Extended Control and Simulation Language

By A. T. Clementson*

This paper describes the provision of compilers for an extended version of Control and Simulation Language for the Honeywell 400 and the Honeywell 200 series computers.

C.S.L. had been published (Buxton *et al.*, 1962) and available for two years before Courtaulds decided to redevelop it for their own computers (Courtaulds Ltd., 1964). Up to that time, C.S.L. had only been available on a service bureau basis and was rather expensive to run; Courtaulds started this work because they were anxious to have a complete range of software available at their own installation.

These realizations of C.S.L. were not, like the first, prompted by a particular urgent project so that we could afford to sit back and carefully study the methods to be adopted and the language used. Considerable care was taken to extend the source language to attempt to overcome the limitations of C.S.L. discovered by previous users. This paper will concentrate on these extensions and the compiling methods used, since the original language has already been published in this journal (Buxton *et al.*, 1962). The paper describes two projects simultaneously—the provision of compilers for the Honeywell 400 and the Honeywell 200 series (the latter being sponsored by Honeywell Controls Limited). Each compiler was written and tested by one man working for one year. Many of the extensions which E.C.S.L. allows are catered for, in rather different ways, in C.S.L. 2 (Buxton, 1966) which was produced by IBM during the same period.

C.S.L. was originally compiled into FORTRAN. This was clearly undesirable in this case because the performance of the FORTRAN compilers for these computers was not known at the time. Secondly, many of the limitations on C.S.L. were necessary only because of the restrictions on FORTRAN. But by far the most important reason was the need for an improved and efficient communication between the computer system and the programmer. In particular we wished to make it easy for the programmer to proceed successfully without *any* knowledge of other, more basic, programming languages.

The operating system

We have found that because C.S.L. programs are written by non-programmers, and have relatively short lives, that the number of operator errors is unusually high. With this in mind, and also considering the program writer's unfamiliarity with the computer, a completely automatic system of operation was devised. It is a simple, and far from original, system because this is what the circumstances required.

The operation is controlled by "system control cards" which are easily recognized by an asterisk (*) in card column 1. Columns 7–15 contain the directive to the system. These directives are:

Cols. 7–15	*Cols.* 16–72	*Action*
REMARK		none—remark typed on operator console
DATE	12/03/45	enters date into system
COMPILE	title	expects that a CSL program follows—this is compiled
EXECUTE		executes the last program entered
GET *C*	title	reads in an object program from binary cards
GET *Tn*	title	finds an object program from library on tape *n*
KEEP *C*		punches out binary card deck of object program
KEEP *Tn*		adds or updates program into library on tape *n*
LIBRARY		adds machine code functions to library
POST		produces a POST MORTEM dump of last program executed
WAIT		causes system to stop until operator presses "RUN"
END RUN		terminates run.

The language

Extended C.S.L. assumes that a simulation program is written, or rather presented to the computer, with the following block structure.

BLOCK 1	Definitions
BLOCK 2	Initialization
BLOCK 3 (Headed by ACTIVITIES statement)	Activities
BLOCK 4 (Headed by FINALIZATION statement)	Finalization
BLOCK 5 (Headed by DATA statement)	Fixed Data

The execution of a program first carries out the Initialization (which automatically sets CLOCK to zero) and then enters the *monitor*. The monitor scans the time cells and advances time to the first (or next) event (the smallest positive *T*-cell). Then the activities are

* *Department of Engineering Production, University of Birmingham, Birmingham*, 15.

carried out. At the end of this scan of the activities the program returns to the monitor. This loop continues until CLOCK reaches the value specified in the ACTIVITIES statement; the Finalization is then carried out.

The difference between this structure and the original C.S.L. structure is that Blocks 4 and 5 have been added and the ACTIVITIES statement has been modified to the form

ACTIVITIES ⟨cell-name⟩

where ⟨cell-name⟩ specifies the ultimate value of CLOCK. This increase in the number of sections of the program is both an aid in programming and a help to the compiler. (We are planning to add another block—a block of EVENTS (Laski, 1966)—which will further simplify writing and speed up execution.)

Block 3—the activities—is further divided into sub-blocks each corresponding to one activity. An activity is a section of program concerned with a particular section of the model, (e.g. the activity of loading a lorry). The first statement of each activity is

BEGIN ⟨descriptive comment⟩

The statements within an activity are normally a series of tests followed by some actions. These correspond to the conditions necessary for the commencement of the activity and the action of the activity (e.g. if the lorry is available, has a driver and petrol, then the load can be moved into the lorry). In E.C.S.L. any test is written as a statement which, in the event, will be true or false. If the statement is true, execution proceeds to next statement. If, however, the statement is false, a branch occurs. The destination of this branch may be explicitly mentioned as a statement number but is more usually implicit. On unindented statements (i.e. statements not contained in a loop) a failure will, unless an explicit destination is given, cause control to leave the activity (or sub-block). For an indented statement, however, the destination is implied by the context of the loop containing it. (C.S.L. allowed the user to specify a success branch also but this has been dropped—a GOTO would be used in this context.) Where statement numbers are used, they are local to the activity in which they occur. Thus the only communication between activities is via the data which is, of course, common to the whole program.

Blocks 2 and 4 may also be divided into the sub-blocks if required. In which case the same structure exists. Equally some of the blocks might not be present at all. For example a program consisting only of blocks 1 and 2 is very common; it is an ordinary—i.e. non-simulation,—program written in the language of C.S.L. (Such programs have been found quite successful.)

The definitions

The definitions in C.S.L. are similar to, but more extensive than, those in FORTRAN. Variable names consist of an unlimited string of letters—although only the first six are recognized. All variables are taken to be integers unless otherwise specified. The CLASS statement defines a group of entities (with or without time cells). Extended C.S.L. (in common with C.S.L.2) has discarded the attribute notation of C.S.L. preferring the use of associated arrays (e.g. instead of having one cell attached to each entity which we had to remember contained, say, its size, we now use an array SIZE (see below). For each class there may be one list of sets. Each set is capable of holding, if necessary, the whole class. The entries in a set are restricted to the one class (unlike C.S.L.2, for example) so that an efficient method of execution can be used. Set names can now be subscripted to form a vector of sets, e.g.

CLASS SHIP 60 SET PORT 10

defines a class of 60 ships and 10 sets PORT 1, PORT 2, . . ., PORT 10 (note that the inconvenient full stop between class name and index is now optional because all full stops (except between the two digits) are ignored).

Histograms may also be subscripted in the same way, (see below).

The definition of arrays is now somewhat more flexible. Each dimension may now be specified either as a constant or by a class name e.g. (following above example)

ARRAY CARGO (SHIP, 3)

defines a (60×3) matrix of integers. The advantage of this is that when a program is being used experimentally less changes have to be made. The clarity of the program is also somewhat improved.

Boolean variables and strings are now available, e.g.

BOOLEAN TRUTH 7
STRING TITLE 15

define a vector of 7 Boolean variables and a string of 15 characters.

Sets

The basis of C.S.L. is the logical formulation of complicated problems, and it was decided that most of the extensions would be to assist in this aspect of the work. The framework of any C.S.L. program is the manipulation of sets (or queues). Most of the new features of the language of E.C.S.L. are in fact designed for this purpose. The Honeywell compilers for C.S.L. both extend the language of the sets and also improve the method of internal implementation.

The extensions in the language largely result from allowing set names to be subscripted, thus allowing several queues of similar types to the represented by the same name followed by a subscript. This allows one section of program to manipulate a number of queues at the same time. For example, if we were simulating a shipping system consisting of half-a-dozen ports we

would previously have had to have written one section of program for each port. Now, by using subscription we could do it all in one section.

The set subscription is written in exactly the same way as entity subscripting, i.e.

⟨set name⟩ ⟨. ?⟩ ⟨primary⟩

where

⟨primary⟩ ::= ⟨const⟩ | ⟨cell name⟩ | (⟨expression⟩)

One other minor area of improvement in the language occurs in the listing of actions and tests which involve sets. For example, the statement:

SHIP I FROM PORT 7 INTO ATSEA

would previously have been written as two separate statements. This simple expedient has been found very helpful in preventing the "loss" of entities due to source program errors.

The C.S.L. compiler used a very crude method to represent the sets inside the computer, which was largely tuned to the need to translate into FORTRAN as an intermediate language. As it was decided from the beginning that we would directly translate into machine language, it was now possible, and also obviously very desirable, to improve the way in which sets were manipulated. In accordance with this the method adopted on the Honeywell 200 is in keeping with advanced list processing methods without in any way changing the source language either in expression or in concept.

The set consists of a vector of two character words—one word for each entity in the class. If the entity is not in the set, then the word is zero. If the entity is in the set, then the word contains the name (or subscript number) of the entity which follows it in the queue; unless it is the last member of the queue, when the word contains minus one and has an item mark on it. The item mark on the last member of the set allows this to be found with a single machine instruction. The necessary routines for manipulating these sets are about 20 times faster than the previous methods, and use considerably less storage. It is largely due to this improvement that the compiler is now one of the most efficient compilers on the computer.

Arithmetic

Following C.S.L. the arithmetical statements are virtually the same as those in FORTRAN with the addition of the incremental versions (i.e. ⟨cellname⟩ ± ⟨expn⟩ is used as meaning ⟨cellname⟩ = ⟨cellname⟩ ± ⟨expn⟩). Although floating point is not often used in the simulation work, we have allowed the use of floating point in the language—even to the extent of permitting mixed mode expressions. C.S.L. allowed general expressions to be used as subscripts. Extended C.S.L. permits the use of any arithmetic expression at any point where a numerical value is required, although it must sometimes be in brackets to avoid ambiguity. (For an example see next paragraph.)

Loops and test chains

The loops in E.C.S.L. are controlled in three ways. Firstly the incremental form, similar to the FORTRAN form:

FOR ⟨name⟩ ⟨= ?⟩ ⟨expn$_1$⟩, ⟨expn$_2$⟩ ⟨,expn$_3$?⟩

indicating that the variable ⟨name⟩ (*any* variable having an integer value—even if subscripted) is to take firstly the value of ⟨expn$_1$⟩ then the value of ⟨expn$_1$⟩ + ⟨expn$_3$⟩ (or +1, if ⟨expn$_3$⟩ is missing), etc. until the next value would exceed the value of ⟨expn$_2$⟩. (Note that if ⟨expn$_1$⟩ is greater than ⟨expn$_2$⟩ the loop is not entered at all.)

The second form of loop, originated by C.S.L.1 is:

FOR ⟨name⟩ ⟨= ?⟩ ⟨set name⟩

indicating that the variable name is to go successively through the values of the class subscripts in the set name in the order that they occur in that set. The loop is not entered at all if the set is empty. While this is the same as in C.S.L., its power is very considerably increased by the relaxation of the restrictions on the control variable name and the fact that the set name can now be subscripted.

A third form of loop control has been added to the repertoire in E.C.S.L.

FOR ⟨name⟩ ⟨= ?⟩ ⟨class name⟩

indicating that the variable name takes the values 1, 2, 3, . . . up to the number of entities in the class. The convenience of this method lies in the fact that if the size of the class is to be altered during the development of the program this statement need not be altered. In addition it indicates to the reader what is being done in the loop rather more clearly than did the previous notation.

These three forms of loop control have been illustrated in the FOR statement. However, C.S.L. introduced a number of other forms of loop. All these new forms have been perpetuated in E.C.S.L., but whereas in the original only the set controlled type was permitted, we allow all three forms of loop to be used.

In the original C.S.L., if a FIND loop was contained in a FIND loop, the value of the control variable of the inner loop appropriate to the selected value of the control variable of the outer loop was lost. As the control variable may be subscripted, then this problem disappeared. For example:

```
FIND I = INDEPOT MIN (SIZE (I))
  FIND J(I) = SHIFT K MIN (RATE (J(I)))
    MAXTONAGE (J(I)) GT SIZE (I)
```

This finds the smallest lorry in the depot for which there is a driver whose "maximum tonage" permits him to drive it. If there is more than one such driver available in this shift choose the one whose rate of pay is least. (C.S.L.2 has solved this problem in a more elaborate way which would not be possible on a machine as small as we have available.)

Histograms

In extended C.S.L. histograms serve two purposes. The first obvious use is to record the statistical information produced during the running of the program. This is accomplished by the statement:

ADD expression, histogram

In the original version of C.S.L., histograms which specified empirically a probability distribution for random number sampling were manipulated in a different way from histograms. This distinction was found to be very inconvenient and it was therefore necessary to modify the concept of a distribution. The solution adopted was to make distributions and histograms equivalent—both taking an intermediate form. The random number sampling made it necessary to specify exactly which values would be obtained and therefore it was decided that we would specify the mid-points of the histogram intervals instead of extremes. Thus the ADD instruction illustrated above, means find the interval whose value is nearest to the value of the expression and add one to the frequency of that value. In order to make the use of histograms and distributions more flexible, it was decided that these also should be subscripted. The subscripting takes exactly the same form as in sets. This in fact simplifies the form of the ADD statement and of all other statements referring to histograms.

Input–output

The major shortcoming of C.S.L., as originally conceived, was that it made no addition to the input–output facilities of FORTRAN, despite the fact that the simulation program requires many special facilities. Since we were no longer translating C.S.L. into FORTRAN, we replaced the FORTRAN terminology for input–output of a completely new set of instructions. These new instructions are, however, still very similar to the FORTRAN form. The main difference is that the format specification is included in the I/O statement rather than forming a separate statement. In order to simplify the specification of format it was decided that a standard form would be adopted. Except where otherwise specified this standard format is what FORTRAN would describe as I10 (or F10·5 for real numbers). An I/O statement consists of a key word (Print, Read, Punch, Readtape and Writetape) followed by a list of variable names. This list may be punctuated by commas (which are just separators), slashes (which indicate that a new line is to be commenced), double asterisks (which indicate the beginning of a new page) or a specification of a new standard format, (this standard form is written as *n—where n is a number specifying a field width). This new field width is continued until either a new specification is encountered or the end of a statement is reached.

Variables mentioned in the list can be any C.S.L. variables. Array names without subscripts indicate that the whole array is to be printed. As in FORTRAN, loops may be included within the list, the only difference being that the contents of the loop are enclosed in brackets but the values of the control variable are specified outside the closed brackets. The specification of these values may take any of the three forms indicated above for loops.

The way in which titles and headings are specified is by including them in inverted commas, for example,

PRINT "JOB HEADING"/ . . . . .

The CHECK statement as originated in C.S.L.1 is of course still available in E.C.S.L. being one of the many aids for program development.

Experimental design

An extremely important feature of simulation is that any single run of a program is just one sample of those results that may be obtained in any investigation. It is essential that the program be run many times to produce accurate results. The objective will be to obtain averages which are sufficiently accurate. Many writers have shown ways in which accuracy can be improved by "cheating" methods such as antithetic facilities and control variates which are now in common use. But many simulation systems do not provide facilities to assist in their use. We therefore decided, since the provision of antithetic variables is in machine language terms extremely trivial, that we should provide this in extended C.S.L. Arrangements are made so that if the initial number in any random number stream is negative, the statement will produce multiple antithetic runs and therefore gain a considerable improvement in accuracy. So that the language could allow for the program to be run many times in the form of an experimental statistical design, we decided that we would divide the program into three sections rather than two previously adopted. Originally in C.S.L., initialization was followed by a group of activities and it was the responsibility of the writer of the program to provide the final print-out and to stop the program. However, since the statement labels used are all local to a particular section it was not possible to return to the initialization from any activity. This defect was overcome by providing the section called finalization which is only entered when the clock reaches the predetermined value. This predetermined value is specified in the activities statement. In addition to this two new statements were provided;

RESTART which means go back and re-run the initialization; and FINISH which means at the end of this time-cycle go straight to the finalization. The provision of this finish statement was to allow the early termination of any run in which fatal errors had been encountered, while still allowing for the output data to be produced. If a program is to be run many times it will require the reinitialization of a great deal of data. In particular, the initial state of the simulated system will probably be the same in every run. It is therefore desirable for this data, or at least the part of it which is fixed, to be incorporated in the program inside the computer so that the same data does not have to be read in repeatedly. It was therefore decided to add a new section of program known as the *data sector* in which the initial values of fixed data would be specified. This data follows the finalization. Each card of this section contains, starting in column one, the name of a variable. Following this name on the card will be the value or values of this variable (if the name is an array, for example, one number must be specified for each cell of the array). This data is automatically reloaded at the beginning of the initialization of each run (i.e. at the beginning of the execution and immediately after any restart).

Program development in E.C.S.L. is aided in five ways. First by providing a very fast but efficient compiler, with extensive diagnostic routines. Secondly, the four sense switches are used to produce extra output during execution to assist in debugging the program as follows:

Switch 4 When this is on the descriptive comment following each BEGIN statement is printed as the activity is entered.

Switch 2 When this switch is on the check statements are operative.

Switch 3 When this switch is on a complete dump of all the variables, sets and histograms is printed after each execution of the monitor.

Switch 1 When this switch is on the value of the clock is printed after the monitor is executed.

Thirdly, if requested during compilation, the printing of the descriptive comment at the beginning of each activity is followed by the number of the statement last executed in the previous activity—thus providing a simple trace ability.

Fourthly, many types of error detectable during execution cause the dump to occur automatically, including the activity, statement number and cause of error.

Finally, a post-mortem dump may also be called by system card or manually by the operator should an error actually stop the computer.

Compiling method

The original version of C.S.L. was compiled from C.S.L. into FORTRAN on IBM 1401 producing a tape suitable for input to the FORTRAN compiler on the 7090. This method was originally adopted so that a compiler could be written quickly. Now that the language had been proved to be a success it was obviously necessary to write further compilers which were not so roundabout in their methods. On the Honeywell 400 computer Courtaulds produced a compiler for C.S.L. into the machine assembly language. The assembly program was incorporated within the compiler so that the assembly proceeded automatically, after compilation. It was found that this process was rather inefficient since the assembly part of the program took about twenty times as long as the compilation. It is also apparent that much of the work done by the compiler is unnecessarily duplicated by the assembly program. The assembly is now simply an extra part of the compilation. For example, the machine language addresses which are allocated to data are easily calculated during the first part of compilation as this simply involves counting the number of variables. The major inefficiency of separating compilation and assembly comes in the considerable volume of information which has to be put on to and read from the magnetic tape, for storage between these stages. It was found, for example, that the average C.S.L. instruction produces about thirty machine language instructions, so that the magnetic tape holding the symbolic version of the program was rather long. It is thought that the excessive assembly time was largely due to this. The main problem in combining compilation and assembly is that the allocation of branch addresses cannot be made immediately. However, a special system was devised to overcome this problem which did not involve an extra pass of the data. The actual translation of C.S.L. statements into machine language, and the production of a binary version of the program, results in considerable savings in time. Further savings come from the fact that the "constants" used to produce the binary instructions, and therefore the compiler itself, took up a smaller amount of storage. This enables the number of compilation passes to be reduced from 3 to 2 (the assembly being incorporated into the second pass). Passes 2 and 3 of the H.400 compiler had to be separate only because of the small amount of core storage available. This introduced many inefficiencies into the system. These have now been removed by eliminating the third pass.

The first pass of the compiler accepts the C.S.L. source statements, divides them into words according to rules of C.S.L. The definitions are then translated into instructions to lay out word marks in appropriate area storage. Statements which are not definitions are searched for key words, indentations and destinations, and are written on the magnetic tape for input to pass 2. During this pass a source language listing is produced which contains all the diagnostics which are discovered to be necessary during pass one. It was decided to produce the listing at this early stage so that the peripheral operation of printing could be overlapped with

the operation of card reading. This decision probably halves the compilation time, but it means, however, that any diagnostics produced in pass 2 have to be separated from the listing. The savings in machine space in pass 1 due to direct translation to binary have enabled a very extensive diagnostic to be incorporated in this pass. In fact the diagnostics produced by the compiler are as extensive as most FORTRAN systems although no special passes are used. This is possible only because of the convenient structure of C.S.L., e.g. the checking of the validity of branches can easily be done in the single pass because of the indentation. The FORTRAN rule that we must not jump into a loop, becomes simply a rule that the indentation of the statement to which we were jumping must be less than or equal to the indentation of the branch statement. The fact that the labels are local to an activity and that the terminology of C.S.L. makes it unnecessary to use very many labels means that a complete table of label references can be accumulated during the compilation of an activity and comprehensively checked at the end of an activity. The manipulation of sets is very extensively checked since it was found in earlier versions of C.S.L. that the majority of programming errors occurred in this area. The most popular undiagnosed error in the past was the changing of a set while it is controlling a loop. While this was not detected by the compilers it had unfortunately the habit of getting the program into a never ending loop during execution.

The second pass of the compiler largely consists of a number of mutually recursive subroutines. The recursive nature is necessary because of the generality of expression permitted in the C.S.L. statements. However, this in fact simplifies the compiler rather than complicating it, allowing for an efficient fast compiler to be provided in only 16K characters of storage.

Sample program

As an example of a complete E.C.S.L. program the author has provided the following, which is the simulation of a three-stage production belt.

```
CLASS TIME ITEM 100 SET ONLINE 3, BIN,
1 INPROCESS 3
CLASS TIME OPERATOR 3
ARRAY STATIONWIDTH (3), C(3)
HIST STNOUTPUT (3, 1, 1) INTERARRIVAL 3
1       (25, 0, 1)
HIST FREETIME 3 (25, 1, 1) JOBTIME (20, 1, 1)
HIST PICKUP 3 (10, 0, 1) OPTIME 3 (20, 1, 1)
CLASS TIME COMPLETEOPN 3
READ STATIONWIDTH, FEEDTIME, LENGTH,
1  STRA, MINTIME, C|JOBTIME ITEM
100 LOAD BIN
T. FEED = 1
ACTIVITIES LENGTH
BEGIN REJECT ITEM
FOR I = 1, M
    FIND A ONLINE. I. FIRST
    T. ITEM. A LE 0
    ITEM A FROM ONLINE I INTO BIN
    OR CONTINUE
BEGIN TO NEXT STATION
FOR I = 1, M
    T OPERATOR I EQ 0
    FIND A INPROCESS I FIRST
    ITEM A FROM INPROCESS I INTO ONLINE
1       (I + 1)
    T. ITEM A + STATIONWIDTH (I + 1)
    ADD I, STNOUTPUT
    AND — T. COMPLETE. I. INTERARRIV, I
    T. COMPLETE I = 0
    OR CONTINUE
BEGIN WORKING
FOR I = 1, M
    T. OPER TOR. I LE 0
    FIND A ONLINE I FIRST
    ITEM A FROM ONLINE I INTO
1   INPROCESS I
    ADD —T. OPERATOR I, FREETIME I
    T OPERATOR I = MINTIME + (T. ITEM.
1   A + C(I)) + SAMPLE (JOBTIME,
2       STRA)|(STATIONWIDTH (I) + C(I))
    ADD T. ITEM A, PICKUP POINT I
    ADD T. OPERATOR I, OPTIME I
    OR CONTINUE

FINALIZATION
OUTPUT, STNOUTPUT, (INTERARRIVAL I) I = 1,
1       K,/(FREETIME I) I = 1, M
2 (PICKUP I) I = 1, M, /(OPTIME I) I = 1, M
END
```

References

BUXTON, J. N. and LASKI, J. G. (1962). "Control and Simulation language", *The Computer Journal*, Vol. 5, p. 194.

— (1964). *Courtaulds All Purpose Simulator*, Programming Manual, Courtaulds Ltd., Coventry.

BUXTON, J. N. (1966). "Writing simulations in CSL", *The Computer Journal*, Vol. 9, p. 137.

LASKI, J. G. (1966). Letter to Editor, *Operational Research Quarterly*, March 1966.