ENHANCEMENTS TO THE GPSS/SAS® COMPILER

Michael A. Greene, The American University
Gretchen K. Jones, National Center for Health Statistics

ABSTRACT

This paper describes Version 3 of the GPSS SAS compiler. GPSS (IBM, 1971) is a popular computer language used for simulating queuing processes. Our implementation of GPSS uses the SAS® system to translate and execute the GPSS code. This permits the use of procedures from SAS/STAT and SAS/GRAFI for analysis and display of the output data from the simulation.

There are a number of enhancements over version 2 (Greene and Jones, 1990) principally including the implementation of a useful subset of GPSS. This version also recognizes SAS statements as GPSS operands so that any SAS function may be used. This represents a significant extension to GPSS, a language which does not contain mathematical or statistical functions. The paper discusses these issues and also how the compiler works.

1. Introduction

GPSS is an event-driven simulation language developed by IBM (IBM, 1971) and is among the more popular of such languages currently in use. Texts describing GPSS include Carson, Banks and Sy (1989), Durning (1986) and Schriber (1974); and the main forms of the language currently in use include GPSS V (IBM, 1973), and GPSS/H (Henricksen and Crain, 1988). Simulation languages differ from general purpose languages in the way that statements are scheduled for execution. General purpose languages execute the next statement unless otherwise instructed by a GO TO or equivalent instruction. In contrast, the simulation language represents a transaction proceeding through a flowchart where the transaction is subject to delays due to unavailable resources and interference from other transactions. Each statement must begin on a new line and contains an optional label, an operation field, an auxiliary operation field required by some operation fields and up to eight operands separated by commas. These statements are illustrated in figure 2 below which shows a complete program for simulating a two server queue. In that figure, SERVER is in the label field of the STORAGE statement, everything directly under SIMULATE to END are operation fields, there are no auxiliary operations and the word SERVER to the right of ENTER and LEAVE and the numbers are operands. Each statement has one operand except SIMULATE and END which have none. Examples of auxiliaries include relational operators such as GE, EQ, and LE found on TEST blocks.

Figure 2
A two server queue in GPSS

The basic idea of the GPSS language is a transaction making progress through a flowchart where the transaction is subject to delays due to unavailable resources and interference from other transactions. Each statement must begin on a new line and contains an optional label, an operation field, an auxiliary operation field required by some operation fields and up to eight operands separated by commas. These statements are illustrated in figure 2 below which shows a complete program for simulating a two server queue. In that figure, SERVER is in the label field of the STORAGE statement, everything directly under SIMULATE to END are operation fields, there are no auxiliary operations and the word SERVER to the right of ENTER and LEAVE and the numbers are operands. Each statement has one operand except SIMULATE and END which have none. Examples of auxiliaries include relational operators such as GE, EQ, and LE found on TEST blocks.

Data step       Function
FILED           FILEREFS associating internal and external file names
TABLES          symbol table

LEXAN error messages
ERRMSGS code generation and error checks
PASS2 further compiling and execution
RUNSTEP compiler in Section 4.

Version 2 also contained two blocks not found in other implementations, namely the REGS and REGS block which implement the regenerative method (Iglehart, 1978). Also, the queue was changed from a linear list to a linked list for gains in efficiency. The current version, Version 3 implements most of the functionality of GPSS V (IBM, 1971) and was developed using SAS 6.06 under VM/CMS. This version is discussed below in Section 3, following a brief introduction to GPSS in Section 2. This is then followed by an example of the output from the compiler in Section 4.

2. The GPSS language (in brief)

The statements of the language are separated into GPSS control statements and GPSS blocks. GPSS control statements are considered non-executable and are all performed before execution of the first block. Examples of control statements in figure 2 include SIMULATE, a
Aho, Sethi and Ullman (1986) define a compiler as a program which reads in a program written in one language (source program) and translates it into another language (target program). There are two main phases to the translation, (1) analysis and (2) code generation. The analysis phase begins with reading in the program as character by character, then dividing the line into labels, operation fields, auxiliaries and operands. These are then assumed to be separated from each other by a comma unless there is a left parenthesis (a reference to a matrix or a SAS function call). The occurrence of the first blanks space in the operand fields signals the end of the statement. The outgoing variables in Work.LEXAN are Lab (label), Operate (operation field), Aux (Auxiliary), Op1-Op8 (Operands), Comflag (Comment Flag) and Nop (number of operands).

### 3.2 PASS2

PASS2 is about 600 lines of SAS instructions to perform GPSS entity translation, symbol table maintenance, storage allocation, macro variable creation for array dimensioning, syntax analysis, compile-time error messaging, translation of GPSS random number calls to SAS random number subroutine calls, and creation of the dynamic portions of RUNSTEP. In that part of the program, GPSS labels are assigned to SAS variables and initialized, storage for various GPSS entities is created, GPSS operands are translated to SAS expressions and pointers of various types are set up. Then RUNSTEP (section 3.3 below) needs only two pieces of information, (1) the type of operation field being executed, and (2) the values of the operands at the time the statement is being executed. The operation field where some final translation occurs and the model is run. The flow looks like this:

![Figure 4](image_url)

#### Compiler Phases

- **SIMDATA** → **LEXAN** ← **TABLES**
- **SELECT** ← **PASS2** → **INITIALS**
- **-----** → **RUNSTEP** ←**-----**

These phases are described below.

#### 3.1 LEXAN

This data step is about 225 lines. It reads in a line or more (if it detects a continuation character), then divides the line into labels, operation fields, auxiliaries and operands. LEXAN requires two input datasets as shown in figure 4. TABLES is a SAS dataset containing a symbol table of reserved words that is used in LEXAN to determine whether a particular operation field is a valid operation field and whether this field takes an auxiliary or not. TABLES is brought in on the first pass of LEXAN and the symbol table is retained through subsequent passes. Then one line at a time is read in from SIMDATA, the GPSS source language simulation program which is to be compiled. The line is parsed into label (optional), operation, auxiliary (required by some operations), and operands. Free format, mixed case input is permitted. The symbol table (in TABLES) is used to determine if the first token on a line is an operation field (symbol found) or a label (not found). The remaining items on the line are operands. These are then assumed to be separated from each other by a comma unless there is a left parenthesis (a reference to a matrix or a SAS function call). The occurrence of the first blanks space in the operand fields signals the end of the statement.

The outgoing variables in Work.LEXAN are Lab (label), Operate (operation field), Aux (Auxiliary), Op1-Op8 (Operands), Comflag (Comment Flag) and Nop (number of operands).
code is passed through the PASS2 SAS dataset, while the operands' values are obtained one of the dynamic portions, the SELECT dataset.

These activities will be illustrated with the GPSS program in Figure 5. The program has already been tokenized by LEXAN. Labels occur in the first few columns, operation fields to their right and operands further right.

**Figure 5**
GPSS program prior to PASS2

<table>
<thead>
<tr>
<th>Line</th>
<th>Operation</th>
<th>Value(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMULATE</td>
<td>3,2</td>
</tr>
<tr>
<td>2</td>
<td>MIKE</td>
<td>MATRIX</td>
</tr>
<tr>
<td>3</td>
<td>GRET</td>
<td>MATRIX</td>
</tr>
<tr>
<td>4</td>
<td>INITIAL</td>
<td>M$MIKE(3,2),1</td>
</tr>
<tr>
<td>5</td>
<td>INITIAL</td>
<td>M$GRET(2,3),55</td>
</tr>
<tr>
<td>6</td>
<td>INITIAL</td>
<td>X$JON,2</td>
</tr>
<tr>
<td>7</td>
<td>INITIAL</td>
<td>X$ERICA,3</td>
</tr>
<tr>
<td>8</td>
<td>XEROX</td>
<td>STORAGE</td>
</tr>
<tr>
<td>9</td>
<td>TOSHI</td>
<td>STORAGE</td>
</tr>
<tr>
<td>10</td>
<td>LABEL1</td>
<td>GENERATE</td>
</tr>
<tr>
<td>11</td>
<td>ASSIGN</td>
<td>1,M$MIKE(3,2)</td>
</tr>
<tr>
<td>12</td>
<td>LOGIC</td>
<td>R SWITCH1</td>
</tr>
<tr>
<td>13</td>
<td>GATE</td>
<td>LR SWITCH1</td>
</tr>
<tr>
<td>14</td>
<td>SAVEVALUE</td>
<td>GRET,3,2,M$MIKE(P1,2)+1</td>
</tr>
<tr>
<td>15</td>
<td>SAVEVALUE</td>
<td>DAN,2</td>
</tr>
<tr>
<td>16</td>
<td>TEST</td>
<td>L M$MIKE(3,2),12,SKIP</td>
</tr>
<tr>
<td>17</td>
<td>ENTER</td>
<td>XEROX</td>
</tr>
<tr>
<td>18</td>
<td>ADVANCE</td>
<td>5#1</td>
</tr>
<tr>
<td>19</td>
<td>LEAVE</td>
<td>XEROX</td>
</tr>
<tr>
<td>20</td>
<td>SKIP</td>
<td>TERMINATE</td>
</tr>
<tr>
<td>21</td>
<td>START</td>
<td>50</td>
</tr>
<tr>
<td>22</td>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

First, the method of storage allocation is discussed. This is then followed by describing the two inserts into RUNSTEP, INITIALS and SELECT.

### 3.2.1 STORAGE SETUP in PASS2

GPSS has three types of variables: (1) Savevalues are scalar variables, prefixed with X$, (2) Two dimensional matrices prefixed with M$, and (3) Parameters prefixed by P and followed by a number, e.g. P1. Savevalues and matrices are static storage, identical to variables and arrays in any programming language. Parameters are specific to a transaction, that is there may be several different P1's active at any given time.

GPSS syntax requires matrices to be dimensioned before they are used. This occurs in the MATRIX statement, where the matrix name (MIKE) appears as the label, the row dimension (3) in the first operand and the column dimension in the second operand (viz. MIKE MATRIX 3,2). The matrix name is installed in a table for subsequent lookup, the number of rows is stored in NROWS[], the number of columns in NCOLS[] and the starting position of the matrix is stored in OFF[]. This is written out as part of the cross reference output below as shown in figure 6 below:

**Figure 6**
Cross Reference Output

<table>
<thead>
<tr>
<th>OBS NAME</th>
<th>LINE OFFSET NUMBER NROWS NCOLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIKE</td>
<td>1  2  1  1  3  2</td>
</tr>
<tr>
<td>GRET</td>
<td>2  3  2  7  2  5</td>
</tr>
</tbody>
</table>

The actual data for the matrix are stored in the M[*] array in row major order. Matrices are detected (1) in operands from the character string M$ or (2) positionally. When detected, the position in the M[*] array is computed and installed in the SELECT dataset, i.e., the I,J position of the Kth matrix is computed as OFF[K]+NC[K]*(I-1)+J-1. For example the statement INITIAL M$MIKE(3,2),1 which is intended to put the value 1 into the 3rd row and 2nd column of MIKE has the array reference translated as OFF[001]+NC[001]*(3-1)+2-1. Savevalues handling is similar to matrices, except that only the symbolic translation from name to number is required.

Parameters are detected by finding a leading P or a P preceded by an operator or parenthesis, followed by a number. Parameters represent a little matrix which belongs to a transaction and are very useful in GPSS because they can carry values that are specific to transactions. That could include, for example, the time that a transaction went through a particular block.

To store parameter values, we allocate first a fixed amount of storage to each transaction in the TA[*] array where the sixth position points to the address of the header in the PARMTX[*] array. Immediately after the header is one position for each parameter as requested in the GENERATE block. For example in Figure 5, the last operand of the GENERATE block indicates that space must be created for 2 parameters. The kth parameter for a given transaction would be found by the following calculation PARMTX{TA[P1+5] + K}, or in our case, referencing the first parameter in line 18 would produce the statement PARMTX{TA(P1+5)+1}. This statement would also be written out to SELECT.

All of the GPSS entities in the program are written out to a cross reference file which is made available to the user for debugging.

Line in the line number where the entity appears and number is the symbolic number for the entity. SWITCH1 does not have a symbolic number because logic switches are handled with the INITIALS dataset and no symbol table is maintained. MIKE and GRET are matrices, MIKE is assigned the number 1 because it was dimensioned first. Note also that there are four entries for MIKE corresponding to the four places where it was referenced.ERICA and JON are savevalues(SAVEX). There are also three references to XEROX corresponding to where that STORAGE was dimensioned, and used in ENTER and LEAVE blocks. See below.
3.2.2 INITIALS

PASS2 also creates the two dynamic portions of RUNSTEP, SELECT and INITIALS. Both are kept as external files, then included into RUNSTEP at execution time. INITIALS contains dimensions for SAS arrays to hold the various GPSS variables, and dimensions for block and storage variables. This permits dynamic sizing of arrays, which is critical because storage space is limited in the SAS program data vector (PDV). Also, INITIALS initializes label variables, storage capacity variables, the number of columns of matrices, and the addresses of switches. An example of INITIALS is shown in Figure 7 below:

Finally, the remaining two arrays, STORCAP(*) and STORUSE(*) represent storage capacity and storage usage and are dimensioned to 2 for the 2 storages in the model.

The last section of INITIALS contains initialization information. The first storage (XEROX) has a capacity of 2 and the second (TOSHI) has a capacity of 3. These are followed by matrix initialization, NC(1) at 2, MIKE has 2 columns and starts at offset 1, while GRET has 5 columns and starts at offset 7. Finally, there are the line numbers assigned to Labels and Switches. SKIP gets the value 20, LABEL1 gets the value 10 and SWITCH1 gets the value 12. Any reference then to SKIP later in RUNSTEP produces the value 20, which is the destination of a false outcome of the TEST block in line 16. Similarly, a reference to SWITCH1 in line 13 causes examination of the status of the SWITCH1 in statement 12 to determine whether a transaction may pass through.

3.2.3 SELECT

As mentioned above, SELECT is the other file created in PASS2 which is then incorporated into RUNSTEP. SELECT contains SAS statements which represent the translation of operands into the SAS language. Every time a transaction enters a block in RUNSTEP, the code segment produced by PASS2 for that particular block is linked (GETOPS) and executed. The general format for this code is as follows:

```
GETOPS:
 T1 = ; T2 = ; ( up to T8 )
SELECT ( curblo );
 WHEN ( block_number ) DO ;
< Sas language statements here >
END;
 WHEN ( block_number ) DO ;
< Sas language statements here >
END;
END;
RETURN;
```

_Tk is a temporary variable which stands for the value of the kth operand. Curblo is a RUNSTEP variable which contains the line number of the block that is being executed, while block_number is a constant for a particular block.

The output is produced by scanning the operands in PASS2, successively refining the output and then writing it out. Consider, for example, the four operands in the MSAVEVAL statement in line 14 in Figure 5:

```
GRET,3,2,MIKE(P1,2)+1
```

The intention of this statement is to place the value of MIKE(P1,2)+1 into GRET(3,2). PASS2 translates it as follows:
The main data structures in RUNSTEP are described below. The four main data structures are described below. The first operand of an MSAVEVAL block is expected to be a matrix name. A table lookup shows that the matrix number 2, so $T_1 = 2$.

The rows and columns appearing in the second and third operand are placed in $T_2$ and $T_3$. When scanning the fourth operand, the parameter reference, P1 is first detected. A temporary variable $T_4$ is created to hold the value of that parameter using the addressing scheme described above. Then the correct address of MIKE(P1,2) is placed into $T_4$ and finally the fourth operand, $T_4$ gets the value in MIKE(P1,2).

Inside RUNSTEP, when the MSAVEVAL logic is executed the addressing is reversed, i.e., $T_4$ gets the value of MIKE(3,2). Inside RUNSTEP, when the MSAVEVAL logic is executed the addressing is reversed, i.e., $T_4$ gets the value of MIKE(3,2). Inside RUNSTEP, when the MSAVEVAL logic is executed the addressing is reversed, i.e., $T_4$ gets the value of MIKE(3,2). Inside RUNSTEP, when the MSAVEVAL logic is executed the addressing is reversed, i.e., $T_4$ gets the value of MIKE(3,2). Inside RUNSTEP, when the MSAVEVAL logic is executed the addressing is reversed, i.e., $T_4$ gets the value of MIKE(3,2).

3.3 RUNSTEP

RUNSTEP is a data step of about 700 lines of SAS statements. It does a small amount of compilation and then executes the simulation. The main data structures are described below, followed by the manner in which the simulation is executed.

3.3.1 Data Structures

The main data structures in RUNSTEP are those associated with transactions TA[*], NEXT[*] and _PARMTX[*] and block arrays (BLKTYPE[*], BLKAUX[*], BLKCNT[*] and BLKMISC[*]). The transaction array is dimensioned beforehand to the best guess at the maximum number of active transactions * 10. The block arrays are dynamically dimensioned to the block counts in INITIALS as shown in figure. The block arrays are described first followed by the transaction arrays.

The block arrays contain information which is specific to the blocks. Filling the block arrays is the last step in compilation and is done in the beginning of RUNSTEP. BLKTYPE[*] gets a number representing the operation field of the block. BLKAUX[*] gets the auxiliary operand. BLKMISC[*] is used for miscellaneous operations on blocks such as the value of the logical evaluation of the test block, etc. BLKCNT[*] is the block count or the number of transactions which have passed through the block.

TA[*] is a linked list with the pointers in NEXT[*]. TA[*] contains most of the relevant information about the transaction including its block departure time (BDT), number, current block occupied by the transaction, transaction status (active, blocked or terminated), maximum number of parameters, pointer to starting place in the parameter matrix (_PARMTX[*]) and priority. BDT represents the time that the transaction may be moved from its current block. Transactions are linked by BDT and priority, that in NEXT[*] points to the transaction with the same (and lower priority) BDT or next larger BDT. This allows scanning the transaction array from the beginning in order to find the next transaction to be moved. Transactions are inserted in the TA[*] array when created in the GENERATE block and removed when they move through the TERMINATE block. The position of the transaction may be modified by traversing through an ADVANCE (which causes revision of the BDT) or a PRIORITY block (changing the priority).

Management of the _PARMTX[*] array is a bit different because a variable rather than a fixed allocation of storage is required. The organization of this array is as follows: the $i$th position contains either a pointer to the TA[*] array pointing to the first position for that transaction (its BDT) or a negative number which was the number of parameters the last transaction used which occupied that space. Space is allocated in the logic for the GENERATE block which contains a pointer to the next free space. Should there not be enough free space, the array is compressed by moving all the space used by currently active transactions to the front of the array (and resetting pointers).

3.3.2 Simulation Execution

Before beginning the simulation, at the point where each GENERATE block is symbolized, its first transaction is made, space allocated in the _PARMTX[*] array, its block departure time computed and it is installed in the TA[*] linked list. The first transaction is then taken off the top of the linked list and the simulation clock is set to its BDT. Then the following pattern ensues:
1. The transaction is moved as far as it can be moved. It is then destroyed or put back into the linked list.

2. The next transaction is identified.

3. The simulation clock is updated if required in the block departure time for the next transaction.

4. If the termination counter is zero, the simulation stops, otherwise return to step 1.

Figure 11 shows moving the first transaction through the model. Time is the simulated time that the event is occurring, Xact is the transaction number, BDT is the block departure time, Bnum is the line number of the block, Ptr points to the _PARMTX{} array where the parameter values are stored, Bcode is the block number and Btype its type, Status is 1 for active—other values for inactive.

**Figure 11a Transaction 1 history**

<table>
<thead>
<tr>
<th>Time Xact</th>
<th>BDT</th>
<th>Bnum</th>
<th>Ptr</th>
<th>Bcode</th>
<th>Btype Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
<td>10</td>
<td>.</td>
<td>GENE 0</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
<td>11</td>
<td>1</td>
<td>ASS 1</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
<td>12</td>
<td>1</td>
<td>LOG 1</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
<td>13</td>
<td>1</td>
<td>GATE 1</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
<td>14</td>
<td>1</td>
<td>MSAV 1</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>1</td>
<td>SAVE 1</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
<td>16</td>
<td>1</td>
<td>TEST 1</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
<td>17</td>
<td>1</td>
<td>ENTE 1</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
<td>18</td>
<td>1</td>
<td>ADVA 1</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>20</td>
<td>18</td>
<td>1</td>
<td>ADVA 0</td>
</tr>
</tbody>
</table>

Figure 6 shows that the first transaction is to be manufactured at 15 and all subsequent transactions are to be manufactured on 10 unit intervals. At initialization time transaction 1 is created. It is the only transaction in the model, so it gets moved first. As shown above at time 15 it moves through to the advance block where it gets delayed for 5 time units and it cannot leave until 20. At this point, the linked list is scanned and another transaction, transaction 2 is found there as well. However, transaction 2 does not become activated until 22. Consequently, the clock has to be advanced to 20 and transaction 1 reactivated. This occurs in Figure 11b.

**Figure 11b Continuation of Transaction 1**

<table>
<thead>
<tr>
<th>Time Xact</th>
<th>BDT</th>
<th>Bnum</th>
<th>Ptr</th>
<th>Bcode</th>
<th>Btype Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>18</td>
<td>1</td>
<td>ADVA 0</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>19</td>
<td>1</td>
<td>LEAV 1</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>1</td>
<td>TERM 1</td>
</tr>
</tbody>
</table>

Transaction 1 at time 20 can move all the way to the terminate block. It can now be removed from the model and transaction 2 activated. This requires advancing the clock to 25. The progress of transaction 2 is similar to transaction 1.

The output shown above is principally useful for debugging the simulation model. For real world analyses, a much more abbreviated SAS dataset is prepared from the model containing the transaction number, time, and the block number that was entered. This dataset can then be processed by DATA and PROC steps using the powerful statistical and graphical procedures in the SAS system for analyses. See Jones and Greene, 1988 for an example.

4. Conclusion

This paper has described a GPSS compiler written in the SAS language. The compiler has been enhanced since its last presentation at SUGI to the point that it contains a reasonable subset of GPSS. We plan to test it with some practical modelling problems in the near future.

REFERENCES


