SIMCOM—The Simulator Compiler*

THOMAS G. SANBORN†

IN MANY present-day activities involving the use of digital computers, the need often arises to run programs on a computer other than the one for which they are written. For example, the computer on which a program is intended to be run may exist only as a proposed design, or it may be in some stage of construction, or it may simply be at a remote location[1]. One solution to the problem posed by such a situation is to prepare a program for an available computer which, in effect, transforms the available computer into the unavailable computer. Such a transformation program is called a *computer simulation program*, since it gives one computer the ability to simulate another.

Because of the intricate logical relationships which prevail in computers, the preparation of a simulation program is time consuming and fraught with opportunity for error. Furthermore, changes in the specifications of the computer being simulated may necessitate a major overhaul of the simulation program. For these reasons a new programming language and its associated compiler, SIMCOM (standing for *Sim*ulator *Com*piler), are being developed to assist in the preparation and modification of simulation programs which are to be run on the IBM 709.

It must be clearly understood that SIMCOM is *not*, itself, a simulation program. It is a generating program which accepts statements written in a specialized simulation-oriented source language, and from these statements generates instructions in SCAT language similar to those a human programmer would write in preparing a simulation program.

The fundamental unit of SIMCOM coding is the *statement*. Each statement is either a definition of a component of the simulated computer or a description of some data manipulation or control function which occurs during the execution of instructions within the simulated computer. These two kinds of statements are known, respectively, as *definition statements*, and *procedural statements*. Related statements are grouped into *paragraphs*. SCAT coding, including SCAT-type remarks, may be intermixed with the paragraphs should the SIMCOM language prove inadequate for describing some involved procedure. The characters which may be used to write statements include the upper-case Roman letters, the decimal digits, and certain special characters. Combinations of alpha-numeric characters are called *symbols*. The three uses of symbols are: 1) to represent components of the simulated computer; 2) to identify locations within the simulation program; and 3) to denote integers. Every symbol, unless it represents an integer, must contain at least one alphabetic character, and no symbol may be identical to a word of the basic SIMCOM vocabulary.

A simulation program written in the SIMCOM language consists of three parts: the "Machine Definition;" the "Instruction Interpretation;" and the "Panel Operation." These sections describe, respectively, the static machine, the machine in operation, and the effect of operator intervention.

The Machine Definition is given in six paragraphs labeled "REGISTERS," "MEMORY," "INPUT," "OUTPUT," "KEYS," and "INDICATORS." Each definition statement describes a machine component or cell, giving its name, bit structure, and, if appropriate, its address or range of addresses. For coding convenience, a register may be defined as being synonymous with part or all of another register. Furthermore, registers can be defined which have no counterpart in the real computer being simulated. No distinction is made by SIMCOM between so-called primary and secondary storage.

LOCAT. 1 6	8	TEXT 16 72
		REGISTERS. MR(S,1-35). AC(S,Q,P,1-35).
	PCR(35-0).	UAK(14-0) SYN PCR(29-15).
		MEMØRY. CØRE(S,1-35) 0-4095.
	DRUM(35-0)	16384-32767.
		INPUT. CARD(S,1-35).
		ØUTPUT. PRNTR(S,1-35).
		KEYS. RESET. LØAD. START. MJ1. MJ2. MS1. MS2.
		INDICATØRS. ØVFLØW. IØCK. DVCK. EØTA. RUN.
	TRAP.	

2992

Fig. 1—Simcom coding form showing typical definition paragraphs.

Fig. 1 shows examples of some typical definitions selected from several well-known computers. This figure also illustrates the basic requirements of the form on which SIMCOM coding is to be written. Some users may wish to use a form on which each

* Presently being developed under a purchase order from Thompson Ramo Wooldridge, Inc., in support of their contract to supply technical direction to the Automatic Data Processing Facility, U. S. Army Electronic Proving Ground, Fort Huachuca, Arizona.

† Space Technology Laboratories, Inc.

[1] For a description of the applications of SIMCOM anticipated by USAEPG see: A. B. Crawford, "Automatic Data Processing in the Tactical Field Army", Proceedings of the Western Joint Computer Conference, pp. 187–189; San Francisco, March 1959.

From the collection of the Computer History Museum (www.computerhistory.org)

card column is marked since blank positions are frequently essential in the language. Note that the first line of each paragraph is indented to column 16 and that subsequent lines begin in column 8. The compiler uses this convention to aid in distinguishing between SIMCOM statements and SCAT instructions which may be included in the source code.

In all of the machine component definitions, the symbols used to identify the various devices are quite arbitrary, the only limitations being that they conform to the previously stated rules pertaining to symbols, and that they do not conflict with the basic vocabulary. The most common method of selecting symbols will undoubtedly be to adopt those used by the machine manufacturer in his manual since these are usually highly mnemonic.

The Instruction Interpretation and Panel Operation sections of the simulation program are written in terms of procedural statements. A procedural statement consists of a *primary operation* together with one or more operands called *expressions*, arranged to form a stylized sentence. Each primary operation is denoted by one or more words from the basic SIMCOM vocabulary. This vocabulary includes words for transferring, clearing, complementing, testing, comparing, and shifting arrays of bits in the various components of the simulated computer, plus words which control the logical flow of the simulation program and the compilation process itself.

The expressions upon which the primary operation act consist of symbols combined by means of secondary operations. These secondary operations include + (add), − (subtract), * (multiply) $ (indirect address), and certain words of the basic vocabulary which denote logical arithmetic and scaling. A symbol in an expression may have bit designators appended to it if only part of the component identified by the symbol is to participate in the operation.

Fig. 2 shows a few paragraphs of procedural statements. In the figure the expressions are underscored for emphasis but this would not be the normal practice on a coding form. Note that the first paragarph bears a location symbol. In many instances, however, cross references between paragraphs are implicit in their relationship to one another. Therefore, many paragraphs will need no location symbols attached to them.

The instruction interpretation section is the heart of the source program. It will normally contain statements which describe the procurement of instructions from the simulated computer's storage, followed by statements which describe the effects of each instruction. The various instruction interpretations can usually best be initiated by use of a "table look-up" statement. Depending on the complexities of the instructions and the associated timing, each instruction description may require as little as one simple statement or as many as several paragraphs including, perhaps, entries to subroutines and additional table look-ups. Fig. 2 illustrates a simple example of this technique.

LOCAT. 1 6	8	TEXT 16 72
LØØP		IC $ TØ IR. IC + 1 TØ IC. LØØK UP IR(1-5) IN
	ØPER TABLE.	CLØCK 4. EXECUTE LØØP.
ØPER		TABLE.
		0. IR(6-17) TØ IC. TURN RUN ØFF.
		1. IR(6-17) TØ IC.
		2. IF ØVFLØW IS ØN, TURN ØVFLØW ØFF.
	IR(6-17)	TØ IC.
		5. EXECUTE GET. EXECUTE SUBTRA.
		CLØSE.

Fig. 2—Typical procedural paragraphs showing instruction procurement and interpretation technique. "Expressions" underscored for emphasis.

A "Table," as understood by SIMCOM, is an ordered set of paragraphs of procedural statements, each paragraph being identified by an integer. The table look-up operation provides a means for selecting one of these paragraphs for execution, depending on the value of the argument expression. If a paragraph in a table does not terminate with an explicit transfer of control to some other point in the simulation program, then control returns to the statement following the "LOOK UP . . ." statement which invoked the paragraph. Thus each paragraph in a table is like a closed subroutine.

The panel operation section of the simulation program includes an interrogation of the status of each console key and a description, written in SIMCOM statements, of the behavior of the simulated computer if the key has been activated.

It is not uncommon for certain keys on computer consoles to be so constructed that they are turned off as soon as the function which they perform has been initiated. The programmer's statements must include this action, if appropriate. Furthermore, in some cases certain keys are inoperative unless other keys or indicators are in a particular status. The programmer must also provide this logic.

One of the most interesting features of the system is the subroutine library. Subroutines are stored in the library in the SIMCOM language, except that the symbols denoting the subroutine parameters are replaced by variable symbols of a special kind. At compilation time, as a subroutine is called from the library, its special variable symbols are replaced by the parameter symbols given in the library call statement, and its location symbols are replaced by arbitrary unique symbols. The subroutine is then inserted into the source code where the SIMCOM decoder and instruction generator processes

From the collection of the Computer History Museum (www.computerhistory.org)

it in the same manner as any other set of SIMCOM statements. This process is illustrated in Fig. 3.

```
THE SUBROUTINE IS ORIGINALLY CODED AS
                SUBRØUTINE ADD, A, C.
        A + B TØ C. CLØSE.

THE LIBRARY MAINTENANCE ROUTINE WILL PLACE THE SUBROUTINE IN
        THE LIBRARY IN THE FORM
ADD             V1 +B TØ V2. CLØSE.

AT A SUBSEQUENT COMPILATION, A STATEMENT OF THE FORM
                LIBRARY ADD, P, Q.

WILL CAUSE THE SUBROUTINE TO BE INCORPORATED INTO THE PROGRAM AS
ADD             P + B TØ Q. CLØSE.
```

Fig. 3—Sample subroutine showing generalized variable technique.

A given subroutine may be called from the library any number of times during one compilation and, depending on the parameters listed in the library call statement, each version may give rise to a different number of 709 instructions. Each version of a subroutine called from the library is a "closed" routine which can be executed from any point in the simulation program.

The output from SIMCOM is a translation into SCAT language of the source program. This includes a direct expansion of the procedural statements, plus certain pseudo operations for assigning storage and certain utility routines whose necessity is only implied by the source language. These include routines for loading the simulated computer, diagnostic output routines and, of fundamental importance, a routine which allocates the simulated computer's storage to the various 709 storage media. This storage management routine must partition oversize words, should such have been defined, into the 36-bit words of the 709, and shuttle simulated computer storage to and from 709 tape units if it exceeds the capacity of the 709 core storage. The endowment of the compiler with the ability to generate efficient storage management routines is the most challenging problem facing the creators of SIMCOM.

Fig. 4 is a schematic representation showing the allocation of the generated program to the various parts of the 709. The heavily outlined areas indicate the parts of the 709 used to represent the various registers and storage of the simulated computer. The remainder of the 709 contains the generated simulation program and its associated utility routines. The arrows indicate the communication paths between the various areas of the 709.

Because the SIMCOM output is in SCAT language, the compiler need not contain within itself an assembly program, nor does it have to be able to process the SCAT instructions included in the input code other than to recognize them as SCAT instructions. Most important, however, is the fact that generation from SIMCOM statements to SCAT instructions can be done during a single pass through the source program. In addition to generating SCAT language instructions, the compiler transforms each line of SIMCOM coding into a SCAT-type remark (* in column 1) and inserts each paragraph into the generated code immediately ahead of its SCAT language expansion. Thus each paragraph serves as commentary to describe the function of the generated SCAT instructions which follow.

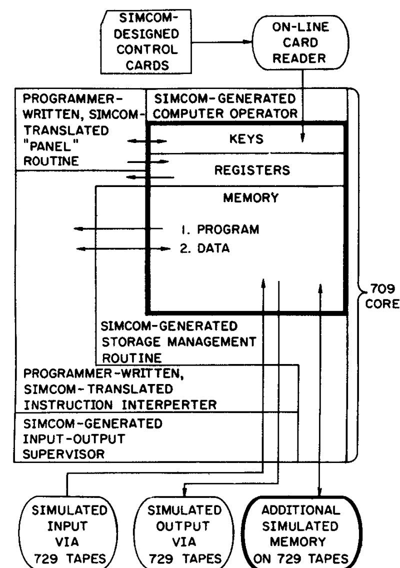

Fig. 4—Allocation of elements of a Simcom-prepared simulation program to 709 core and tapes.

The SIMCOM language is such that apparently minor modifications to the input statements can completely alter the character of the generated program. For example, a change in the definition of a register of the simulated computer may cause SIMCOM to generate instructions to do multiple precision arithmetic where single precision arithmetic was formerly sufficient, or a change in word size in the simulated computer may cause SIMCOM to reorganize completely the simulated computer's storage in the 709 core. Thus changes which could be made to a machine-like language simulation program only by completely rewriting the program can be incorporated into a SIMCOM-written simulation program by a simple re-compilation.

The SIMCOM system will provide a means whereby users who are not necessarily professional programmers may prepare simulation programs for binary computers in a language not unlike that used by computer manufacturers in their manuals. There seems to be no escaping the fact that the user will need to be more than casually familiar with the computer to be simulated before he can write an adequate simulation program, even with SIMCOM.

From the collection of the Computer History Museum (www.computerhistory.org)

Discussion

P. Armer: Is your work far enough along so you can comment on two things with respect to timing? One, let us say with a simulated machine not too different from the 709 that didn't have to do with double precision arithmetic. Could you give us some notions on the efficiency time lapse?

Mr. Sanborn: Since we have not actually constructed any simulation programs yet, I couldn't give you an answer from experience, but we are optimistic that the simulation programs generated by SIMCOM will be relatively efficient as compared to handwritten codes. I am not sure it would be too efficient for a machine similar to the 709, because it could not capitalize on the similarities of the machines.

Mr. Armer: In simulating a machine one would be interested in how fast the program would run. Have you anything to solve this problem?

Mr. Sanborn: There are language statements for keeping track of time; the clock statement we saw was one. One can, in writing down the description of the instructions, also include information as to the execution time — as a matter of fact, keep a running total of the elapsed time in the computer. This may become particularly important where you have independent devices running and may want to keep several clocks running in order to determine which unit is going to run next and keep them in proper synchronization.

G. L. Foster (IBM): Do you assume the machine being simulated has a fixed word length? Will SIMCOM handle variable word lengths as on the 705?

Mr. Sanborn: I think it would if the registers are defined in terms of smallest accessible storage. I question whether it would be practical, but it would be possible.

R. Cornish (IBM): Is this program universal only with respect to the particular family of computers?

Mr. Sanborn: No, I wouldn't say this. We tried to make it general for binary computers with word sizes not greater than 72 bits.

Mr. Cornish: Approximately how many man years were involved?

Mr. Sanborn: I haven't checked the figures recently but I imagine up to this time we have invested two man years.

D. J. Campbell (AGT): How much machine time would be used in a typical compilation?

Mr. Sanborn: I don't think I would want to predict this. I haven't been at all concerned on compiling time. I have been concerned about execution time of the generated program.

R. J. Scott (Dept. of Defense): Could SIMCOM simulate a machine with an automatic interrupt feature as is used on STRETCH?

Mr. Sanborn: I think if a machine has a feature like this, you must write statements in the language which quite frequently enters the panel operation section to test interrupt conditions.

I. Flores (Dunlop Assoc.): In debugging a program for a computer to be simulated, how will the 709 indicate program faults or other errors?

Mr. Sanborn: I am not sure that the 709 will indicate faults in the program of the simulated computer. There are diagnostic facilities in the generated program. For example, statements calling for certain registers to be printed out. This may be diagnostic or it may be results from the program running in the simulated computer.

R. W. Bemer (IBM): Does it simulate only binary machines?

Mr. Sanborn: Only binary machines in the sense that for any other sort of machine one would have to define a representation of the information in the computer in terms of binary digits.

Mr. Bemer: Does the syntax correspond to or use Gorn's microflow chart technique?

Mr. Sanborn: I don't know, because I am not familiar with this technique but I would guess not.

E. B. Shore (Pratt & Whitney): Can SIMCOM be used to compare various computers on a benchmark program? How about multi-sequence computers?

Mr. Sanborn: I am not sure what is meant by benchmark program.

Mr. Armer: Well, anyone can dream up a program and see how various machines do on this same program.

Mr. Sanborn: SIMCOM would provide the means for constructing the simulation program for trying out various computers. It has in itself no mechanism for evaluating these computers on the basis of running the benchmark program.

Mr. Armer: If I understood the question of the clocks, it would tell you how long it would take each machine to run the particular program.

Mr. Sanborn: Yes, the clocks would tell you running time but if you wanted to get more involved information about the relative merits of the computers this is beyond the scope of SIMCOM.

From the collection of the Computer History Museum (www.computerhistory.org)