Proceedings of the 1990 Winter Simulation Conference
Osman Balci, Randall P. Sadowski, Richard E. Nance (eds.)

MODELING EFFICIENTLY WITH GPSS/H

Robert C. Crain
Daniel T. Brunner

Wolverine Software Corporation
4115 Annandale Road
Annandale, VA 22003-2500

ABSTRACT

Efficient modeling has two components. First, the simulation model itself should be as economical as possible in its use of computer time. Second, the modeling tool should contain features to ease the modeler's burden when creating the model. Both components yield time savings, which can be used either to reduce the project budget or to do additional work within the original budget. This tutorial will discuss ways in which GPSS/H contributes to both components.

1. INTRODUCTION

There are many reasons for variations in the speed with which simulation models execute, not the least of which is the choice of simulation tool. GPSS/H has long been known for its exceptional performance, especially when large, complicated models are to be run. Even among GPSS/H models, however, noticeable differences in performance can occur, due to the size and complexity of a model, the number of simultaneously active Transactions in a model, and the management of Transactions which become blocked until a change-of-state occurs somewhere in a model. The first part of this tutorial will focus primarily on the latter two of these three causes.

Over the past ten years, many features have been added to GPSS/H which provide greater functionality and ease of use, while retaining the intrinsic strengths of the Transaction-flow approach. The second part of this tutorial will examine some of these features and show how they may be used to solve modeling problems with less effort on the part of the modeler.

2. ACHIEVING MODEL EXECUTION EFFICIENCY

2.1 Minimize Model Size and Complexity, Especially the Number of Transactions

Some of the most important decisions that a simulationist makes have to do with how much detail should be present in a model, and how that detail should be represented given the simulation tool to be used. In GPSS/H, many of these decisions boil down to questions of which things in a system should be represented by Transactions.

By definition, Transactions are *active* objects of *individual* interest. Unfortunately, the *active* property sometimes dominates a modeler's thinking so that many, many Transactions are brought into the model at one time, in order to represent all the "widgets" moving through the system. This is done even though most of the Transactions do not represent objects of individual interest. The result is a great glut of Transactions that slows down model execution and consumes large amounts of memory.

There are two rules of thumb that should always be applied during model design. First, do not use a Transaction to represent anything in a system that is not of individual interest. If a lot of fifty identical parts is to be tracked through a system, then usually there should be one Transaction per lot of fifty parts, not one Transaction per part. In addition to the memory required for any Parameters (modeler-specified Transaction data items), a Transaction requires about 80 bytes of memory for system housekeeping purposes. The 80-byte-per-Transaction overhead can add up in a hurry. In the fifty-part example above, forty-nine superfluous Transactions would waste almost 4000 bytes of memory.

Second, don't create Transactions before they are actually needed (can begin to move through the model). In a typical model, Transactions come into the model, move through it over time, and are destroyed. After a Transaction is destroyed, its memory is available for reincarnation as a new Transaction. The total memory needed to run a model consists of that amount necessary to hold the model's code and data, plus enough to hold the *maximum number of Transactions that are in the model at any one time.* Unnecessary Transactions thus artificially drive up the apparent memory needs of a model. Transactions that have come into a model, moved, and then become blocked serve an important purpose: representing the contention and queueing that the modeler is trying to study. Transactions that are unable to move at all upon initial entry to the model are seldom, if ever, necessary. The same is true of Transactions in the midst of an initial movement that is not itself of interest (*e.g.*, Transactions on a long buffer conveyor from a loading dock).

2.2 Manage Waiting Transactions

After having made sure that there are no (or relatively few) unnecessary Transactions in a model, the modeler should concentrate on the management of Transactions that are blocked, waiting for a state change before they can resume their movement through the model. The way in which such Transactions are handled can have a substantial effect on the model's execution efficiency.

When first introduced to GPSS/H, modelers learn about two lists or "Chains" of waiting Transactions. Both lists are kept in a defined order.

Transactions waiting until a specific time are listed on the Future Events Chain. Transactions waiting for a busy resource or waiting for a specific state change wait on the Current Events Chain. The Current Events Chain is so named because all Transactions listed on it are trying to move forward through the model at the current simulated time.

The Future Events Chain (FEC) should be easy to understand. A Transaction that does not need to try to move now can be tucked away on this list and safely ignored until its "move time" has come. The list is maintained in order of each Transaction's move time. Because this is the order in which Transactions are removed from the FEC, very little CPU time is spent managing the FEC. (In very large models, the *insertion* of a Transaction into the FEC can be time-consuming, but GPSS/H is exceptionally good at performing this step.)

The operation of the CEC is more complicated. All Transactions, and only those Transactions, that *would* move "now" if they *could* move now are listed on the CEC. This can degrade performance under heavy queueing in models that rely too heavily on the CEC. The problem is that each Transaction on the CEC that is presently unable to move is waiting for one or more state changes that will allow it to resume its progress. State changes occur with nearly every Block executed by a moving Transaction. When should which Transaction still queued up on the CEC get its chance to move?

In this tutorial, we cannot detail the algorithms that manipulate the GPSS/H CEC. See Schriber (1990) for a detailed discussion. For our purposes here, all we need to know is that each Transaction on the CEC can potentially chew up lots of CPU time due to the the test(s) that must be performed on its behalf "can this Transaction now move ahead?") during processing of the CEC.

Consider a Transaction that needs to wait for a queue level to change. In a complicated model, a single Transaction waiting even a short time for such a condition to be met might need to be tested hundreds or even thousands of times. Some tests might get performed over and over in a single instant of simulated time.

If hundreds of Transactions need to be tested hundreds of times each on hundreds of different occasions during a single run, then many million tests must be performed. Obviously, much depends on fast execution of necessary tests, and avoidance of unnecessary tests. Ways to accomplish these goals will be discussed below.

2.2.1 Shorten the CEC by Eliminating Transactions

One obvious step is to reduce the number of Transactions that are queued up. Perhaps many Transactions can be processed together via an early ASSEMBLE Block or a delayed SPLIT. This type of solution will always work, but is not always available.

2.2.2 Speed Up the Test

Speeding up the test is the principal mechanism by which GPSS/H maintains its performance in the face of all this testing of CEC Transactions. GPSS/H divides all CEC Transactions into two categories. One category is for Transactions that must be fully tested each time they are examined. Such Transactions are said to be "scan-active."

The other group, Transactions which are "scan-inactive," consists of Transactions for which only certain well-defined state changes ("unique blocking conditions") will allow movement. For scan-inactive Transactions, only a perfunctory test is performed during CEC processing.

Whether a Transaction is scan-inactive is represented by a single flag bit in the Transaction's private data. The flag is called the "Scan Skip Indicator." It can be examined in just a few microseconds on most of today's computers, including fast PCs.

If the flag is off, the entire test to allow movement is performed each time. If the flag is on, the Transaction is essentially ignored because the test is already known to be false. Whenever a unique blocking condition is removed, GPSS/H automatically turns off the flag for ONLY those Transactions waiting for that particular condition.

In GPSS/H models, the scan-skip mechanism should be used wherever possible. It isn't necessary to know the details of this mechanism. We can provide simple rules:

1. NEVER use a TEST Block in "refusal mode." (Always use the label of an "escape Block" as the C-operand.) Except as described below, use GATE Blocks instead, even if this means adding Logic Switches or other devices to the model.

2. Avoid refusal-mode GATE Blocks with auxiliary operands of FS, FNS, M, and NM. (GATE M and GATE NM are rarely used anyway.) These GATE Blocks work just like refusal-mode TEST Blocks. All other GATE Blocks automatically use the Scan Skip Indicator and are much more efficient.

FS ("Facility Seizable") and FNS ("Not Seizable") are new in GPSS/H Release 2.0. For FS to be true, the Facility must be (1) Not Seized, (2) Available, and (3) Not Preempted. Unless a model uses FUNAVAIL, FNU ("Facility Not in Use") or FU can be used instead. If the model does use FUNAVAIL, and performance is an issue, one can use GATE FNU, followed by GATE FS and TRANSFER SIM (see CHECKPOINT, Puzzle and Solution, Spring and Fall 1988).

That's it! The rules are simple because the Scan Skip Indicator is automatically used in conjunction with SEIZE and ENTER, the most common Blocks used to create queueing.

Unfortunately, the scan-skip mechanism is less able to increase efficiency when a model has one or more sizable groups of Transactions waiting for a *single* blocking condition. Upon each occurrence of the state change that removes the blocking condition, *all* the waiting Transactions will have their Scan Skip Indicators turned off, even though only *one* of them will be able to move. As soon as the first Transaction moves, the blocking condition recurs. The rest of the Transactions then have their full movement tests performed, only to fail and have their Scan Skip Indicators turned on again. When one or a few queues dominate the CEC in this fashion during model execution, efficiency can be restored quite easily by employing User Chains, as discussed below.

*2.2.3 Add User Chains to Shorten the CEC **and** Avoid the Test*

User Chains are a critical element in successful GPSS/H modeling of complex systems. User Chains provide dedicated ordered lists of waiting Transactions. (Remember that a GPSS/H "Queue" is NOT a data structure – it is only a data-gathering device and has no bearing on Transaction movement.) Contrary to popular belief, User Chains are quite simple to understand and use in most cases.

Once a Transaction has been removed from the CEC and placed on a User Chain, it is examined *only* when the modeler so directs. Moreover, it remains on the chain until the modeler explicitly removes it. This can result in much greater efficiency than leaving the Transaction on the CEC – the Transaction never "beats its head against the wall" as it would at a refusal-mode TEST Block, or even at a SEIZE block in the presence of a very large waiting line.

Some blocking conditions do not fit a GATE, SEIZE, or ENTER Block. For example, what if the system is waiting for an inventory level to change *and* a vehicle to become available to pick an item up? Making the two tests with sequential Blocks is usually a bad idea, because in GPSS/H the Transaction could end up waiting at the second Block beyond the time at which the first condition fails again. (Sometimes sequential testing can be made to work via a TRANSFER SIM Block, but one must still avoid refusal-mode TEST Blocks.)

We present below a new approach to learning User Chains. By teaching only the unconditional mode of the LINK Block, this approach ignores the infamous "Link Indicator" that sometimes confuses people learning User Chains. It turns out that the Link Indicator offers nothing that can't be done with an unconditional Link.

First, we consider the basic SEIZE/RELEASE Block pair, depicted below. This discussion will apply identically to ENTER/LEAVE and to other types of blocking conditions. Note that we do not discuss QUEUE and DEPART beyond mentioning here that QUEUE and DEPART have no bearing on the execution flow of a GPSS/H model. They are present only to gather statistics.

```
QUEUE       SERVQ
SEIZE       SERVER          Steps 1, 2, 3, and 4
DEPART      SERVQ
ADVANCE     PL(SERVTIME)
RELEASE     SERVER          Steps 5, 6, and 7
```

The SEIZE Block is a complex Block that actually performs four different functions:

(1) First, it checks to see if the server can be SEIZEd.
(2) If the server cannot be SEIZEd, then SEIZE will intelligently "deny entry."
    (a) Denying entry means leaving the active transaction on the CEC and suspending its movement temporarily.
    (b) "Intelligently" means the SEIZE turns on the Scan Skip Indicator of the Transaction being suspended.
(3) If the server can be SEIZEd, then the server's own capacity must be updated, accomplishing the actual "seize."
(4) Statistics relating to the server must be altered to track the change of state in the utilization integral.

When the service has been completed and the Transaction is ready to give up the Facility, a RELEASE Block performs three additional steps:

(5) Update the utilization integral and any other statistics.
(6) Reset the state of the Facility itself (decrement the contents, in the case of a Storage)
(7) Inform the next waiting Transaction of the state change.

In order to understand how to employ User Chains to increase execution efficiency, one must think of explicitly modeling the four SEIZE functions and the three RELEASE functions. As we discuss how this is done, the reader should make reference to the code fragment below. (Remember that this technique is useful for an arbitrary waiting condition, not just for SEIZEing a Facility.)

```
        QUEUE       SERVQ
        GATE FNS    SERVER,GO4IT        Step 1
        LINK        SERVERCH,FIFO       Step 2
*
GO4IT   SEIZE       SERVER              Steps 3 and 4  This
*                                       SEIZE will never deny
*                                       entry
        DEPART      SERVQ
SVC     ADVANCE     PL(SERVTIME)
        RELEASE     SERVER              Steps 5 and 6
        UNLINK      SERVERCH,1,GO4IT    Step 7   Free first
*                                       chain member to
*                                       compete
```

The LINK and UNLINK Blocks can be used to manage queueing for *any blocking* condition, as long as there exists a *unique unblocking* location in the model where an appropriate UNLINK Block can be placed.

A model can have any number of User Chains, each referenced by name or number. The name or number of the User Chain is the A-operand for the LINK and UNLINK Blocks. For LINK, the modeler needs to supply an ordering rule as the B-operand, the simplest of which are LIFO and FIFO. Typically, FIFO is used, because the UNLINK Block unlinks in FIFO order by default.

UNLINK requires an expression that is the maximum number of Transactions that can be UNLINKed as the B-operand, and a Block name or number to which to route the UNLINKed Transaction(s) as its C-operand. That's all there is to it.

We shall now employ User Chains to model the seven functions in the preceding example. First, one must perform an explicit check of the condition to see if it is met. For a SEIZE Block, (1) should be a conditional-mode GATE FNS ("execute the next Block if the Facility is Not Seizable, otherwise take the branch"). If the condition is not met, then one must explicitly deny entry. This means placing the Transaction on a User Chain by having it execute an unconditional LINK Block. This is equivalent to (2a) above, but the Transaction is now permanently in a "deep freeze" until another Transaction frees it. Because the Transaction's status is no longer subject to being implicitly checked, step (2b) is not necessary. Steps (3) and (4) can now be accomplished via a SEIZE Block. Note that this Block will never deny entry, because the SEIZEability of the Facility has already been confirmed.

When GPSS/H executes a RELEASE Block, it accomplishes step (7) by turning OFF the Scan Skip Indicator of all Transactions that are already on the CEC waiting for that Facility. A flag is also set to cause a CEC re-scan after the RELEASEing Transaction eventually comes to rest. This allows the highest ranking of the waiting Transactions (the one nearest the front end of the CEC) to succeed in SEIZEing the Facility.

When the modeler employs a User Chain to manage waiting Transactions, the RELEASEing Transaction must accomplish step (7) explicitly. This is done by executing an UNLINK Block, which will explicitly remove one Transaction from its "deep freeze" on the User Chain and place it into contention for the Facility.

Note that User Chains provide basic waiting statistics, including average contents and maximum contents. In the code fragment above, these statistics will be identical for the Queue and for the User Chain. The QUEUE and DEPART Blocks could be eliminated in the User Chain example.

The User Chain is also handling all aspects of contention for the facility, reducing the SEIZE and RELEASE Blocks to a purely statistics-gathering function (steps 4 and 5). If you don't need server utilization statistics, and no other Blocks use the same Facility, then steps 3 and 6 are unnecessary. One could thus take out the SEIZE and RELEASE, substituting `TEST G  W(GO4IT),0,GO4IT` for the GATE Block. The current contents of the ADVANCE Block accurately states the server status (idle or busy); if it's 0, then the server is free. The resulting streamlined model fragment is shown below.

```
        GATE FNS    SERVER,GO4IT        Step 1
        LINK        SERVERCH,FIFO       Step 2
*
GO4IT   ADVANCE     PL(SERVTIME)
        UNLINK      SERVERCH,1,GO4IT    Step 7   Free first
*                                       chain member to
*                                       compete for server
```

3. ACHIEVING MODELER EFFICIENCY

3.1 Use a 386 or 486 PC, with Interactive Debugging

For many modelers, making simulation runs on a shared central computer has been a major cause of inefficiency. Long waits for simulation output can't always be filled with other productive work. In addition, the use of interactive debugging tools often has been discouraged on central computers, out of a mistaken belief that interactive tools consumed unreasonable amounts of CPU time.

Unfortunately, a modeler has not always had the option of moving simulations off of a central machine and onto a dedicated PC because of model size and run-time requirements. Many models were simply too large to run within the DOS 640K memory limit, and using EMS or OS/2 to get around the memory limit required that the modeler pay a substantial speed penalty.

In early 1990, however, a new version of GPSS/H was introduced which overcomes both of these difficulties. GPSS/H 386 uses DOS-extender technology on 80386 and 80486 PCs to provide the large memory access ***and the speed*** needed for big simulation models. Furthermore, since GPSS/H 386 is fully compatible with other versions of GPSS/H, even users who still need the extraordinary execution speed of a mainframe (300,000+ Blocks per second) for long-running models can do their development and debugging on a PC with a substantial gain in convenience and cost-effectiveness.

The GPSS/H Interactive Debugger is central to rapid model development, verification, and modification. Several simple commands are provided by the debugger for controlling a model's execution and examining its status. Among the more frequently used commands are:

STEP	Execute 1 Block
STEP *n*	Execute *n* Blocks
DISPLAY *xxx*	Display statistics on one or more entities or SNAs
BREAKPOINT *yyy*	Set a Block Breakpoint (stop when any Transaction reaches Block *yyy*)
AT *yyy*	Set a Block Breakpoint with an attached Debugger procedure
CONTINUE	Execute to completion, or to the next Breakpoint
CONTINUE *yyy*	Execute until Block *yyy* is about to be executed by any Transaction
TRAP XACT *n*	Stop when Transaction *n* tries to move
TRAP CLOCK *n*	Stop when the Clock reaches or exceeds *n*
CHECKPOINT	Save the complete state of the model
RESTORE	Return to the CHECKPOINTed state
QQ	Quit quickly (exit immediately to the operating system)

The Debugger can be invoked at the beginning of a run, or (except for runs in batch) by interrupting a long-running model to be sure all is OK before continuing. Usually the Debugger is invoked at the beginning of a run. In fact, its execution-speed penalty is so small (less than 5 percent) that many modelers use it exclusively as their runtime environment for GPSS/H.

The GPSS/H Debugger also supports a "windowing" mode on many of the machines and operating systems on which it runs. The windowing mode, known as TV *(test video)*, displays source code and model status information as the model is run.

3.2 Automate Experiments with the Control Statement Language

Getting good results from simulations requires multiple runs and careful statistical housekeeping. For example, consider simulating a hypothetical assembly line. One statistic of interest will be the average number of widgets produced per hour, and since the simulation results will probably be used to make management decisions, it is highly desirable to specify confidence limits on the statistics where possible.

A simulationist might approach this problem by first developing a model of the assembly line that records the widgets-per-hour produced for each hour of a 40-hour work week, then calculates the average widgets-per-hour value for each week. With only this basic model, but using GPSS/H Control Statements such as DO and ENDDO, the simulationist could straightforwardly apply the technique of batch means, running the model automatically for a multiple-week period, and collecting data on the average value of widgets-per-hour for each week. The weekly means of widgets-per-hour will be normally distributed (thanks to the Central Limit Theorem), even though the hour-by-hour production figures are not. Having a normally distributed statistic greatly eases the construction of the needed confidence intervals.

The principal statements that support general purpose programmability in the GPSS/H Control Statement language are:

DO ENDDO	DO-loop iteration

IF
ELSEIF
ELSE
ENDIF
	IF-THEN-ELSE conditional branching

GOTO HERE	Unconditional branching Dummy branch target ("CONTINUE")
INITIAL LET	Assignment of values to data items
GETLIST GETSTRING	Input (list-directed read) Input (unformatted string read)
PUTPIC PUTSTRING	Output (picture-directed formatted write) Output (unformatted string write)
CALL	Call an external (user-supplied) routine

3.3 Let a Smart Random Number Generator Help

A problem for modelers, particularly after a model is largely complete and the modeler is concentrating on validation and the running of experiments, is the need to provide multiple *independent* streams of random numbers for use in different parts of the model (or in the same parts for different runs.). Simulation languages ordinarily allow their users to alter the output of random number generators, but typically have no convenient way to ensure that the specified changes produce *independent, non-overlapping* streams of numbers. A modeler seeking to ensure independence is often required, with some simulation tools, to gain a detailed understanding of both the algorithm and implementation used in a particular generator. Few users have the time or inclination to gain such an understanding, and rightfully so. The end result too often has been that modelers were unable to exercise intelligent control over a critical part of their experiments.

GPSS/H uses an ***indexed Lehmer*** random number generator, designed and implemented with special attention to the problem just discussed. This *indexed* implementation of Lehmer's multiplicative congruential algorithm, with parameters selected on the basis of work by Fishman and Moore, means that modelers can simply and straightforwardly control any number of random number streams in a model and easily guarantee that they will be independent (that they will not be autocorrelated due to overlap). GPSS/H also provides automatic detection of any overlap that might accidentally occur during a run, providing an extra measure of protection to users.The generator behaves as follows:

> The "next" random sample is produced by applying the algorithm to the most recent sample. In other words, the *n*th sample is produced purely as a function of the *n-1*st sample.
>
> The 32-bit value from which the very first sample is produced is known as the *seed* value.
>
> The stream of samples produced repeats after a fixed number of samples. This number is called the *period* of the generator. In the case of the Lehmer algorithm, the period is 2**31-2, so a supply of over two billion unique random numbers is available.
>
> Values supplied by the user (as operands of the RMULT statement or BRMULT Block) are interpreted as offsets from the starting point of the generator. Thus, an RMULT operand value of 50000 means "start with the 50,000th number in the period of the generator." A different starting offset can be supplied for each independent stream used in a model, so that the streams "tap in" at different points in the period of the generator.
>
> Because the generator's true seed value is known, the *n*th sample can be calculated from *n* itself. GPSS/H takes advantage of this *very* special property so that a modeler can specify that a stream start with the one-millionth value in its period without GPSS/H having to generate and discard the first 999,999 values.

GPSS/H's indexed Lehmer implementation thus lets the modeler provide independent streams simply by knowing approximately how many samples will be needed for each stream in the model. To get two streams, with approximately 300,000 samples to be drawn from each, the simulationist might specify RMULT operands of "100000,500000". The second stream would start 400,000 samples downstream from the first, leaving a comfortable margin before overlap would occur. To aid in the determination of appropriate offset values, GPSS/H provides standard output for each stream used in a model, showing its starting offset, ending position, and number of samples drawn during a run.

Note that the RMULT statement is part of the GPSS/H Control Statement language, and allows random number streams to be controlled very flexibly outside of the model proper, as part of the run controls defining an experiment. GPSS/H also provides the ability to save the state of random number streams in a model, so that they can be restarted from the same point in a subsequent run.

3.4 Use Macros

GPSS/H offers a powerful *macro* capability. Macros are similar to subroutines in what they do and how they are used. However, the way they work is completely different. Instead of being compiled once and then called many times from wherever they are used, macros are *expanded* into statements at each point where they are used in the model. This happens *before* GPSS/H generates the executable code, and means that redundant code is generated each time a Macro is invoked. This costs memory and a little compilation time, but when used in moderation has a number of rewards.

The most important reasons for using Macros are the same as those for using subroutines. The main benefit is central maintenance of redundant code. That is, one can make a single global change in the way something is coded without having to risk errors and waste time by making changes throughout a model. Macros and subroutines provide other benefits by packing more functionality into less source code. This saves typing, enhances readability, and reduces the possibility of stray errors.

Why would a modeler want to use Macros instead of subroutines? There are at least two good reasons. First of all, Macros improve performance by avoiding the overhead of passing arguments, transferring to a subroutine, and returning results. In GPSS/H, Block subroutines can be somewhat clumsy to code, whether via TRANSFER SBR or by "rolling your own." Also, modeling and execution time can be consumed in parameterizing Transactions or setting global variables, either of which is like passing arguments and results as described above.

The other reason for using Macros instead of subroutines is unique to GPSS/H: each use of a Macro yields uniquely numbered Blocks. It is even possible to have these Blocks uniquely labeled. Experienced GPSS/H simulationists, who often rely on Current and Total Block counts when debugging or using a model, should appreciate this.

Some systems lend themselves more readily to being modeled with the help of GPSS/H Macros than others. Macros are especially helpful in modeling transfer lines and other manufacturing systems, where the need to model many similar "stations" often occurs.

GPSS/H Macro syntax is summarized as follows.

1. Define a Macro using a STARTMACRO Compiler Directive with the name of the Macro as the statement label.

2. Compose the macro, using #A, #B, etc., wherever you want the '#x' argument replaced (in literal text fashion) by the corresponding Macro operand prior to compilation. Note that you can put '#x' replacement anywhere – even embedded in the text of any symbol, including Block labels. (An exception is any comment line that begins with a '*', where no substitution occurs.) A Macro can consist of either Blocks or Control Statements (or both, which would be a rare case).

3. Use the ENDMACRO Compiler Directive at the end of the Macro definition.

4. Invoke the Macro as needed by specifying the name of the Macro as the statement label, the word MACRO as the statement name, and the operands for that particular invocation as the operands of the MACRO statement. You can have up to ten operands (#A through #I).

You can even have nested Macros, as long as a Macro doesn't invoke *itself* (directly or indirectly). For more details about Macros, refer to the Spring 1988 issue of CHECKPOINT, or to Chapter 3 of the GPSS/H Reference Manual.

3.5 Build Data-Driven Models

GPSS/H contains a very flexible, easy-to-use input capability. This makes it possible to build data-driven models without the need to write programs in FORTRAN or other languages outside the simulation tool.

Input data that are used for run control, or as part of the experiment specification for a run, can be read in from files via the GETLIST Statement before or after the model proper executes each time. Data that are a standard part of the model itself can be read in during model execution via the BGETLIST Block.

GETLIST and BGETLIST read files in which data values are separated by blanks, and can handle integer, double precision floating point, and character data. Both also allow user-specified actions to be taken for error and end-of-file conditions.

With forethought and (B)GETLIST, the modeler can place any amount of specific model and/or run-control data into external files. This is the same as the "experimental frame" notion employed in some other simulation tools, except that it is much more flexible.

Because GPSS/H also has built-in, general-purpose *output* statements (PUTPIC and BPUTPIC), a data-driven model can easily be made interactive. (B)GETLIST and (B)PUTPIC can be used for reading from and writing to devices such as terminals, as well as files, and character-type Ampervariables facilitate both reading and writing text. Consequently, it is a relatively straightforward task to have a model, or its experimental control program, or both, run with interactive control. When properly designed, such interactive models can be used by someone whowho knows nothing about simulation or programming.

4. SUMMARY

Efficient modeling, in terms of both model execution time and model implementation time, is important to simulationists. To achieve this goal, a modeler must choose a suitable simulation tool *and* use it wisely. In this tutorial, we have shown how modelers can use some of the powerful performance and ease-of-use features of GPSS/H to achieve the efficiency they need.

REFERENCES

Brunner, D.T., Ed. (1988), *CHECKPOINT 4*, 2 (Spring 1988), Wolverine Software Corporation, Annandale, VA.

Brunner, D.T., Ed. (1988), *CHECKPOINT 4*, 3 (Fall 1988), Wolverine Software Corporation, Annandale, VA.

Crain, R.C., D.T. Brunner, and J.O. Henriksen (1987), "Advanced Features of GPSS/H," In *Proceedings of the 1987 Winter Simulation Conference*, A. Thesen, H. Grant, and W.D. Kelton, Eds. 269-275.

Crain, R.C. and D.T. Brunner (1988), "New Advanced Features of GPSS/H," In *Proceedings of the 1988 Winter Simulation Conference*, M. Abrams, P. Haigh, and J. Comfort, Eds. 146-151.

Crain, R.C. and D.T. Brunner (1989), "Extended Features of GPSS/H," In *Proceedings of the 1989 Winter Simulation Conference*, E.A. McNair, K.J. Musselman, and P. Heidelberger, Eds. 249-253.

Fishman, G.S. and L.S. Moore III (1986), "An Exhaustive Analysis of Multiplicative Congruential Random Number Generators with Modulus 2**31-1," *SIAM Journal on Scientific and Statistical Computing 7*, 1, 24-45.

Henriksen, J.O. and R.C. Crain (1989), *GPSS/H Reference Manual*, Third Edition, Wolverine Software Corporation, Annandale, VA.

Schriber, T.J. (1990), *An Introduction to Simulation Using GPSS/H*, John Wiley & Sons, New York, NY.

Wightman, L.B., Ed. (1990), *CHECKPOINT 6*, 3 (Summer 1990), Wolverine Software Corporation, Annandale, VA.