<tr>
<td>3000.0</td>
<td>3</td>
</tr>
<tr>
<td>3500.0</td>
<td>5</td>
</tr>
<tr>
<td>4000.0</td>
<td>16</td>
</tr>
<tr>
<td>4500.0</td>
<td>12</td>
</tr>
<tr>
<td>5000.0</td>
<td>8</td>
</tr>
<tr>
<td>5500.0</td>
<td>6</td>
</tr>
<tr>
<td>6000.0</td>
<td>4</td>
</tr>
<tr>
<td>6500.0</td>
<td>3</td>
</tr>
<tr>
<td>7000.0</td>
<td>2</td>
</tr>
<tr>
<td>7500.0</td>
<td>2</td>
</tr>
<tr>
<td>8000.0</td>
<td>4</td>
</tr>
<tr>
<td>8500.0</td>
<td>0</td>
</tr>
<tr>
<td>9000.0</td>
<td>2</td>
</tr>
<tr>
<td>9500.0</td>
<td>0</td>
</tr>
<tr>
<td>10000.0</td>
<td>0</td>
</tr>
<tr>
<td>10500.0</td>
<td>0</td>
</tr>
<tr>
<td>11000.0</td>
<td>0</td>
</tr>
<tr>
<td>11500.0</td>
<td>0</td>
</tr>
<tr>
<td>12000.0</td>
<td>0</td>
</tr>
<tr>
<td>12500.0</td>
<td>0</td>
</tr>
<tr>
<td>13000.0</td>
<td>0</td>
</tr>
<tr>
<td>13500.0</td>
<td>0</td>
</tr>
<tr>
<td>14000.0</td>
<td>0</td>
</tr>
<tr>
<td>14500.0</td>
<td>0</td>
</tr>
<tr>
<td>15000.0</td>
<td>0</td>
</tr>
</tbody>
</table>

Mean of Table: 4640.8500

Standard Deviation: 1365.2080

Sum of all entry values: 278651.0000 x 10^6

Number of Table Entries: 60

Table Name is TAB10021

Appendix II
<table>
<thead>
<tr>
<th>UPPER LIMIT</th>
<th>NUMBER</th>
<th>PER CENT</th>
<th>CUMULATIVE</th>
<th>MULTIPLE OF MEAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000.0</td>
<td>0</td>
<td>0.00</td>
<td>0.00</td>
<td>0.4021</td>
</tr>
<tr>
<td>2500.0</td>
<td>0</td>
<td>0.00</td>
<td>0.00</td>
<td>0.5026</td>
</tr>
<tr>
<td>3000.0</td>
<td>1</td>
<td>1.54</td>
<td>1.54</td>
<td>0.6031</td>
</tr>
<tr>
<td>3500.0</td>
<td>1</td>
<td>1.54</td>
<td>3.08</td>
<td>0.7042</td>
</tr>
<tr>
<td>4000.0</td>
<td>13</td>
<td>20.00</td>
<td>23.08</td>
<td>0.8044</td>
</tr>
<tr>
<td>4500.0</td>
<td>15</td>
<td>23.00</td>
<td>46.15</td>
<td>0.9047</td>
</tr>
<tr>
<td>5000.0</td>
<td>9</td>
<td>13.85</td>
<td>60.00</td>
<td>1.0052</td>
</tr>
<tr>
<td>5500.0</td>
<td>9</td>
<td>13.85</td>
<td>73.85</td>
<td>1.1057</td>
</tr>
<tr>
<td>6000.0</td>
<td>9</td>
<td>7.69</td>
<td>91.54</td>
<td>1.2063</td>
</tr>
<tr>
<td>6500.0</td>
<td>3</td>
<td>4.62</td>
<td>96.15</td>
<td>1.3068</td>
</tr>
<tr>
<td>7000.0</td>
<td>3</td>
<td>4.62</td>
<td>98.77</td>
<td>1.4073</td>
</tr>
<tr>
<td>7500.0</td>
<td>3</td>
<td>4.62</td>
<td>95.38</td>
<td>1.5078</td>
</tr>
<tr>
<td>8000.0</td>
<td>1</td>
<td>1.54</td>
<td>96.92</td>
<td>1.6084</td>
</tr>
<tr>
<td>8500.0</td>
<td>0</td>
<td>0.00</td>
<td>96.92</td>
<td>1.7089</td>
</tr>
<tr>
<td>9000.0</td>
<td>1</td>
<td>1.54</td>
<td>98.46</td>
<td>1.8094</td>
</tr>
<tr>
<td>9500.0</td>
<td>1</td>
<td>1.54</td>
<td>100.00</td>
<td>1.9099</td>
</tr>
<tr>
<td>10000.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.0104</td>
</tr>
<tr>
<td>10500.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.1110</td>
</tr>
<tr>
<td>11000.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.2115</td>
</tr>
<tr>
<td>11500.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.3120</td>
</tr>
<tr>
<td>12000.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.4129</td>
</tr>
<tr>
<td>12500.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.5131</td>
</tr>
<tr>
<td>13000.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.6133</td>
</tr>
<tr>
<td>13500.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.7141</td>
</tr>
<tr>
<td>14000.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.8146</td>
</tr>
<tr>
<td>14500.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.9151</td>
</tr>
<tr>
<td>15000.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>3.0157</td>
</tr>
<tr>
<td>UPPER LIMIT</td>
<td>NUMBER</td>
<td>PER CENT</td>
<td>CUMULATIVE</td>
<td>MULTIPLE OF MEAN</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
<td>----------</td>
<td>------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>2000.0</td>
<td>0</td>
<td>0.00</td>
<td>0.00</td>
<td>0.3612</td>
</tr>
<tr>
<td>2500.0</td>
<td>0</td>
<td>0.00</td>
<td>0.00</td>
<td>0.4765</td>
</tr>
<tr>
<td>3000.0</td>
<td>2</td>
<td>3.39</td>
<td>3.39</td>
<td>0.6671</td>
</tr>
<tr>
<td>3500.0</td>
<td>10</td>
<td>16.95</td>
<td>20.34</td>
<td>0.7624</td>
</tr>
<tr>
<td>4000.0</td>
<td>12</td>
<td>20.34</td>
<td>40.66</td>
<td>0.8577</td>
</tr>
<tr>
<td>4500.0</td>
<td>10</td>
<td>16.95</td>
<td>57.63</td>
<td>0.9530</td>
</tr>
<tr>
<td>5000.0</td>
<td>6</td>
<td>10.17</td>
<td>67.80</td>
<td>1.0483</td>
</tr>
<tr>
<td>5500.0</td>
<td>4</td>
<td>6.78</td>
<td>74.58</td>
<td>1.1436</td>
</tr>
<tr>
<td>6000.0</td>
<td>4</td>
<td>6.78</td>
<td>81.36</td>
<td>1.2389</td>
</tr>
<tr>
<td>6500.0</td>
<td>2</td>
<td>3.39</td>
<td>84.75</td>
<td>1.3342</td>
</tr>
<tr>
<td>7000.0</td>
<td>2</td>
<td>3.39</td>
<td>88.14</td>
<td>1.4295</td>
</tr>
<tr>
<td>7500.0</td>
<td>4</td>
<td>6.78</td>
<td>94.92</td>
<td>1.5248</td>
</tr>
<tr>
<td>8000.0</td>
<td>0</td>
<td>0.00</td>
<td>94.92</td>
<td>1.6201</td>
</tr>
<tr>
<td>8500.0</td>
<td>1</td>
<td>1.69</td>
<td>96.61</td>
<td>1.7154</td>
</tr>
<tr>
<td>9000.0</td>
<td>0</td>
<td>0.00</td>
<td>96.61</td>
<td>1.8107</td>
</tr>
<tr>
<td>9500.0</td>
<td>1</td>
<td>1.69</td>
<td>98.31</td>
<td>1.9060</td>
</tr>
<tr>
<td>10000.0</td>
<td>0</td>
<td>0.00</td>
<td>98.31</td>
<td>2.0013</td>
</tr>
<tr>
<td>10500.0</td>
<td>1</td>
<td>1.69</td>
<td>100.00</td>
<td>2.0966</td>
</tr>
<tr>
<td>11000.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.1919</td>
</tr>
<tr>
<td>11500.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.2872</td>
</tr>
<tr>
<td>12000.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.3825</td>
</tr>
<tr>
<td>12500.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.4778</td>
</tr>
<tr>
<td>13000.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.5731</td>
</tr>
<tr>
<td>13500.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.6684</td>
</tr>
<tr>
<td>14000.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.7637</td>
</tr>
<tr>
<td>14500.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.8590</td>
</tr>
<tr>
<td>15000.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.9543</td>
</tr>
<tr>
<td>UPPER LIMIT</td>
<td>NUMBER</td>
<td>PER CENT</td>
<td>CUMULATIVE</td>
<td>MULTIPLE OF MEAN</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
<td>----------</td>
<td>------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>2000.0</td>
<td>0</td>
<td>0.00</td>
<td>0.00</td>
<td>0.3898</td>
</tr>
<tr>
<td>3000.0</td>
<td>2</td>
<td>0.00</td>
<td>0.00</td>
<td>0.4873</td>
</tr>
<tr>
<td>4000.0</td>
<td>5</td>
<td>9.26</td>
<td>12.96</td>
<td>0.6822</td>
</tr>
<tr>
<td>5000.0</td>
<td>11</td>
<td>20.37</td>
<td>43.59</td>
<td>0.7792</td>
</tr>
<tr>
<td>6000.0</td>
<td>4</td>
<td>7.41</td>
<td>50.97</td>
<td>0.9746</td>
</tr>
<tr>
<td>7000.0</td>
<td>4</td>
<td>9.41</td>
<td>59.00</td>
<td>1.0721</td>
</tr>
<tr>
<td>8000.0</td>
<td>1</td>
<td>1.85</td>
<td>60.85</td>
<td>1.1695</td>
</tr>
<tr>
<td>9000.0</td>
<td>1</td>
<td>1.85</td>
<td>62.70</td>
<td>1.2670</td>
</tr>
<tr>
<td>10000.0</td>
<td>0</td>
<td>0.00</td>
<td>62.70</td>
<td>1.3645</td>
</tr>
<tr>
<td>11000.0</td>
<td>0</td>
<td>0.00</td>
<td>62.70</td>
<td>1.4610</td>
</tr>
<tr>
<td>12000.0</td>
<td>0</td>
<td>0.00</td>
<td>62.70</td>
<td>1.5575</td>
</tr>
<tr>
<td>13000.0</td>
<td>0</td>
<td>0.00</td>
<td>62.70</td>
<td>1.6540</td>
</tr>
<tr>
<td>14000.0</td>
<td>0</td>
<td>0.00</td>
<td>62.70</td>
<td>1.7505</td>
</tr>
<tr>
<td>15000.0</td>
<td>0</td>
<td>0.00</td>
<td>62.70</td>
<td>1.8470</td>
</tr>
</tbody>
</table>

**Number of Table Entries:** 54

**Sum of All Entry Values:** 2770320000

**Mean of Table:** 5130.222

**Standard Deviation:** 1558.6936
<table>
<thead>
<tr>
<th>UPPER LIMIT</th>
<th>NUMBER</th>
<th>PERCENT</th>
<th>CUMULATIVE</th>
<th>MULTIPLE OF MEAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000.0</td>
<td>0</td>
<td>0.00</td>
<td>0.00</td>
<td>0.4108</td>
</tr>
<tr>
<td>2500.0</td>
<td>0</td>
<td>0.00</td>
<td>0.00</td>
<td>0.5135</td>
</tr>
<tr>
<td>3000.0</td>
<td>0</td>
<td>0.00</td>
<td>0.00</td>
<td>0.6161</td>
</tr>
<tr>
<td>3500.0</td>
<td>3</td>
<td>4.69</td>
<td>4.69</td>
<td>0.7188</td>
</tr>
<tr>
<td>4000.0</td>
<td>10</td>
<td>15.65</td>
<td>20.31</td>
<td>0.8219</td>
</tr>
<tr>
<td>4500.0</td>
<td>25</td>
<td>39.06</td>
<td>59.38</td>
<td>0.9242</td>
</tr>
<tr>
<td>5000.0</td>
<td>2</td>
<td>3.12</td>
<td>62.50</td>
<td>1.0269</td>
</tr>
<tr>
<td>5500.0</td>
<td>11</td>
<td>17.19</td>
<td>79.69</td>
<td>1.1296</td>
</tr>
<tr>
<td>6000.0</td>
<td>5</td>
<td>7.81</td>
<td>87.50</td>
<td>1.2323</td>
</tr>
<tr>
<td>6500.0</td>
<td>1</td>
<td>1.56</td>
<td>89.06</td>
<td>1.3350</td>
</tr>
<tr>
<td>7000.0</td>
<td>1</td>
<td>1.56</td>
<td>90.62</td>
<td>1.4377</td>
</tr>
<tr>
<td>7500.0</td>
<td>1</td>
<td>1.56</td>
<td>92.19</td>
<td>1.5404</td>
</tr>
<tr>
<td>8000.0</td>
<td>2</td>
<td>3.12</td>
<td>95.31</td>
<td>1.6430</td>
</tr>
<tr>
<td>8500.0</td>
<td>2</td>
<td>3.12</td>
<td>98.44</td>
<td>1.7457</td>
</tr>
<tr>
<td>9000.0</td>
<td>0</td>
<td>0.00</td>
<td>98.44</td>
<td>1.8484</td>
</tr>
<tr>
<td>9500.0</td>
<td>0</td>
<td>0.00</td>
<td>98.44</td>
<td>1.9511</td>
</tr>
<tr>
<td>10000.0</td>
<td>1</td>
<td>1.56</td>
<td>100.00</td>
<td>2.0538</td>
</tr>
<tr>
<td>10500.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.1565</td>
</tr>
<tr>
<td>11000.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.2592</td>
</tr>
<tr>
<td>11500.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.3619</td>
</tr>
<tr>
<td>12000.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.4646</td>
</tr>
<tr>
<td>12500.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.5673</td>
</tr>
<tr>
<td>13000.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.6700</td>
</tr>
<tr>
<td>13500.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.7725</td>
</tr>
<tr>
<td>14000.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.8753</td>
</tr>
<tr>
<td>14500.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>2.9780</td>
</tr>
<tr>
<td>15000.0</td>
<td>0</td>
<td>0.00</td>
<td>100.00</td>
<td>3.0807</td>
</tr>
</tbody>
</table>
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 pseudocode, 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.

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-
IEEE TRANSACTIONS ON ELECTRONIC COMPUTERS

Appendix III

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 program that would be necessary 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 central 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.

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 six 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], ..., TU[6], TU[1], ... 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), if the appropriate queue until the typewriter is free.

These people will send one of three kinds of messages.

<table>
<thead>
<tr>
<th>Message</th>
<th>Frequency</th>
<th>Compute Time</th>
<th>Number of Response Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>40 per cent</td>
<td>250 m sec</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>30 per cent</td>
<td>300 m sec</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>30 per cent</td>
<td>400 m sec</td>
<td>5</td>
</tr>
</tbody>
</table>

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 six 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], ..., TU[6], TU[1], ... 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.

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 [2]-[4].) As simulation begins, there are 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 transaction (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 only to pass before executing the next statement. The expression E; E always denotes a random integer chosen between E1 and E2, and therefore "wait 0:5000" has the meaning stated above. A unit of time in this case represents 1 m sec for 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-
begin
facility tu[6], sb[3], line, computer;
store 10 queue[6];
integer tustate[6], snumber[6], tumessage[6];
table (2000 step 500 until 15000) table[6];
process master control;
begin snumber[1] = 1; snumber[2] = 2;
wait 60 X60 X1000; 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, 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 pro; begin integer s, t, words;
new transaction to scan; t = 3;
scan: t = t + 1; if t > 6 then t = 1; wait 1;
s = snumber[t];
seize line;
wait 5; if sb[s] busy then (wait 80; release line; go to
scan);
seize sb[s]; wait 15; if tustate[r] #= 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[r] = 2; cancel;
computation: seize computer; words = tumessage[r]
+ 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[r] = 0; cancel end;
process other pro;
begin integer i = 1:6;
create: new transaction to compute;
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.
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**

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<—(1,1,2,2,2,2,2,3,3,3,3);**

The expression 

\[(E_1, E_2, \ldots, E_n)\]

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.

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;**

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;**
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 sn[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 sn[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 sn[s] received the control signal, it is brought under the control of this PBU. Fifteen milliseconds later, the number s is 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 sn[s];
    release line; tustate[t] = 1; 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[r] + 2;
    wait (if words = 3 then 250 else
    if words = 4 then 300 else 500);
    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 sn[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 sn[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 PBUs as follows:

    process other pbus;
    begin integer i; i = 6;
    create: new transaction to compute;
    t = t + 1; if t > 0 then go to create; cancel;
    compute: wait 3200; seize computer;
    wait (250, 250, 300, 300, 300, 300, 400, 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], sn[3], line, computer;
    store 10 queue[6];
    integer tustate[6], snnumber[6], tumessage[6];
    table (2000 step 500 until 15000) table[6];
    process master control;
    begin snnumber[1] = 1; snnumber[2] = 2;
    wait 60x60x1000; 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.
CLOCK TIME AT END OF SIMULATION WAS 3600000
NUMBER OF TIMES LABELS WERE ENCOUNTERED

<table>
<thead>
<tr>
<th>LABEL</th>
<th>COUNT</th>
<th>LABEL</th>
<th>COUNT</th>
<th>LABEL</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORIGIN</td>
<td>1655</td>
<td>START</td>
<td>1657</td>
<td>SCAN</td>
<td>1730</td>
</tr>
<tr>
<td>SEND</td>
<td>1477</td>
<td>COMPUTATION</td>
<td>1446</td>
<td>OUTPUT</td>
<td>2022</td>
</tr>
<tr>
<td>RECEIVE</td>
<td>5990</td>
<td>CREATE</td>
<td>6</td>
<td>COMPUTE</td>
<td>4764</td>
</tr>
</tbody>
</table>

NAME OF FACILITY      FRACTION OF TIME IN USE

<p>| TUC001 | 0.6316 |
| TUC002 | 0.6056 |
| TUC003 | 0.6497 |
| TUC004 | 0.6096 |
| TUC005 | 0.6302 |
| SBC001 | 0.5505 |
| SBC002 | 0.6091 |
| SBC003 | 0.4221 |
| LINE   | 0.3120 |
| &quot;COMPUTER&quot; | 0.5509 |</p>
<table>
<thead>
<tr>
<th>NAME OF Store</th>
<th>CAPACITY</th>
<th>MAXIMUM USED</th>
<th>AVERAGE OCCUPANCY</th>
<th>AVERAGE UTILIZATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUEUE(001)</td>
<td>10</td>
<td>10</td>
<td>2.5272</td>
<td>0.2527</td>
</tr>
<tr>
<td>QUEUE(002)</td>
<td>10</td>
<td>10</td>
<td>2.4255</td>
<td>0.2926</td>
</tr>
<tr>
<td>QUEUE(003)</td>
<td>10</td>
<td>10</td>
<td>2.3936</td>
<td>0.2306</td>
</tr>
<tr>
<td>QUEUE(004)</td>
<td>10</td>
<td>7</td>
<td>1.7696</td>
<td>0.1770</td>
</tr>
<tr>
<td>QUEUE(005)</td>
<td>10</td>
<td>8</td>
<td>2.1844</td>
<td>0.2184</td>
</tr>
<tr>
<td>QUEUE(006)</td>
<td>10</td>
<td>5</td>
<td>1.4971</td>
<td>0.1497</td>
</tr>
</tbody>
</table>

**Table Name IS TABLE(003)**

**NUMBER OF TABLE ENTRIES:** 235

**SUM OF ALL ENTRY VALUES:** 1201078

<table>
<thead>
<tr>
<th>MEAN OF TABLE</th>
<th>910.9617</th>
</tr>
</thead>
</table>

**NUMBER OF TABLE ENTRIES**

<table>
<thead>
<tr>
<th>NUMBER</th>
<th>PER CENT</th>
<th>CUMULATIVE</th>
<th>MULTIPLE OF MEAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>0</td>
<td>0.00</td>
<td>0.3913</td>
</tr>
<tr>
<td>2500</td>
<td>0</td>
<td>0.00</td>
<td>0.4991</td>
</tr>
<tr>
<td>3000</td>
<td>5</td>
<td>2.13</td>
<td>0.5670</td>
</tr>
<tr>
<td>3500</td>
<td>14</td>
<td>5.99</td>
<td>0.6868</td>
</tr>
<tr>
<td>4000</td>
<td>36</td>
<td>15.32</td>
<td>0.7826</td>
</tr>
<tr>
<td>4500</td>
<td>32</td>
<td>13.60</td>
<td>0.8855</td>
</tr>
<tr>
<td>5000</td>
<td>41</td>
<td>19.59</td>
<td>0.7830</td>
</tr>
<tr>
<td>5500</td>
<td>23</td>
<td>9.79</td>
<td>1.0761</td>
</tr>
<tr>
<td>6000</td>
<td>25</td>
<td>10.64</td>
<td>1.1739</td>
</tr>
<tr>
<td>6500</td>
<td>18</td>
<td>7.66</td>
<td>1.2718</td>
</tr>
<tr>
<td>7000</td>
<td>12</td>
<td>5.11</td>
<td>1.3676</td>
</tr>
<tr>
<td>7500</td>
<td>10</td>
<td>4.26</td>
<td>1.4674</td>
</tr>
<tr>
<td>8000</td>
<td>5</td>
<td>2.13</td>
<td>1.5693</td>
</tr>
<tr>
<td>8500</td>
<td>1</td>
<td>0.43</td>
<td>1.6631</td>
</tr>
<tr>
<td>9000</td>
<td>4</td>
<td>1.70</td>
<td>1.7609</td>
</tr>
<tr>
<td>9500</td>
<td>1</td>
<td>0.43</td>
<td>1.8587</td>
</tr>
<tr>
<td>10000</td>
<td>1</td>
<td>0.43</td>
<td>1.9566</td>
</tr>
<tr>
<td>10500</td>
<td>1</td>
<td>0.43</td>
<td>2.0544</td>
</tr>
<tr>
<td>11000</td>
<td>0</td>
<td>0.00</td>
<td>2.1522</td>
</tr>
<tr>
<td>11500</td>
<td>1</td>
<td>0.43</td>
<td>2.2501</td>
</tr>
<tr>
<td>12000</td>
<td>0</td>
<td>0.00</td>
<td>2.3479</td>
</tr>
<tr>
<td>12500</td>
<td>0</td>
<td>0.00</td>
<td>2.4457</td>
</tr>
<tr>
<td>13000</td>
<td>0</td>
<td>0.00</td>
<td>2.5436</td>
</tr>
<tr>
<td>13500</td>
<td>0</td>
<td>0.00</td>
<td>2.6414</td>
</tr>
<tr>
<td>14000</td>
<td>0</td>
<td>0.00</td>
<td>2.7392</td>
</tr>
<tr>
<td>14500</td>
<td>0</td>
<td>0.00</td>
<td>2.8370</td>
</tr>
<tr>
<td>15000</td>
<td>0</td>
<td>0.00</td>
<td>2.9349</td>
</tr>
</tbody>
</table>

Fig. 3 (pp. 406-407) — Samples of the output obtained.
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
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 used 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 initiated within one process may not refer to the local variables of another process or 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.

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 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 statements." 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.
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
(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)
(variable declaration):: = integer *(declared item)*
(real* (declared item)*
(facility declaration):: = facility *(declared item)*
(store declaration):: = store *(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, store 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

(terml):: = (primary) (primary) (term) + (primary) (term) // (primary) (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 "mod b," the positive remainder obtained upon dividing a by b; "max(a1, • • • , an)" and "min(a1, • • • , an)," which denote the maximum and minimum values, respectively, of the a expressions; and there are also notations for expressing random values. The expression "(ei, • • • , en)" 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 poison 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 $e_{\text{int}}$ which denotes a random integer between the limits $c_1$ and $c_2$.

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

$$\text{normal}(M, S) = S \times \sqrt{-2 \ln \text{random}} \times \sin(2 \pi \text{random}) + M$$
$$\text{exponential}(M) = -M \ln \text{random}$$
$$\text{poisson}(M) = n \text{ if } e^{-M} \left(1 + M + \frac{M^2}{2!} + \cdots + \frac{M^n}{n!}\right) \leq \text{random} < e^{-M} \left(1 + M + \cdots + \frac{M^n}{n!}\right)$$
$$\text{geometric}(M) = \left[1 + \ln \left(\text{random}/M\right)/\ln \left(1 - \frac{1}{M}\right)\right]$$

(The poison 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 poison(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; initially, time is zero. The value of a store name is the current number of occupants of the store.

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.

B. Labels
A statement may be named by any identifier as follows:

<table>
<thead>
<tr>
<th>relational operator</th>
<th>(relational operator)(unconditional expression)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(faciilty name) busy</td>
<td>(facility name) not busy</td>
</tr>
<tr>
<td>(store name) full</td>
<td>(store name) not full</td>
</tr>
<tr>
<td>pr(express)</td>
<td>(relational)</td>
</tr>
<tr>
<td>(relation): 1 = (relation)</td>
<td></td>
</tr>
<tr>
<td>(relation primary)</td>
<td>(relation primary)</td>
</tr>
<tr>
<td>(relation primary) \text{V} (relation primary)</td>
<td></td>
</tr>
<tr>
<td>(relation primary)</td>
<td>(relation primary)</td>
</tr>
<tr>
<td>(relation primary)</td>
<td></td>
</tr>
</tbody>
</table>

These relations have obvious meanings except for the construction "pr(0)", which stands for a random condition which is true with probability 0. (Here 0 must be less than or equal to 1.) Thus we might say

$$\text{if } pr(0.12) \text{ then (12 per cent of the time)} \text{ else (88 per cent of the time)}.$$
Appendix III  
IEEE TRANSACTIONS ON ELECTRONIC COMPUTERS

August

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-I-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) | (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) | (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) | (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 $E_b$ be the control strength with which the facility was seized and let $E_r$ be the control strength of this seize statement. If $E_r \geq E_b$, the transaction waits until the facility is not busy. If $E_r > E_b$, however, interrupt occurs. The transaction $T_i$ which had control of
the facility is stopped wherever it was in its program, and the present transaction \( T_i \) seizes the facility. When \( T_i \) releases the facility, the following occurs:

1) If \( T_i \) 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 \( E_u \), then \( T_i \) 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

\[
\text{(release statement)} ::= \text{release (facility name)}
\]

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 \( \text{PRIORITY} \),
3) first to request the facility.

M. Go To Statements

\[
\text{(go to statement)} ::= \text{go to (label)} \mid \text{go to ([label]), (expression)}
\]

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:

\[
\text{(statement list)} ::= \text{(statement)} \mid \text{(statement list); (statement)}
\]

\[
\text{(compound statement)} ::= \text{begin (statement list) end)
\]

\[
\text{((statement list))}
\]

O. Conditional Statements

\[
\text{(conditional)} ::= \text{if (relation) then (unconditional statement)}
\]

\[
\text{if (relation) then (unconditional statement) else (statement)}
\]

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

P. Tabulate Statements

\[
\text{(tabulate statement)} ::= \text{tabulate (expression) in (table name)}
\]

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

Q. Output Statements

\[
\text{(carriage control)} ::= (\text{empty}) | \text{page} | \text{line} | \text{double (string)} \mid (\text{any sequence of characters excluding "$"}) \mid (\text{output list item}) \mid \text{#}\text{string (expression)}} \mid \text{(store name) | (table name) | (facility name)}
\]

\[
\text{(output statement)} ::= \text{output *(carriage control) (output list item)*}
\]

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

\[
\text{(stop statement)} ::= \text{stop}
\]

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

\[
\text{(procedure declaration)} ::= \text{procedure (identifier); (statement)}
\]

\[
\text{(procedure statement)} ::= (\text{procedure name)}
\]

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

\[
\text{(transaction read statement)} ::= \text{read (constant) to (label)}
\]

\[
\text{(transaction write statement)} ::= \text{write (constant)}
\]

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-
curing 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 description) |
  (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 correspondence 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

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:

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
The following identifiers are now reserved:
B,LINK1, B,LINK2, COLON, HH2, T,CNT, UMARK, X1, X2

Page 1.1
line 10:
   
   real array *<identifier>[1:2]

line 23:
   
   UNIFORM....; COLON(<number>, <number>);

Page 1.2
line 4:
   
   switch <switch identifier> := *<process

line 6:
   
   integer array <array identifier>[1:<number>];

line -10:
   
   STOP(<label>);

add: procedure <identifier> Same as in Algol. For SOL-procedure restric-
   <procedure statement> tions, 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
\[ L = \text{length of time for which the store remained at its current value.} \]
\[ U = \text{total unit-occupancy} = \sum L \cdot (\text{storage contents}) \]
Then average occupancy = \( \frac{U}{\text{total time of run}} \)

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];