SOL

Paul Bryant

November 1966

ACKNOWLEDGEMENTS

The author wishes to thank the Atlas Computer Laboratory staff for their help and advice during the implementation of this compiler. Also grateful thanks are due to J. McNeley for kind permission to reproduce his original papers on SOL, and to the Institute of Electrical and Electronics Engineers in whose transactions these papers first appeared.

Foreword

This is a provisional manual. Dr. Bryant has written a compiler for the simulation language SOL and this language is now available to any user of the Atias Laboratory. We believe that the compiler is free from, at any rate serious, errors and we would not have undertaken this task if we did not believe that the language is powerful and flexible and provides a unified method for attacking complicated problems of the operatlonal research type. But we need praotical experience and have therefore decided to put it on field trial as quickly aa possible. Dr. Bryant will be glad to help anyone who has difficulty in interpreting these notes or in using the compiler, and.will be glad also to know of errors and to receive suggestions for changes or additions and for the form and content of a final version of the manual.

J Howlett, Director

Atlas Computer Laboratory

15th December 1966

CONTENTS

CHAPTER I

Introduction

The simulation of complex systems on digital computers is a powerful tool in the design of such systems. Examples of the types of problems amenable to this form of solution are the flow of traffic through a road network, the flow of material through a factory, or the most economical way of deploying a fleet of ships. Simulation models have the advantage over the live system that the parameters associated.with a given system may be varied at will, cheaply and easily and the resulting effects can be computed and the inferences drawn without the vast cost of making the changes to the actual system.

The earliest simulation problems were tackled with hand coded programs and were one off jobs tailored to a particular need but with the development of large and powerful computers the essentials of simulation problems were extracted and put into the framework of compilers which were then capable of tackling large problem areas.

One such compiler is SOL (Simulation Orientated Language) designed by D. E. Knuth and J. L. McNeley. The author first met this language at Carnegie Institute of Technology where it was being used extensively for the simulation of multi-access computing systems. There is no better introduction to the language than the original papers which are reproduced in Chapters II and III. Chapter II gives an easy introduction to SOL by way of a fairly complicated example and the reader is advised to fully understand this Chapter before proceeding. Chapter III gives a formal definition of the language; the Atlas implementation has adhered to this as far as possible. This Chapter may be omitted at a first reading. Chapter IV gives the exact differences between SOL as defined in Chapter III and Atlas SOL. These differences have been demanded by the card character set on Atlas and the limitations of the Atlas Compiler Compiler language by means of which Atlas SOL has been implemented. Chapter VIII gives a listing of the Sample problem of Knuth and McNeley in Atlas SOL followed by the output from the actual running on Atlas of the problem. The results differ from those of Knuth and McNeley only because a different random number generator was used.

CHAPTER II

SOL—A Symbolic Language for General-Purpose Systems Simulation

D. E. KNUTH AND J. L. McNELEY

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

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

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

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-

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

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

Example: Communication with Remote Terminals

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

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

These people will send one of three kinds of messages.

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

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

Communication between the typewriters and the processor is handled by *site buffers* SB[1], SB[2], SB[3], one at each remote site, and by two *processor buffers* PBU's, which receive the information and transmit it to the computer. These processor buffers sequentially scan TU[1], TU[2], · · · , 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.

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

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

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

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

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

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

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

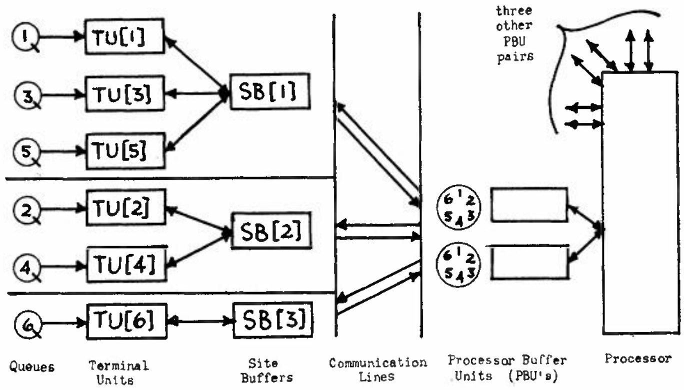

Fig. 1—Multiple console on-line communication system.

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

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

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

The program now proceeds as follows:

START: Q←1:6; **enter** QUEUE[Q];

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

MESSAGE TYPE←(1,1,2,2,2,2,2,3,3,3);

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

seize TU[Q];

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

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

TUMESSAGE[Q]←MESSAGE TYPE;

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

wait 6000:8000;

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

START TIME←**time**;

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

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

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

TUSTATE[Q]←1;

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

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

The next statement

wait until TUSTATE[Q] = 0;

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

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

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

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

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

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

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

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

SCAN: T←T+1; **if** T > 6 **then** T←1; **wait** 1;

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

S←SBNUMBER[T];

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

seize LINE;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Remarks

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

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

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

CLOCK TIME AT END OF SIMULATION WAS    3600000

NUMBER OF TIMES LABELS WERE ENCOUNTERED

LABEL        -      COUNT     LABEL          -     COUNT     LABEL        -     COUNT
     ORIGIN  -       1455            START   -      1457           SCAN   -     17303
       SEND  -       1477      COMPUTATION   -      1446         OUTPUT   -      8021
    RECEIVE  -       5990           CREATE   -         6        COMPUTE   -      4764

             NAME OF FACILITY     FRACTION OF TIME IN USE

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

```
NAME OF STORE        CAPACITY       MAXIMUM USED       AVERAGE OCCUPANCY       AVERAGE UTILIZATION

 QUEUE[001]             10              10                  2.5272                 0.2527
 QUEUE[002]             10              10                  2.4255                 0.2426
 QUEUE[003]             10              10                  2.3835                 0.2384
 QUEUE[004]             10               7                  1.7696                 0.1770
 QUEUE[005]             10               8                  2.1844                 0.2184
 QUEUE[006]             10               5                  1.4971                 0.1497

                     TABLE NAME IS   TABLE[003]

     NUMBER OF TABLE ENTRIES        235                  SUM OF ALL ENTRY VALUES          1201076

     MEAN OF TABLE          5110.9617          STANDARD DEVIATION          1441.51124

     UPPER LIMIT          NUMBER          PER CENT          CUMULATIVE          MULTIPLE OF MEAN

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

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

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

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

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

Acknowledgment

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

References

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

CHAPTER III

A Formal Definition of SOL

D. E. KNUTH AND J. L. McNELEY

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

I. General Description

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

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

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

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

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

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

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

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

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

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

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

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

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

II. Syntax and Semantics of SOL

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

⟨A⟩: : = ⟨A⟩ | *⟨A⟩*, ⟨A⟩

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

A. Identifiers and Constants

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

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

B. Declarations

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

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

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

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

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

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

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

C. Expressions and Relations

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

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

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

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

The new elements here are "a **mod** b," the positive remainder obtained upon dividing a by b; "max($e_1, \cdots, e_n$)" and "min($e_1, \cdots, e_n$)," which denote the maximum and minimum values, respectively, of the n expressions; and there are also notations for expressing random values. The expression "($e_1, \cdots, e_n$)" indicates that a random selection is made from among the n expressions with equal probability of choosing any expression. The expressions normal(M, S), poisson(M), geometric(M) and exponential(M) indicate random values with special distributions which occur frequently in applications. A random number drawn from the normal distribution with mean M and standard deviation S is denoted by normal(M, S) and is a real (not necessarily integer) value. A number drawn from the exponential distribution with mean M is denoted by exponential(M) and is also of type real. The poisson distribution signified by poisson(M), on the other hand, yields only integer values; the probability that poisson(M) = n is $(e^{-M}M^n/n!)$. The geometric distribution with mean M, denoted by geometric(M), also yields integer values, where the probability that geometric(M) = n is $1/M(1-1/M)^{n-1}$. The symbol **random** denotes a random real number between 0 and 1 having uniform distribution. Finally, we have the notation $e_1{:}e_2$, which denotes a random integer between the limits e_1 and e_2; more formally

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

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

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

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

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

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

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

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

⟨relational operator⟩: : = $=$ | $\neq$ | $<$ | $\leqq$ | $>$ | $\geqq$
⟨relation primary⟩: : = ⟨unconditional expression⟩
⟨relational operator⟩⟨unconditional expression⟩|
⟨facility name⟩ **busy** | ⟨facility name⟩ **not busy**|
⟨store name⟩ **full** | ⟨store name⟩ **not full**|
⟨store name⟩ **empty**| ⟨store name⟩ **not empty**|
pr(⟨expression⟩)| (⟨relation⟩)
⟨relation⟩: : = ⟨relation primary⟩|
⟨relation primary⟩ $\vee$ ⟨relation primary⟩|
⟨relation primary⟩ $\wedge$ ⟨relation primary⟩|
$\neg$⟨relation primary⟩

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

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

III. Statements

A. Processes

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

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

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

B. Labels

A statement may be named by any identifier as follows:

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

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

C. Creation of Transactions

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

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

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

D. Disappearance of Transactions

Transactions "die" when they execute a cancel statement.

⟨cancel statement⟩: : = **cancel**

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

E. Replacement Statements

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

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

F. Priority

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

PRIORITY←⟨expression⟩.

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

G. Wait Statements

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

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

H. Wait-Until Statements

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

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

I. Enter Statements

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

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

J. Leave Statement

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

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

K. Seize Statements

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

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

The expression appearing above represents the "control strength" which is normally zero. Allowance is made, however, for one transaction to interrupt another. If the facility is busy when the seize statement occurs, let E_1 be the control strength with which the facility was seized and let E_2 be the control strength of this seize statement. If $E_2 \leq E_1$, the transaction waits until the facility is not busy. If $E_2 > E_1$, however, *interrupt* occurs. The transaction T_1 which had control of the facility is stopped wherever it was in its program, and the present transaction T_2 seizes the facility. When T_2 releases the facility, the following occurs:

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

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

L. Release Statements

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

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

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

M. Go To Statements

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

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

N. Compound Statements

Several statements may be combined into one, as follows:

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

O. Conditional Statements

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

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

P. Tabulate Statements

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

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

Q. Output Statements

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

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

R. Stop Statements

⟨stop statement⟩: : = **stop**

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

S. Procedures

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

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

T. Transaction Input-Output

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

The read statement inputs a set of values of local variables for a transaction of the same type as the one executing the read statement; this set of values is used in the creation of a new transaction which begins executing the program at the statement mentioned. The write statement writes the current values of the local variables of the transaction onto the unit specified and does not cancel the present transaction. The constant in each refers to a tape or card unit number. The same tape should not be used for both input and output in the same simulation run.

U. Summary of Statements

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

IV. The Model as a Whole

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

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

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

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

References

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

CHAPTER IV

Differences in Atlas SOL

A. Some character and phrase substitutions have been made to suit the Atlas character set. These are as follows:

```
          SOL              Atlas SOL
 (1)       [                  (
 (2)       ]                  )
 (3)       ;               New line or new card
 (4)       ×                  *
 (5)      <-                  ≈
 (6)       =                 .EQ.
 (7)       ≠                 .NE.
 (8)       <                .LT.
 (9)       ≤                 .LE.
(10)       >                 .GT.
(11)       ≥                 .GE.
(12)       :             ) after a label
(13)       :             : in <sim> : <sum>
(14)       #                  |
(15)      busy               .BUSY
(16)      not busy           .NOT BUSY
(17)      full               .FULL
(18)      not full           .NOT FULL
(19)      empty              .EMPTY
(20)      not empty          .NOT EMPTY
```

(21) Spaces are ignored everywhere including text to be output. In the case of text to be output * is interpreted as space. For example

```
OUTPUT A,'**THIS*IS*TEXT**',B
```

will be printed

```
54  THIS IS TEXT     129
```

All letters are upper case. It is therefore unwise to use syntactically meaningful letter combinations as identifiers or as the first characters of an identifier. WAIT or OUTPUTABC would not be wise choices for identifiers. Only the first eight characters or identifiers are recognised although identifiers may be of arbitrary length, hence the first eight characters must uniquely define an identifier.

B. Other syntactic changes are as follows:

1. (1) Statements or declarations may be terminated by either end of line or π. Where there is no ambiguity no terminator is necessary. For example the following sequences are equivalent

   ```
   (i)
   BEGIN
   INTEGER I,J
   REAL K
   (ii)
   BEGIN INTEGER I,J,*
   REAL K
   (iii)
   BEGIN π INTEGER I,Jπ REAL K π
   ```

2. The dictionaries for identifiers and labels (procedure names are treated as labels) are distinct and the same names may appear in each. All labels must be distinct and there is no check that jumps from one transaction to another are not made and this could cause trouble. The names of local variables are local to a process.

C. A few changes have also been made in the interpretation of statements.

1. If not explicit in a seize statement, the seize strength is taken to be one, not zero. SEIZE FAC,0 has the effect of making the facility NOT BUSY but is not recommended as the facility is not released correctly.
2. There is no check that transactions do not release store not entered by them.
3. The cancel statement does not release the facilities or leave the stores associated with the transaction.
4. TIME is a preloaded integer which may, in fact, be used on the left hand side of an expression, probably with disastrous effects.

D. Two extra statements have been included:

1. DUMP: this statement causes all the variables, stores, facilities and tables to be printed together with information about each transaction. The dumping is in octal and is designed as a last ditch debugging aid.

2. CODE: This statement causes the code produced by the compiler to be dumped and also the jump table and the directories. This statement can be used in conjunction with the DUMP statement.

CHAPTER V

Compile time diagnostics

(1) IDENTIFIER NOT DEFINED
No corresponding REAL, INTEGER, or FACILITY declaration
(2) IDENTIFIER DEFINED TWICE
Caused by a REAL, INTEGER, or FACILITY declaration
(3) IDENTIFIER DEFINED TWICE, STORE
Caused by STORE declaration
(4) IDENTIFIER DEFINED TWICE, TABLE
Caused by TABLE declaration
(5) INSTRUCTION NOT RECOGNISED
The SOL declarations or any process is syntactically checked before the more detailed compilation takes place. Any failure in this checking will cause the above diagnostic which will be printed after the first line which could not be recognised. In general the compilation will have difficulty proceeding and will continue producing erroneous INSTRUCTION NOT RECOGNIZED on the rest of the source material
(6) LABEL SET TWICE
Self explanatory
(7) STORE BUSY
Non-facility found in a .BUSY or .NOT BUSY relation
(8) FACILITY FULL
Non-store found in a .FULL etc relation
(9) NON STORE IN ENTER
Faulty enter statement
(10) NON STORE IN LEAVE
Faulty leave statement
(11) NON FAC IN SEIZE
Faulty seize statement
(12) NON FAC IN RELEASE
Faulty release statement
(13) L.H.S NOT VAR
Left hand side of a replacement statement is not real or integer
(14) FAC. OR TAB. IN EXPRESSION
Facility or table used in an expression

CHAPTER VI

Run time diagnostics

(1) END OF SIMULATION
Caused by the stop statement
(2) NOTHING TO DO
All transactions halted
(3) ILLEGAL SEIZE
Transaction seizing a facility it already controls
(4) NEGATIVE SEIZE
Negative seize strength
(5) ILLEGAL RELEASE
Transaction releasing a facility it does not control
(6) LEAVE NEGATIVE
Transaction leaving a negative amount of store
(7) LEAVE TOO BIG
Transaction leaving more store than has been entered
(8) ENTER NEGATIVE
Transaction entering a negative amount of store
(9) ENTER TOO BIG
Transaction trying to enter more store than the capacity of the store
(10) TABULATE OUT OF RANGE
Tabulated quantity out of range of the table

Both NOTHING TO DO and END OF SIMULATION will be followed by the statistics. After an error, the simulation will attempt to continue and the statement in error will be omitted. This would probably, of course, cause further errors.

CHAPTER VII

Limitations

In general all arithmetic is executed in floating point and a number is truncated if an integer is required at any stage.

Real variables and expressions are printed with 8 places before the decimal point and 4 after. Integers are printed with a maximum of 8 digits.

Seize strengths and priorities may be from 0 to 1048575.

The size of program which can be compiled is governed by two factors - the store request, and the size of the largest process (or maybe the size of the global declarations if these are large). The exact size is difficult to estimate but processes of 150 lines have been compiled successfully.

There is no check on array bounds.

CHAPTER VIII

Sample program output

There follows a listing of the sample problem given by Knuth and McNeley rewritten in Atlas SOL. The program is prefaced by the job description to give the user some idea of the store and time requirements of a typical program. The final END card would normally be followed by a file card but in this case it is followed immediately by the program output. A listing of the program is always produced before execution.

```
JOB SOL EXAMPLE
COMPUTING 12500 INSTRUCTIONS
STORE 45/70 BLOCKS
OUTPUT 0 LINE PRINTER 700 LINES
TAPE 1 S.R.C. COMPILERS
COMPILER SPECIAL
SOL
          BEGIN
          FACILITY TU(6),SB(3),LINE,COMPUTER#
          INTEGER TUSTATE(6),SBNUMBER(6),TUMESSAGE(6)#
          TABLE (2000 STEP 500 UNTIL 15000)TABLE(6)#
          STORE 10 QUEUE(6)#

PROCESS MASTER CONTROL
          BEGIN S8NUMBER(1)=1# SBNUMBER(2)=2#
                S8NUMBER(3)=1# SBNUMBER(4)=2#
                S8NUMBER(5)=1# SBNUMBER(6)=3#
          WAIT 360000# STOP END#

PROCESS USERS#
          BEGIN INTEGER Q,START TIME,MESSAGE TYPE#
          NEW TRANSACTION TO START# NEW TRANSACTION TO START#
ORIGIN)   NEW TRANSACTION TO START# WAIT 0,5000# GO TO ORIGIN#
START)    Q=1,6# ENTER QUEUE(Q)#
          MESSAGE TYPE=(1,1,2,2,2,2,2,3,3,3)#
          SEIZE TO(Q)#
          TUMESSAGE(Q)=MESSAGE TYPE#
          WAIT 6000'8000#
          START TIME=TIME#
          OUTPUT 'TU',Q,'****SENUS*MESSAGE',MESSAGE TYPE.'*****AT*TIME',TIME #
          TUSTATE(Q)=1#
          WAIT UNTIL TUSTATE(Q).EQ.0#
          RELEASE TU(Q)# LEAVE QUEUE(Q)#
          TABULATE (TIME-START TIME) IN TABLE(Q)#
          OUTPUT '****TU',Q,'****RECIEVES*REPLY*AT*TIME',TIME#
          CANCEL END#
PROCESS PBU# BEGIN INTEGER S,T,WORDS#
        NEW TRANSACTION TO SCAN# T=3#
SCAN)   T=T+1# IF T.gt.6 THEN T=1# WAIT 1#
        S=SRNUMBER(T)#
        SEIZE LINE#
        WAIT 5# IF SB(S).BUSY THEN (WAIT 80# RELEASE LINE# GO TO SCAN#
        SEIZE SB(S)# WAIT 15# IF TUSTATE(T).NE.1 THEN{WAIT 65#
            RELEASE LINE# RELEASE SB(S)# GO TO SCAN)#
        WAIT 225# SEND) WAIT 170# IF PR(0.02) THEN (WAIT 20# GO TO SEND)#
        NEW TRANSACTION TO COMPUTATION# WAIT 20# RELEASE SB(S)#
        RELEASE LINE# TUSTATE(T)=2# CANCEL#
COMPUTATION) SEIZE COMPUTER# WORDS=TUMESSAGE(T)+2#
        WAIT (IF WORDS.EQ.3 THEN 250 ELSE IF WORDS.EQ.4 THEN 300 ELSE 400)#
        RELEASE  COMPUTER#
OUTPT)  WAIT 1#SEIZE LINE# WAIT 5#
        IF SB(S).BUSY THEN (WAIT 80# RELEASE LINE# GO TO OUTPT)#
        SEIZE SB(S)# WAIT 75#
RECEIVE) WAIT 80# IF PR(0.01) THEN (WAIT 20# GOTO  RECEIVE)#
        RELEASE LINE#
        WORDS=WORDS-1#
        IF WORDS.EQ.0 THEN  NEW TRANSACTION TO SCAN#
        WAIT 325# RELEASE SB(S)# WAIT 170#
        IF WORDS.GT.0 THEN  GO TO OUTPT#
        TUSTATE(T)#0# CANCEL END#

PROCESS OTHER PBUS#
        BEGIN INTEGER I# I=6#
CREATE) NEW  TRANSACTION TO COMPUTE#
        I=I-1# IF I.GT.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
```

```
TU         6   SENDS MESSAGE        1     AT TIME        6546
TU         3   SENDS MESSAGE        2     AT TIME        7532
TU         1   SENDS MESSAGE        3     AT TIME        7704
    TU        6     RECIEVES REPLY AT TIME     9610
    TU        1     RECIEVES REPLY AT TIME    13248
TU         2   SENDS MESSAGE        2     AT TIME       14390
    TU        3     RECIEVES REPLY AT TIME    14423
    TU        2     RECIEVES REPLY AT TIME    18152
TU         1   SENDS MESSAGE        1     AT TIME       26172
TU         3   SENDS MESSAGE        1     AT TIME       20979
    TU        1     RECIEVES REPLY AT TIME    23159
TU         2   SENDS MESSAGE        2     AT TIME       25651
    TU        3     RECIEVES REPLY AT TIME    26467
TU         1   SENDS MESSAGE        3     AT TIME       29706
    TU        2     RECIEVES REPLY AT TIME    30114
TU         6   SENDS MESSAGE        2     AT TIME       32347
TU         3   SENDS MESSAGE        2     AT TIME       32347
    TU        1     RECIEVES REPLY AT TIME    35001
    TU        6     RECIEVES REPLY AT TIME    36358
TU         2   SENDS MESSAGE        1     AT TIME       37507
TU         5   SENDS MESSAGE        3     AT TIME       37936
    TU        3     RECIEVES REPLY AT TIME    39065
    TU        2     RECIEVES REPLY AT TIME    41078
TU         1   SENDS MESSAGE        3     AT TIME       42975
    TU        5     RECIEVES REPLY AT TIME    43521
TU         6   SENDS MESSAGE        2     AT TIME       44910
    TU        1     RECIEVES REPLY AT TIME   318350
    TU        5     RECIEVES REPLY AT TIME   319030
TU         3   SENDS MESSAGE        1     AT TIME      319619
TU         4   SENDS MESSAGE        1     AT TIME      320555
    TU        2     RECIEVES REPLY AT TIME   322250
    TU        6     RECIEVES REPLY AT TIME   323066
    TU        3     RECIEVES REPLY AT TIME   324769
TU         5   SENDS MESSAGE        3     AT TIME      325871
    TU        4     RECIEVES REPLY AT TIME   325948
TU         2   SENDS MESSAGE        2     AT TIME      329381
    TU        5     RECIEVES REPLY AT TIME   330182
TU         3   SENDS MESSAGE        1     AT TIME      331590
    TU        2     RECIEVES REPLY AT TIME   333560
    TU        3     RECIEVES REPLY AT TIME   334481
TU         1   SENDS MESSAGE        3     AT TIME      335006
    TU        1     RECIEVES REPLY AT TIME   339821
TU         2   SENDS MESSAGE        3     AT TIME      340371
TU         4   SENDS MESSAGE        2     AT TIME      341458
TU         3   SENDS MESSAGE        2     AT TIME      341820
    TU        2     RECIEVES REPLY AT TIME   344806
    TU        3     RECIEVES REPLY AT TIME   346163
TU         5   SENDS MESSAGE        3     AT TIME      347808
    TU        4     RECIEVES REPLY AT TIME   348864
TU         2   SENDS MESSAGE        2     AT TIME      351765
    TU        5     RECIEVES REPLY AT TIME   351912
TU         6   SENDS MESSAGE        2     AT TIME      352493
TU         3   SENDS MESSAGE        1     AT TIME      353549
    TU        2     RECIEVES REPLY AT TIME   354698
TU         1   SENDS MESSAGE        3     AT TIME      354974
TU         4   SENDS MESSAGE        2     AT TIME      356456
    TU        6     RECIEVES REPLY AT TIME   356606
    TU        3     RECIEVES REPLY AT TIME   357202
TU         5   SENDS MESSAGE        3     AT TIME      359688
END OF SIMULATION
```

CLOCK TIME AT END OF SIMULATION WAS 380000

NUMBER OF TIMES LABELS WERE ENCOUNTERED

LABEL	-	COUNT	LABEL	-	COUNT	LABEL	-	COUNT
START	-	150	ORIGIN	-	148	SCAN	-	1993
SEND	-	141	COMPUTAT	-	138	OUTPT	-	649
RECEIVE	-	564	CREATE	-	6	COMPUTE	-	477

NAME OF FACILITY		FRACTION OF TIME IN USE
TU	(001)	0.7839
TU	(002)	0.9145
TU	(003)	1.0000
TU	(004)	0.7281
TU	(005)	0.5014
TU	(006)	0.6470
SB	(001)	0.5967
SB	(002)	0.4437
SB	(003)	0.1959
LINE	(001)	0.8771
COMPUTER	(001)	0.5412

NAME OF STORE		CAPACITY	MAXIMUM USED	AVERAGE OCCUPANCY	AVERAGE UTILZATION
QUEUE	(001)	10	5	1.6720	0.1672
QUEUE	(002)	10	6	2.4227	0.2422
QUEUE	(003)	10	8	3.6986	0.3698
QUEUE	(004)	10	5	1.7048	0.1704
QUEUE	(005)	10	4	0.7010	0.0701
QUEUE	(006)	10	3	0.8416	0.0841

TABLE NAME IS TABLE (001)

NUMBER OF TABLE ENTRIES IS 22 SUM OF ALL ENTRY VALUES

MEAN OF TABLE 4542.2727 STANDARD DEVIATION 1276.9288

UPPER LIMIT	NUMBER	PER CENT	CUMULATIVE	MULTIPLE OF MEAN
2000		0.00	0.00	0.4403
2500		0.00	0.00	0.5503
3000	3	13.63	13.63	0.6604
3500	1	4.54	18.18	0.7705
4000	5	22.72	40.90	0.8806
4500	4	18.18	59.09	0.9906
5000	1	4.54	63.63	1.1007
5500	4	18.18	81.81	1.2108
6000	1	4.54	86.36	1.3209
6500	1	4.54	90.90	1.4310
7000		0.00	90.90	1.5410
7500	2	9.09	99.99	1.6511
8000		0.00	99.99	1.7612
8500		0.00	99.99	1.8713
9000		0.00	99.99	1.9813
9500		0.00	99.99	2.0914
10000		0.00	99.99	2.2015
10500		0.00	99.99	2.3116
11000		0.00	99.99	2.4216
11500		0.00	99.99	2.5317
12000		0.00	99.99	2.6418
12500		0.00	99.99	2.7519
13000		0.00	99.99	2.8620
13500		0.00	99.99	2.9720
14000		0.00	99.99	3.0821
14500		0.00	99.99	3.1922
15000		0.00	99.99	3.3023

TABLE NAME IS TABLE (002)

NUMBER OF TABLE ENTRIES IS 28 SUM OF ALL ENTRY VALUES

MEAN OF TABLE 4630.4265 STANDARD DEVIATION 1318.5882

UPPER LIMIT	NUMBER	PER CENT	CUMULATIVE	MULTIPLE OF MEAN
2000		0.00	0.00	0.4319
2500		0.00	0.00	0.5399
3000	2	7.14	7.14	0.6478
3500	1	3.57	10.71	0.7558
4000	4	14.28	24.99	0.8638
4500	10	35.71	60.71	0.9718
5000	6	21.42	82.14	1.0798
5500		0.00	82.14	1.1877
6000	1	3.57	85.71	1.2957
6500	1	3.57	89.28	1.4037
7000	1	3.57	92.85	1.5117
7500		0.00	92.85	1.6197
8000	1	3.57	96.42	1.7277
8500		0.00	96.42	1.8356
9000	1	3.57	99.99	1.9436
9500		0.00	99.99	2.0516
10000		0.00	99.99	2.1596
10500		0.00	99.99	2.2676
11000		0.00	99.99	2.3755
11500		0.00	99.99	2.4835
12000		0.00	99.99	2.5915
12500		0.00	99.99	2.6995
13000		0.00	99.99	2.8075
13500		0.00	99.99	2.9154
14000		0.00	99.99	3.0234
14500		0.00	99.99	3.1314
15000		0.00	99.99	3.2394

TABLE NAME IS TABLE (003)

NUMBER OF TABLE ENTRIES IS 30 SUM OF ALL ENTRY VALUES

MEAN OF TABLE 4930.6000 STANDARD DEVIATION 1157.1875

UPPER LIMIT	NUMBER	PER CENT	CUMULATIVE	MULTIPLE OF MEAN
2000		0.00	0.00	0.4056
2500		0.00	0.00	0.5070
3000	1	3.33	3.33	0.6084
3500	1	3.33	6.66	0.7098
4000	7	23.33	30.00	0.8112
4500	3	10.00	39.99	0.9126
5000	3	10.00	50.00	1.0140
5500	6	19.99	69.99	1.1154
6000	4	13.33	83.33	1.2168
6500	2	6.66	89.99	1.3182
7000	2	6.66	96.66	1.4197
7500		0.00	96.66	1.5211
8000	1	3.33	99.99	1.6225
8500		0.00	99.99	1.7239
9000		0.00	99.99	1.8253
9500		0.00	99.99	1.9267
10000		0.00	99.99	2.0281
10500		0.00	99.99	2.1295
11000		0.00	99.99	2.2309
11500		0.00	99.99	2.3323
12000		0.00	99.99	2.4337
12500		0.00	99.99	2.5351
13000		0.00	99.99	2.6365
13500		0.00	99.99	2.7380
14000		0.00	99.99	2.8394
14500		0.00	99.99	2.9408
15000		0.00	99.99	3.0422

TABLE NAME IS TABLE (004)

NUMBER OF TABLE ENTRIES IS 22 SUM OF ALL ENTRY VALUES

MEAN OF TABLE 4442.2272 STANDARD DEVIATION 870.2028

UPPER LIMIT	NUMBER	PER CENT	CUMULATIVE	MULTIPLE OF MEAN
2000		0.00	0.00	0.4502
2500		0.00	0.00	0.5627
3000		0.00	0.00	0.6753
3500	4	18.18	18.18	0.7878
4000	2	9.09	27.27	0.9004
4500	7	31.81	59.09	1.0130
5000	6	27.27	86.36	1.1255
5500	2	9.09	95.45	1.2381
6000		0.00	95.45	1.3506
6500		0.00	95.45	1.4632
7000		0.00	95.45	1.5757
7500	1	4.54	99.99	1.6883
8000		0.00	99.99	1.8008
8500		0.00	99.99	1.9134
9000		0.00	99.99	2.0260
9500		0.00	99.99	2.1385
10000		0.00	99.99	2.2511
10500		0.00	99.99	2.3636
11000		0.00	99.99	2.4762
11500		0.00	99.99	2.5887
12000		0.00	99.99	2.7013
12500		0.00	99.99	2.8139
13000		0.00	99.99	2.9264
13500		0.00	99.99	3.0390
14000		0.00	99.99	3.1515
14500		0.00	99.99	3.2641
15000		0.00	99.99	3.3766

TABLE NAME IS TABLE (005)

NUMBER OF TABLE ENTRIES IS 14 SUM OF ALL ENTRY VALUES

MEAN OF TABLE 5292.9285 STANDARD DEVIATION 1676.4334

UPPER LIMIT	NUMBER	PER CENT	CUMULATIVE	MULTIPLE OF MEAN
2000		0.00	0.00	0.3778
2500		0.00	0.00	0.4723
3000		0.00	0.00	0.5667
3500	2	14.28	14.28	0.6612
4000		0.00	14.28	0.7557
4500	5	35.71	50.00	0.8501
5000	1	7.14	57.14	0.9446
5500		0.00	57.14	1.0391
6000	2	14.28	71.42	1.1335
6500		0.00	71.42	1.2280
7000	2	14.28	85.71	1.3225
7500		0.00	85.71	1.4169
8000		0.00	92.85	1.5114
8500	1	7.14	99.99	1.6059
9000	1	7.14	99.99	1.7003
9500		0.00	99.99	1.7948
10000		0.00	99.99	1.8893
10500		0.00	99.99	1.9837
11000		0.00	99.99	2.0782
11500		0.00	99.99	2.1727
12000		0.00	99.99	2.2671
12500		0.00	99.99	2.3616
13000		0.00	99.99	2.4561
13500		0.00	99.99	2.5505
14000		0.00	99.99	2.6450
14500		0.00	99.99	2.7395
15000		0.00	99.99	2.8339

```
                    TABLE NAME IS   TABLE    (006)
    NUMBER OF TABLE ENTRIES IS       20                 SUM OF ALL ENTRY VALUES
MEAN OF TABLE        4581.3500     STANDARD DEVIATION      1180.9796
```

UPPER LIMIT	NUMBER	PER CENT	CUMULATIVE	MULTIPLE OF MEAN
2000		0.00	0.00	0.4365
2500		0.00	0.00	0.5456
3000		0.00	0.00	0.6548
3500	2	10.00	10.00	0.7639
4000	4	19.99	30.00	0.8731
4500	6	30.00	60.00	0.9822
5000	2	10.00	69.99	1.0913
5500	3	14.99	84.99	1.2005
6000	1	5.00	89.99	1.3096
6500		0.00	89.99	1.4187
7000	1	5.00	94.99	1.5279
7500		0.00	94.99	1.6370
8000		0.00	94.99	1.7462
8500	1	5.00	99.99	1.8553
9000		0.00	99.99	1.9644
9500		0.00	99.99	2.0736
10000		0.00	99.99	2.1827
10500		0.00	99.99	2.2919
11000		0.00	99.99	2.4010
11500		0.00	99.99	2.5101
12000		0.00	99.99	2.6193
12500		0.00	99.99	2.7284
13000		0.00	99.99	2.8375
13500		0.00	99.99	2.9467
14000		0.00	99.99	3.0558
14500		0.00	99.99	3.1650
15000		0.00	99.99	3.2741

```
SOL EXAMPLE
DATE 19.12.66
TIME 15.52.41
SERIAL NUMBER    15537143
                              REQUESTED        USED      COMPILE
INSTRUCTION INTERRUPTS            12500       10304         1098
COMPILE STORE                        70          68
EXECUTION STORE                      45          40
                                  DECKS      BLOCKS      WAITING
MAGNETIC TAPES                        2          32           15
                             STORE TIME  DRUM TIME    DECK TIME
                                   4058           1          156
COMPILER NUMBER                      29

INPUT    0            2    BLOCKS   READER    0
OUTPUT   0          564    RECORDS PRIVATE TAPE
```