A FLOWCHART-TO-SOURCE-CODE
CONVERSION PROGRAM FOR GPSS/360

by

Robert Edward Purvy

December 1973

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

UIUCDCS-R-73-613

A FLOWCHART-TO-SOURCE-CODE
CONVERSION PROGRAM FOR GPSS/360*

by

Robert Edward Purvy

NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

December 1973

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

* Supported in part by the Atomic Energy Commission under contract US AEC AT(11-1)2383 and submitted in partial fulfillment of the requirements of the Graduate College for the degree of Master of Science in Computer Science.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

fy

Acknowledgement

I would like to thank my advisor, Professor C. W. Gear for his initial advice and suggestions, Mr. Al Whaley for similar help and also frequent debugging assistance, and Mrs. Debra Sweet for her typing of this thesis. Thanks also go to the Department of Computer Science and the Atomic Energy Commission for supporting this work, and to everyone who helped me get the program working.

Contents

	page
Acknowledgement	
I. Introduction.....	1
II. Basic Concepts.....	2
A. GRASS.....	3
B. GPSS.....	5
C. Flowchart Drawing.....	6
D. Multiple Branches.....	13
E. Linking Pages Together.....	17
III. Using the Flowchart Program	22
A. Details on How to Draw Flowcharts.....	23
B. Constructing Mnemonics from Scratch.....	27
C. Sending Pictures to the 360.....	28
D. Operation of the Program	28
E. Tabbing.....	29
F. Options.....	30
G. Output from GPSS.....	31
H. Error Messages.....	32
IV. Suggested Improvements.....	35

Introduction

A system for specifying a program for IBM's discrete simulation package, GPSS, as a flowchart has been developed, using the graphical capabilities of GRASS, or Graphical Remote Access Support System. The system takes the data structures constructed by GRASS and creates an equivalent source program in GPSS assembly format. It uses the communication routines described in (3) to provide interaction with the user, allowing him to specify what is to be done with the completed source program, and to submit the program to GPSS. Output from GPSS may be routed to the user terminal, where he may select the portions for which hard copy is desired.

Following is an introduction to the GRASS system, which may be skipped by readers already familiar with it. Some comments on the nature of flowcharts and GPSS, and how GPSS is well adapted to specification via a block diagram follow, after which there are details on how to draw flowcharts and how to use the system. This paper describes a working system; the section on Suggested Improvements suggests some ways the system could be extended.

II . Basic Concepts

GRASS

This section is intended only as a general review of GRASS, since it is extensively documented. The reader is referred to (1,2) for further details.

The general purpose of GRASS is to allow the user to construct models of a system and analyze them. Elements can be constructed which are models of the individual components of the system, and the element definitions can then be used to build up a network graphically. The elements can have terminals, which are used to join the elements and specify the interaction between them.

As defined in (1), an element consists of the set of data:

- a) Graphical mnemonic
- b) Terminal set
- c) Equations
- d) Local Variables
- e) Global Variables
- f) Parameters
- g) Name

The data items are designed to allow arbitrary networks to be designed. The equations, for example, might be used in the case of an electrical component to specify the effect of the element on voltage and current. The terminal set is a set of points on the graphical mnemonic that represent the terminals; these terminals can have different types. In modeling an electrical network the types might be electrical, control, etc. Variables and parameters are fairly intuitive.

These data items are more than sufficient for drawing flowcharts; in fact, some of them are not needed. We have no need of any equations,

local variables, or global variables, and the reader can skip over the sections in the papers referenced that deal with these. The graphical mnemonic, terminal set, and parameters, plus the name for reference purposes, are all that is needed. Thus there is a fairly natural correspondence between the sort of flowcharts most people draw and the sort demanded by the program described by this paper. There will be more on this later.

GPSS

GPSS is a language designed to facilitate simulation of discrete systems. The unit of work is a transaction; and it might represent customers moving through a bank, jobs through a computer system, cars through a repair facility, or almost anything which is a discretized unit. (The reader is referred to (5) and (6) for a full discussion of GPSS, from which this section is drawn.)

As facilities move through a system they act upon the entities of the system: "facilities", "storages", and "logic switches". A facility handles only one transaction at a time; it might represent the central processor of a computer, a car wash, etc. Storages can handle a variable number of transactions; examples would be core storage of a computer, or a parking lot. A logic switch is an on/off indicator which can be set and reset by a transaction to govern the flow of transactions. A traffic light, or the working/not working flag of an input/output device could be modeled by a logic switch.

GPSS provides statistical entities to gather data about the behavior of the systems. Queues are used to list transactions delayed at any point in the system, and record the average number of transactions delayed and average length of the delay. Tables are more general and can gather statistics on almost any aspect of the system.

All of these entities are controlled by blocks. The blocks are what actually appears in a GPSS source program; they control the creation and destruction of transactions, and their behavior. Blocks are what we

will be concerned about in the drawing of flowcharts. GPSS has 44 block types available to the user. In general, these blocks are at a very high level; a very complex system can often be simulated with less than one hundred statements. (This includes a number of lines of code other than blocks; these would not appear in the flowchart, and the way they can be inserted by the user is explained below.)

Flowchart Drawing

Block diagrams, or flowcharts, are familiar to almost everyone who has ever used a computer. They have a number of advantages over a source listing in documenting a program: they are graphical, and the human mind usually finds it easier to understand a picture of something than a number of lines of text; they are often used as an aid in writing the final source program, and as such are an intermediate step in the writing of the program. When used in the latter fashion, the flowchart may be very general. Statements like "check whether magnetic tape unit is on" may appear; these leave open the question of how one is to check the status of the unit: where the information is located, how it is coded, etc. On the other hand, the flowchart may be very close to the final source program, right down to the text within the boxes being the same as the text that appears in the program. Which type of flowchart a programmer will draw will be determined in part by the level of the language he plans to use. If he knows he will be writing in assembler language he is unlikely to have a one-to-one relationship between boxes on the flowchart and lines of code in the program. More likely he will

indicate roughly what he plans to do and leave the coding for later. If the ultimate language is to be FORTRAN, on the other hand, at least some of the statements, particularly the arithmetic ones, will probably be identical to their final form. It is this type of flowchart that must be drawn for the flowchart program being described here. In the nomenclature used in GPSS reference manuals, the user must construct a block diagram. Except for linkage blocks used for connecting one page of the block diagram to another, (described below) there is a one-to-one relationship between blocks in the block diagram and blocks in the output GPSS program. The information included with each block is, with a few important exceptions, the same as the information that appears in the output. (The exceptions have to do with the labels of statements, and their appearance in the operand lists of other statements.)

The typical block diagram is the basic model for the input demanded by this program. There are, however, some features of normal block diagrams that have to be changed for GRASS. Normally, the flow of control in a flowchart is indicated by arrows leading out of one box and into another. Thus there is a great deal of freedom in how control flows into and out of a box. There may be an arrow leading into it from the bottom and another leading out the top. The basic problem with this when using the GRASS system is that there are no arrows connecting terminals together, only lines, which are non-directed. Thus some conventions must be established as to how a block may be entered and exited. The first and most important convention is this: No block can be exited from

the top nor entered from the bottom.

This is a fairly natural rule and one that can be complied with without too much difficulty. It preserves the usual top-to-bottom flow of most block diagrams.

The order of "compilation" or the way that the blocks in a block diagram are transformed into a source program is an important concept and one that should be thoroughly understood by anyone wishing to use the flowchart program. One very basic rule, related to the rule mentioned above, is this: all blocks connected in a straight top-to-bottom way will appear in the same order, adjacent to one another in the final output, with no blocks intervening that are not in the top-bottom chain. This may perhaps seem like an obvious rule, but figure 1 shows that it is not. (This example is taken from (4)). The lines of code beneath the block diagram show how it was transformed into source code. We see that statement TELLER follows directly after the TRANSFER statement in the diagram, but not in the source code. This is not the way the flowchart program works. To get the code of figure 1, the block diagram of figure 2 would have to be used. Note two things: one, the obvious, that PENS is now directly under the TRANSFER, and two, the label TELLR need not be present in the TRANSFER's operands. Instead, the name of the terminal on the TRANSFER that branches to TELLR, i.e. RIGHT, is used. (It is also a feature of GPSS that PENS wouldn't have to be present, either, only a null operand, i.e. .900,, RIGHT. The GPSS assembly program substitutes the number of the following block, in this case, PENS.)

INPUT	GENERATE	3,2
	ENTER	BANK,1
WHICH	TRANSFER	.900,PENS,TELLR
PENS	ADVANCE	2, 1
TELLR	ADVANCE	4,3
	LEAVE	BANK,1
	TERMINATE	1

Figure 1

The flowchart program, seeing the branch, first looks at the operands of TRANSFER to see if the terminal type, RIGHT is there. Finding it, it replaces it with TELLR. The operands of TELLR are also scanned for a LEFT. Now, in this case, the statement branched to had a label already assigned, TELLR. This would not be necessary, either, using the flowchart program. If no label was present on the ASSIGN, one would be generated, (of the form ZZYnn or ZZZnn, where n is a digit) put on the ASSIGN, and substituted for RIGHT in the TRANSFER's operands.

This example illustrates the basic operation of the flowchart program. Since all transactions in GPSS originate in a GENERATE block, the program looks for one of these, follows the top-bottom connections as far as they go, and then starts in on the branch connections. Now consider figure 3, the same as figure 2 except that TELLR is connected to a GENERATE on top. Since there is a top connection, we know that TELLR must immediately follow whatever is up there. So, when the branch from TRANSFER is processed, RIGHT will still be replaced with TELLR but TELLR will not be compiled immediately. Instead it will be left for the time being, until the block on top of it has been compiled. Thus, the code for figure 3 will be:

	GENERATE	3,2
	ENTER	BANK,1
	TRANSFER	.900,PENS,TELLR
PENS	ADVANCE	2,1
	GENERATE	1,1
TELLR	ADVANCE	4,3
	LEAVE	BANK,1
	TERMINATE	1

(Never mind that this makes no sense as GPSS input).

So the basic rule, necessitated by the difficulty of globally analyzing a multi page block diagram is the top-bottom rule. Since

Figure 2

Figure 3

drawing a block diagram is not a substitute for thoroughly learning the syntax of GPSS, a user must have at least some knowledge of how certain blocks, particularly TEST and GATE, work, to use the system properly.

Finally, let's look again at figure 1 and see how it could be modified for the flowchart program. The basic problem here is the arrow from the PENS block to the TELLER block. ADVANCE is not a block capable of branching to another block, so this block diagram needs an intervening TRANSFER, directly under PENS and branching to TELLER, to make it compatible with the flowchart program as it works now. (See Suggested Improvements for a description of how this can be modified, and probably will have been by the time this is printed.)

The preceding listings of the source programs output by the flowchart program have not specified the columns that each field begins at, but have instead implied that they will conform to the GPSS assembly requirements, and this is, in fact, the case. The user is asked to specify separately the label, block type, operands, and comments (how this is done is explained in "Using the Flowchart Program"), and the program puts them in columns 2, 8, 19, and 36. (The comments go in column 36, that is, if that will leave at least one blank between the operands and the comments; if not the comments are put one space after the operands.)

Multiple Branches

Quite often in drawing a block diagram, one would like to have several blocks referencing a single block. The following description is taken from (5):

"... lines which join the blocks indicate the flow of traffic through the system or describe a sequence of events to be performed. Alternate courses of action or decision which arise in the system are represented by having more than one line leave a given block. Conversely, one block may have several lines entering it to represent that this block is a common step in two or more sequences of events."

There are two cases here: a multiple branch out of a block, and a multiple branch into a block. Figure 4 illustrates this. The ASSIGN block is referenced by TST1 and TST2. TST2, in turn, is referenced by both TRANSFER blocks. Now, in this case we have simply two chains of top-bottom links, so the compiled code will look like this:

	GENERATE	3,1
ZZY01	ASSIGN	1+,1
	TRANSFER	.9,TST2
	TRANSFER	.8,TST2
	GENERATE	4,2
TST1	TEST GE	C1,K50,ZZY01
TST2	TEST L	N1,N2,ZZY01

A label was generated and put on the ASSIGN block, since it had none, and this label substituted for LEFT1 in both TEST blocks. Note here that in general, a block capable of branching, like TEST, TRANSFER, GATE, etc. needs two terminals on each side, one for branches in and one for branches out. A block which can have two statement labels in its operands, like TRANSFER, can use terminals on both sides, so we might have TRANSFER .9,LEFT1,RIGHT1.

In this example, all blocks were connected in a top-bottom chain to one or the other GENERATE block, so that the order of compilation was completely determined. Now consider figure 5. In this case, the two TEST blocks are connected only to each other and the TRANSFER block, so

Figure 4

Figure 5

that the only way the TEST blocks could be compiled is by the order of compilation proceeding through the TRANSFER block. This will not happen. Recall the distinction between a multiple branch out of a block, and into a block. The branch out of the TRANSFER is ambiguous, since RIGHT could be replaced with TST1 or TST2. Therefore, in this case it would be ignored and neither TST1 nor TST2 would be compiled. As a general rule: every branch out of a block must have its own terminal.

Branch connections have two effects on the output program:

- (1) they affect the order of compilation, and
- (2) they cause the operands of the connected blocks to be scanned for the terminal types involved, and if these are found, they are replaced with the label of the other block. (operand resolution)

Multiple branches have only the second effect. They are only useful for incoming branches; i.e. having a number of other blocks all referring to one block. The same sort of technique is available for off-page references; which brings us to the subject of external terminals.

Linking Pages Together

Since any block diagram of a non-trivial nature cannot legibly be made to fit on a screen, some technique is necessary for splitting a diagram up into sections. Two possibilities present themselves. One is to use the mnemonic facility of GRASS to take a part of the flowchart, create a mnemonic, smaller in size, to stand for it, and creating external terminals on the mnemonic to link to the other "pages". The other is to create certain blocks for off-page references. They have

a parameter attached to them, and any other such block with the identical parameter is considered to be linked to it. The first approach would have certain advantages; for a modular-type program: a mnemonic for each module could lead to a very clear picture of how the different segments of the program are related. But GPSS is not in general the sort of language where one creates subroutines and builds up a program from them.

Furthermore, learning to use mnemonics and external terminals in GRASS is currently a difficult task. It places a great burden on the user, and the final product would look completely unlike the original block diagram. For these reasons, it was felt to be more natural to link the pages together with special blocks, called XTERM's.

Originally when the flowchart program was written, only one type of XTERM was defined. This was analogous to the original rule of only one connection per terminal; only one EXTERM could be linked to a given EXTERM. Multiple connections were then introduced, and this necessitated defining two new XTERM's, the Receiver XTERM (RXTERM) and Sender XTERM (SXTERM). To see why, note that on figure 4, there is no question of TST1 and TST2 being connected via the LEFT1 terminals, even though each is connected to the ASSIGN there. But if the connection were via XTERM's, there would be confusion. Consequently, the way this would be implemented for off-page references would be to attach an RXTERM to the ASSIGN, give it a parameter, say 'AA', and then attach SXTERM's to TST1 and TST2, also with 'AA'. Now there is no mistake as to what is

connected to what. The rules for XTERM's are similar to those for internal terminals, and one is: the order of compilation is not affected by RXTERM's. Even if only one SXTERM is connected to an RXTERM, the flow of control does not go out the RXTERM.

Figure 6 should clarify some of this. Note that this is a two-page block diagram, that TRA2 is linked to TRA3 via the EXTERM's with parameter 'AC', TRA1 and TRA2 both branch to SXTERM's with parameter 'AB', and that the RXTERM that they link to is connected to the top of TERM on page 2. This block diagram will produce the code:

	GENERATE	3,2
TST1	TEST GE	N1,N2,RIGHT
TRA1	TRANSFER	.75 TERM
TRA2	TRANSFER	.80,TERM
TRA3	TRANSFER	,TST1
TERM	TERMINATE	1

The TST1 block did not have 'RIGHT' replaced with 'TRA3' even though there was only one block connected to it via the RXTERM. This is because an RXTERM is always considered to be a multiple branch, even though it may not necessarily be. In this case, TRA3 is connected to TRA2 via the EXTERM's but if it were not, if, say, TRA3 had no top connection, it would never get compiled at all.

One other point about figure 6 is worth mentioning. TERM has its RXTERM connected on top, even though technically it is part of a branch, and so one might expect the RXTERM to be on the left or right. This is allowed merely to add flexibility to the system. TERM had nothing connected to it on top, so there is no reason not to put

(page 1)

(page 2)

Figure 6

the RXTERM there. The effect is the same as if it had been on the left or right.

This concludes the general explanation of how the flowchart program works. The next sections explain in detail how to use the system.

III Using the Flowchart Program

Details on how to Draw Flowcharts

The reader is again referred to (1) and (2) for details about using GRASS.

As mentioned earlier, GPSS has 44 blocks available to the user. Page 44 -52 of (6) gives a means of drawing block diagrams where there is one drawing for each block type. This is obviously a lot of shapes to learn, and it is not the scheme used here. The one we use has five basic shapes, plus the circular one for XTERM's, and is described on p. 43 of the same manual. When loading GRASS onto the PDP-8, (see (2)) there is an option for the user to provide his own library. The library will already be available to any prospective user, and should have all the blocks drawn and ready to use.

Theoretically, we could get by with just these five shapes, plus the one circular one, but this would create more work for the user; he would have to add comment text to each block as he brought it up, and then add a parameter explaining what the block is, which brings up the subject of altering parameters.

In GRASS, only a mnemonic can appear as an instance in another picture. Mnemonics can have parameters associated with them. When drawing a flowchart, what one is actually doing is constructing a series of pictures out of mnemonics. In the menu area, there might be something like this:

```

ADVANC
ADVANC.Ø
ASSIGN
ASSIGN.Ø
ASSIGN.1

```

There has to be a picture before a mnemonic can be constructed to represent it, so ADVANC and ASSIGN are there, but are never referenced by the user. (The mnemonics are the names ending in a period and a number.) If the user wants to add an ASSIGN block to his picture, he "hits" the point on the screen where he wants it to go, and then "hits" ASSIGN.0 or ASSIGN.1. (There may be more than one mnemonic for a given picture; either may be used. The difference could be merely the size or orientation of the mnemonic.) The graphical representation of the mnemonic hit will then appear on the screen.

In the examples given so far, all the blocks had their pertinent information (operands, block type, label) listed either in the block or on the upper left. These items are all implemented via mnemonic parameters in GRASS, and unfortunately in the current system these are not normally visible. The ASSIGN.0 mnemonic discussed above will have 'ASSIGN' somewhere inside it, but this is comment text and is purely for display purposes. Each mnemonic when constructed (see below for how to draw a block type if one does not exist on the library) has four parameters associated with it: LAB, BLOCK, OPS, and COM, for the label, block type, operands, and comments. XTERM's have a single parameter: PARM. When bringing up a block, the user must hit the "alter assigned mnemonic parameters" function at the bottom of the screen, and then hit the instance. For ASSIGN.0, something like this will appear:

```
LAB = DEFLT
BLOCK = ASSIGN
OPS
COM
```

The LAB = DEFLT parameter is preassigned so that if no label is assigned by the user, 'DEFLT' will appear up until the final processing of the output, (See Suggested Improvements for a note on this.)

Now if the user wants to label this statement 'ASN1', and to give it the operands '1+, 1' he uses the text editor (entered automatically by the 'alter parameters' function) to do this:

- 1) delete the 'LAB = DEFLT' line and insert:

```
'LAB = ASN1'
```

- 2) use the 'change' function to get 'OPS = 1+, 1'.

the screen should now read:

```
LAB = ASN1
BLOCK = ASSIGN
OPS = 1+, 1
COM
```

'COM' could be extended in a fashion similar to 'OPS' if comments were desired. Now if no label were to be assigned, all the user would have to do would be to extend the OPS line. It is important that the parameters appear in exactly this order, and in the form

<type> = <parms>.

If an XTERM is being added, there is only one parameter to alter, PARM, and it is done just like OPS above. For omitted parameters, e.g. COM when no comments are added, no equal sign should appear, i.e. 'COM=' would be an error.

In general, the columns where each parameter goes are as described earlier, i.e. 2, 8, 19, and 36, for label, block, operands, and comments, respectively. Some GPSS block types make an exception to this, namely

TEST, LOGIC, and GATE. For these blocks, one additional operand is generally needed, in column 13 for TEST and GATE, and column 14 for LOGIC. Rather than complicate the parameter structure unnecessarily, this extra operand is added on to the block type. Note that if the word TEST or GATE appears in column 8, and something else begins in column 13, we have one blank separating them and similarly, for the word LOGIC in column 8 and another operand in column 14. Therefore, the rule for these three blocks is, add the extra operand exactly one space after the block type, so that for a TEST GE block, the block parameter would be 'BLOCK = TEST GGE'. On figure 6, page 1, the parameters for TST1 would be:

```
LAB = TST1
BLOCK = TEST bGE
OPS = N1, N2, RIGHT
COM
```

The flowchart program moves the entire string 'TEST GE' into column 8, which puts the 'GE' in column 13, as required.

Now only one other thing remains to be done with the block just put up, and that is to connect it to the desired other blocks. This can be done via either the 'visible connect' or 'invisible connect' functions. The choice is mainly for purposes of visual clarity; 'visible connect' draws a straight line between the terminals, regardless of what is in between. If this is undesirable, one can use 'invisible connect' and then use the line drawing option to draw a line representing the connection.

When one page of the block diagram is complete, the user should give it a name and save it, by typing 'S#name' where <name> has up to six

alphanumeric characters. The screen can be cleared by typing '##name' where <name> is anything that is not the name of a picture in the library. Now the next page can be drawn.

Constructing mnemonics from scratch.

If some block type needed is not present on the library provided, or, worse yet, no library is provided, then the picture and mnemonic must be drawn from scratch. Again, the user should consult (2) for details, but the general idea is this: (1) draw the shape of the block. Preferably it should be very small, because it is desirable to fit as much as possible on the screen. (2) add terminals, one on top, one on bottom, and two on each side. Make all terminals 'external'. (This is a GRASS convention having nothing to do with XTERM's.) (3) create the terminal type list. Unfortunately they must be done anew for each picture even though the types are all the same. 'TOP' and 'BOTTOM' must be two of the types, and the others can be whatever is desired, although 'LEFT', 'RIGHT', etc. seem good choices. One possibility would be to just use 'L' and 'R' instead of 'LEFT' and 'RIGHT'; this may even have been done in the library by the time this is printed. (4) assign the terminal types to the terminals. Obviously care should be taken that the right types are assigned, since otherwise, the operands might not get resolved. (5) add comment text, one word indicating what block this is. (6) now, give the picture a name by typing 'S#name' (7) type in 'C&' for 'create mnemonic'. This puts the user in the 'assign mnemonic parameters' mode. Now type in 'LAE=DEFLT', 'return', 'BLOCK=block', 'return', 'OPS', 'return', and 'COM', 'return'. Hit the return area in the lower left corner

of the screen (to be distinguished from hitting the return key). (8)
 Finally, save the mnemonic by typing 'S&name .n' where n is a digit from 0 to 7, and (name) is extended with blanks, if necessary, to be exactly six characters long.

Sending pictures to the 360

All pictures must be entered on the graphics filing system on the 360 before they can be used by the flowchart program. The storage, retrieval, and manipulation of data structures is done by the library maintenance program LSD (see (3)). LSD is invoked from the PDP-8 remote communications program REACT (see (2)). After the user logs on to the remote computer via REACT he starts LSD by typing !S LSD, and then, using the commands described in (3) he saves the pictures on the filing system. The command !HALT will stop LSD, and, if desired, the flowchart program can then be started by typing !S GPSSGRSS, and the pictures just saved by LSD can then be referenced.

Operation of the program

As mentioned above, when the user is logged on to the 360, he can start the program by typing !S GPSSGRSS. When the message 'GPSS READY' appears it is ready for input. The only input currently allowed is 'COMPILE pic1,...,picn' where <pic i> is the name that the ith picture was stored under. Except for the first picture, they can be in any order. The reason the first picture is important is that all GENERATE blocks must be in the first picture. Since the pictures are all linked together by XTERMS's the others can be in any order.

If the flowchart is error-free (see Error Messages) the message 'TYPE IN BEGINNING CARDS?' appears. This refers to the fact mentioned earlier that not all the statements in a program will be blocks and

consequently not all will appear in the flowchart. Function definitions, entity definitions (such as storages and variables) and tables all have to be typed in by the user. In some cases they may appear either before or after the main body of the program.

The user can type either 'YES' 'NO' or just start typing in lines at this point. 'NO' might be desired if some warnings had appeared (see Error Messages) that led the user to believe that the program contains errors, or if for any reason he didn't want to add statements just now. If the answer is 'NO' the question 'PUNCH DECK?' appears; if this is answered 'YES' the program that was created is punched onto cards. This option is available under all courses of action, so it can still be done later if statements are added.

Tabbing

Several aids are provided for the adding of statements to the program. The most important is tabbing. Since GPSS is a fixed field language, only two sets of tab stops are available: the assembly format, (columns 8, 19, 36, and 72) and the function follower card format (columns 7, 13, 19, 25, etc.) The assembly format is the default. Now, since GRASS sends a whole line at a time, the tab characters will appear in the line as it is being typed, and they will disappear as the line is expanded to the proper tab stops. The line will be echoed back to the user with tab characters removed; this is because it is possible to make a mistake and put in too many tabs, too few, put them in the wrong place, etc. If this happens, type in '#DE' and the last line will be deleted. An example will illustrate: let's say the user types '%START%%50'

('%' is the tab character). The system echoes back:

```
col 8          col 36
START          50
```

but the user really wanted the 50 in column 19. (The two tab characters put it in the third tab stop.) He types '#DE' and then '%START%50'. Now he gets:

```
col 8          col 19
START          50
```

which is what he wanted. Only tabbed lines can be deleted.

Options

The '#' in '#DE' is a signal that the line being typed is not a text line. There are other options, among them the 'change tab stops' option. '#A' institutes the assembly format set of stops, and '#F' sets up the function follower card format.

Another option that could be very useful is to allow some sort of abbreviation for commonly used cards or sets of cards. This has been done so far for two of these: the SIMULATE card and the four card negative exponential distribution. Typing '#SI' gives 'SIMULATE' and '#EX' gives:

```
1,1 |.1,.104 |.2,.222 |.3,.355 |.4,.509 |.5,.69
.6,.915 |.7,1.2 |.75,1.38 |.8,1.6 |.84,1.83 |.88,2.12
.9,2.3 |.92,2.52 |.94,2.81 |.95,2.99 |.96,3.2 |.97,3.5
.98,3.9 |.99,4.6 |.995,5.3 |.998,6.2 |.999,7.0 |.9997,8.0
```

saving a great deal of tedious and error-prone typing. Clearly this could be very easily extended to have a whole library of functions available to the user. The exponential is one of the most common, and one that the author always got wrong at least twice every time he tried it.

Each line of text entered by the user and not deleted on the next line is copied into a temporary data set. When he is through with the beginning cards, he should type 'END' and the main body of the program is then copied. Then the message 'TYPE IN ENDING CARDS' appears. At this point all other entity definitions plus control cards (e.g. START, REGET) and output editor cards should be typed in. The ending cards are terminated with 'END' again. GPSS also requires an END card, except it is in column 8, and one will be inserted upon typing 'END' in column 1 if none was typed. Thus, typing:

col 8

END

END and just:

END

have the same effect.

Now there is a complete GPSS program on disk, and if desired it can be submitted as input to GPSS. The question 'RUN GPSS?' will appear, and again, type 'YES' or 'NO' (or any words beginning with 'Y' or 'N'). If 'NO' the question 'PUNCH DECK?' will appear and the answer again is 'YES' or 'NO'. In either case the program will terminate after executing the command.

Output from GPSS

If GPSS is run, the output will be sent to the user's terminal, a frame at a time, where he can choose the portions for which he wants hard copy.

The first important fact about getting output this way is that the screen is smaller in both directions than a sheet of computer paper

(About 85 characters wide and 33 lines long). The second is that the way output is routed to the terminal is completely blind to what the output is. (It's done by finding GPSS's output DCB and substituting our routine's address for the PUT routine's.) This is important to the user for the following reason: for special output, that which uses GPSS' Output Editor, one would obviously like all of the output on one frame, and not have part of a graph on one frame and the other part on the next. This can be done, but the burden is necessarily on the user, for reason number 2 above. It is done as follows: First, any graph, table, etc. should be less than 33 lines long. Second, line length should be carefully planned so that no line is longer than 80 characters. Third, an EJECT card should immediately precede the Output Editor cards producing the output. Whenever our routine sees a '1' carriage control (page skip) it sends whatever it has and starts a new frame. The rule about 80 character lines is necessary because any line sent by GPSS longer than 80 characters (with trailing blanks removed) is split up into two lines. This could destroy the usefulness of a graph or table.

When a frame appears, the system is waiting for a user response. Typing 'P' will print what's on the screen (except that long lines are not split into two lines), and typing anything else will cause only a row of asterisks to appear on the paper, to mark the missing output.

Error Messages

In general, all the pictures named by the user are completely scanned for syntax errors, so one run should turn up most, if not all,

of the errors in a set of pictures. The following is a list of error messages and their meanings.

- 1) 'PICTURE <name> NOT FOUND' The picture given by <name> is not present on PICLIB.
- 2) 'INVALID COMMAND' The line typed in, after 'GPSS READY' appeared, did not begin 'COMPILE'.
- 3) 'ERROR: INST # n_1 PICTURE <name> CONNECTED TO ITSELF' where n_1 is the number of the number of the incorrect subpicture. This means that one terminal of the block is connected to another terminal of the same block, or to itself.
- 4) 'ERROR: TWO OR MORE LOCAL CONNECTIONS TO AN XTERM: PIC # n_1 INST # n_2 ' where n_1 is the number of the offending picture, in the order given in the COMPILE command. Each XTERM may be connected to only one other block within the picture.
- 5) 'ERROR: UNRESOLVED XTERMS' This means that there were XTERM's whose parameters did not match those of any other XTERM. This message is followed by a list of the unmatched XTERMS's.
- 6) 'ERROR: NO PARM PRESENT FOR XTERM, INST # n_1 , PICTURE # n_2 ' An XTERM did not have a parameter of the form 'PARM = xx'
- 7) 'ERROR: NO GENERATE BLOCK FOUND IN FIRST PICTURE' The first picture (again, using the order given in the COMPILE command) must have at least one GENERATE block to start the compilation process.
- 8) 'ERROR: RXTERM CONNECTED ON TOP, INST # n_1 , PIC # n_2 ' Having an RXTERM connect to the bottom of something could violate the rule that

the flow of control never goes into a block from the bottom.

- 9) 'NO SUCH OPTION' This refers to typing '# xx' when inputting statements. The '#' signals an option, e.g. change tab stops, and this means there is no option matching the one typed in.
- 10) 'A SIMPLE YES OR NO WILL SUFFICE' This means the system asked the user a yes-no question and he answered with neither 'yes' nor 'no'.
- 11) 'SOURCE DECK NOT CREATED DUE TO FLOWCHART ERRORS' Any of the messages beginning 'ERROR' sets a flag preventing the attempt to output a source deck from taking place.
- 12) 'WARNING: NO OPERAND TERM TYPE, PIC # n_1 , INST # n_2 TO PIC # n_3
INST # n_4 '

This means there was a branch connection between the two instances and neither one had in its operands the type of the terminal involved in the branch. It will generally be an error, resulting from putting the wrong terminal type in the operands, but it is allowed because it provides more flexibility. If one instance has no bottom connection, and the second no top connection, then a branch connecting them has the same effect as a top-bottom link. This allows block diagrams in some cases to conform more closely to the natural ways of drawing them.

Suggested Improvements

There are a number of possible ways of making the system easier and more convenient to use. One of these concerns the specification of parameters. The current rules are very rigid; they require that the parameters be in order according to their ultimate appearance on the card, and, in the case of the label parameter, the default is always 'LAB = DEFLT'. To change this, the user must delete the line and then type in what he wants. This is currently being worked on. The new parameter routine will allow the parameters to be in any order, it will check for keywords like 'LAB =', 'BLOCK =', etc. (the current system assumes they are there, and in the right place), and for the label it will simply have 'LAB', and the user will just extend it if desired.

Another inconvenient feature of the present system is the COMPILE command. If there are many pictures in the flowchart, typing out all their names can be very tedious, and conceivably they might not even fit on one line. The new routine will encourage the user to name his pictures with numbers, e.g. PIC1, CH3, and then allow him to use dashes to indicate the range of pictures. An example will make this clearer. Suppose the flowchart consists of the pictures CH1, CH2, CH3, CH4, FLC1, FLC2, BP1, BP2, and BP3. Whereas formerly the user would have to type: 'COMPILE CH1, CH2, CH3, CH4, FLC1, FLC2, BP1, BP2, BP3' with the new system he will simply type: 'COMPILE CH1-3, FLC1-2, BP1-3'

One major extension to the program currently being worked on involves the problem of figure 1, explained earlier. Recall that the problem with this flowchart was that the connection from PENS to TELLR implied the existence of a TRANSFER block, which is currently not acceptable. The new rules will allow this sort of implied branch, and

will automatically insert the TRANSFER blocks needed. An implied branch will be considered to be a connection from the bottom of one instance, to a terminal other than TOP on the other instance. Details of this extension are still being worked out.

When a flowchart does contain errors, one would like to provide as much information as possible to the user for debugging. The code to print out, for each picture, every instance in the picture, by name and number, and all its connections to other instances is already available, and all that needs to be done is to allow the user to specify whether he wants to see this information or not. The point about the instance number is relevant because unless one remembers the order in which he put up the instances, it is difficult to determine what number a particular instance is. (It can be done by the terminal numbers, but this would be time consuming since not all mnemonics have the same numbers of terminals.) Most if not all error messages give only the instance number. If the user has the debug information described here and also a hard copy of the picture (one of the graphics terminals currently in use has a hard copy device attached to it) he should have no trouble locating the error.

References

- [1] Gear, C. W., et al., The Simulation and Modeling System: A Snapshot View. Department of Computer Science, file no. 824, University of Illinois, Urbana, Illinois 61801 (February 1970).
- [2] Michel, M. S. and Kooh, J. A., GRACC" Terminal User's Guide. Department of Computer Science, report no. 467, University of Illinois, Urbana, Illinois 61801.
- [3] Michel, M. J., and Haskin, R., GRASS: Extended Remote Facilities Guide. Department of Computer Science, report no. 867, University of Illinois, Urbana, Illinois 61801.
- [4] Maisel, H., and Gnugnoli, G., Simulation of Discrete Stochastic Systems. Science Research Associates, Inc., 1972.
- [5] General Purpose Simualtion System /360 OS Version 2. Introductory User's Manual, IBM Program Product H20-0693-0.
- [6] General Purpose Simulation System /360. User's Manual, IBM, Application Program GH20-0326-4.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-73-613	2.	3. Recipient's Accession No.
4. Title and Subtitle A FLOWCHART-TO-SOURCE-CODE CONVERSION PROGRAM FOR GPSS/360			5. Report Date December 1973
7. Author(s) Robert Edward Purvy			6.
9. Performing Organization Name and Address Department of Computer Science University of Illinois Urbana, Illinois 61801			8. Performing Organization Rept. No.
			10. Project/Task/Work Unit No.
			11. Contract/Grant No. US AEC AT(11-1)2383
12. Sponsoring Organization Name and Address UE AEC Chicago Operations Office 9800 South Cass Avenue Argonne, Illinois			13. Type of Report & Period Covered Thesis Research
			14.
15. Supplementary Notes			
16. Abstracts <p>This paper describes an interactive program which allows a user to specify a GPSS program as a block diagram. The user draws the block diagram using the Graphical Remote Access Support System (GRASS) and sends it to the remote computer where it is converted to an equivalent source program. He can then run the program, if desired, and have the output from GPSS routed back to the graphics terminal, selecting the portion for which hard copy is desired.</p> <p>The program is written in 360 Assembler Language.</p>			
17. Key Words and Document Analysis. 17a. Descriptors <p>graphics flowcharting GPSS</p>			
17b. Identifiers/Open-Ended Terms			
17c. COSATI Field/Group			
18. Availability Statement unlimited		19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 40
		20. Security Class (This Page) UNCLASSIFIED	22. Price