Control and simulation language

By J. N. Buxton and J. G. Laski

CSL is a programming language designed for use in the field of complex logical problems. The basic approach adopted is that groups of items or objects sharing similar qualities can best be represented intensively, by recording the names of these items in predicative groups or sets, rather than by listing them extensively and coding their qualities numerically.

The first application of the language has been in the field of Monte Carlo simulation work, for which special facilities are provided.

An unusual compilation approach has been followed, involving translation into FORTRAN as an intermediate stage.

Introduction

CSL is a programming language designed to aid in the solution of complex logical and decision-making problems. The language is experimental in type; it derives from the observation that thinking on real-life decision-making problems is based on the use of sets, and it is designed to test the applicability of this thesis. The manipulation of sets is formally described by the predicate calculus; therefore the basic approach in CSL is that of the predicate calculus.

There is considerable redundancy in the facilities provided in the language; the less useful of these will be dropped in the light of practical experience rather than by arbitrary decisions in the design stage. No attempt is made in this paper to describe CSL completely as some sixty statement types are available. This experimental approach to language design has led to a corresponding approach in design of the compiler, where rapid preparation and ease of subsequent modification has been emphasized.

The first part of this paper describes in more detail the basic ideas of the language. A representative selection of the available types of statement is given, and the paper concludes with some comments on the compilation approach which has been adopted.

A syntactic description of CSL tests is given in Appendix 1, and a short program is given as an example in Appendix 2.

Terminology

In the examples given in the text, the following terminological conventions have been used.

1. Structural CSL words are printed in bold type in this paper; in handwritten programs it is good practice to underline them to increase transparency.
2. SETA, SETB, – – – are names of sets.
3. ENTITY. P is a typical entity.
4. E, F, G, are single valued arithmetic expressions.
5. l_1 and l_2 are statement labels.
6. I, J, K are cell names.
7. P is an integer, a cell name or an expression in parentheses.

Basic ideas

CSL (Control and Simulation Language) is designed as a tool for the formulation of complex decision-making problems such as those which arise in the control of industrial and commercial undertakings. It may help in the validation of suggested control procedures by constructing Monte Carlo simulation programs to model the system under study, or be used in the implementation of such procedures in a computer-controlled system.

To provide some point of contact between the discussion in this paper and the real world, we will take as an example the problem of distributing goods from a factory to various distribution points. The problem to be solved is roughly that of finding a scheduling procedure to decide which goods should be sent by which truck to which series of unloading points.

In describing such a situation in a computer program, it is necessary to talk about complex objects, such as trucks and loading bays, and to describe their interactions. These objects, which we will call *entities*, have qualities or attributes which may for example, be described by holding numerically coded information in arrays of storage cells. Each truck is a complex object and associated with it are various attributes, such as its capacity, total mileage and average fuel consumption. This data could be stored in arrays in two ways: by associating one attribute array with each truck, or by using the truck serial number to index data arrays which hold data for the entire fleet.

Traditionally the interactions of these entities have been described by the manipulation of queues. The central theme in CSL is that such queues are, in fact, ordered sets, and as such are amenable to treatment by operations on sets similar to those described by predicate logic. Scheduling decisions in the despatching problem will involve discussion of sets of trucks, such as the set of all trucks at present in the factory, or the set of all trucks in the factory, with capacities above 3 tons and with mileage to next scheduled overhaul greater than 1,000 miles.

A set is a storage device which records the names of entities. It may be used to record the names of those entities which are in a specific state or possess a specific range of qualities, and thus it provides a way of recording the attributes of entities which is a third alternative to the holding of numerically coded information in data or attribute arrays as described above. The order in which entity names are placed on a set is preserved, and so a set can be used to provide the facilities of a queue.

The subgroup of entities whose names appear on a given set may be referred to intensively by use of the name of the set, rather than extensively by listing the subgroup. Computing statements can now be designed to operate in various ways on the members of a set, or to use the membership of a set to control their operation. The principal computing technique in CSL is the use of the membership of sets to control computing operations.

Variable storage

CSL names are strings of letters of any length. They may not be CSL structural words. Four basic types of storage are used; cells, arrays, entities, and sets. The first three types are known as *variable storage.* A cell provides space for a single variable, and is addressed by a name. An array of storage locations resembles a subscripted variable as used in FORTRAN and other standard computing languages, and a specific member of the array is addressed by a subscript list thus,

DATA (E, F, G)

Entities of similar size and shape, and used for the same purpose, are regarded as grouped into a class and are given similar names; in the despatching problem trucks are known as

TRUCK. 1, TRUCK. 2, TRUCK. P, . . .

A specific truck of the class is referenced by use of a class subscript separated from the class name, as above, by a dot. This class subscript may be an integer or it may be specified indirectly by giving a cell name or using an expression in parentheses. An expression is any single-valued arithmetic expression, as in FORTRAN.

If an attribute array is associated with each member of the class, then the attribute cells are referenced by a subscript list, thus

TRUCK. P (E, F)

The subscripts used to reference data-array or attribute-array cells may be expressions, and as expressions may involve other reference to subscripted storage, this permits indirect reference or subscripting to any level.

Sets are devices for recording names of entities as described above; more precisely, a set is an ordered list of class subscripts. Though a given entity may have its name recorded on many sets, any given set may only hold the names of entities of a specific class. The reason for this restriction is given in the notes on compilation, but semantic considerations cause this, in fact, to be no limitation.

As an example of the three alternative ways of recording information on the qualities or attributes of entities, consider the *j*th attribute of the *i*th truck, which might be its present location. If each truck has associated with it a one-dimensional attribute array, then the location is expressed by a numerical code in cell TRUCK. I (J). If a one-dimensional data array named LOCATION is used to hold the locations of all trucks, then the coded information is in cell LOCATION (I). The third way of holding the information is by the presence or absence of the name TRUCK. I on a descriptive set, such as a set called GLASGOW. As most computing statements in CSL are controlled by reference to membership of sets in general this latter alternative is preferable.

Simple manipulative statements and tests

The customary facilities for performing arithmetic operations on variable storage are available, and are not further described as they resemble those in FORTRAN and similar languages.

Any statement may be labelled with an unsigned integer below 25,000.

The simple control statement is of the form

$$E \not\phi\ F \quad l_1 \& l_2$$

E and F are expressions, $\not\phi$ is one of the six relational operators $>$, $\geqslant$, $=$, $\neq$, $<$, $\leqslant$ expressed in two-letter form as **GT, GE, EQ, NE, LT, LE,** and l_1 and l_2 are labels. If the result of the test is successful, control is transferred to the statement labelled l_1, and if the test fails control goes to that labelled l_2. A CSL program may be divided into sectors, and either or both of the destination labels may be omitted. If the success destination is not specified, the compiler assumes it to be the next statement, and an unspecified failure destination is assumed to be the first statement in the next sector.

This form of destination is used in all CSL tests. It reverses the customary approach to control transfers in computing languages, in that when specific destinations are not given, transfer takes place on a test failure and not on a success. In a program which consists mainly of logical analysis this approach gives a much easier logical flow, and in most CSL programs explicit destination labels rarely need to be used.

Entity names may be placed on or off sets by statements of the form

ENTITY. P	**HEAD**	SETA	l_1 & l_2
ENTITY. P	**TAIL**	SETA	l_1 & l_2
ENTITY. P	**FROM**	SETA	l_1 & l_2

These statements will add the specified entity to the set at the head or tail of the queue respectively, or will remove it from the set. Note that each of these statements implies that a test is carried out to determine whether the entity is on the set, and so a destination is given. For the first two statements, location of the entity already on the set is a failure, and for the third it is a success. Subsequent detection of the names of the entities currently at the head or tail of the queue is carried out by the **FIND FIRST** and **FIND LAST** statements which are described in a later section of the paper.

The statements

ZERO SETA
ENTITY **LOAD** SETA

will cause the set to be emptied or to be filled with all members of its class.

The statements

```
ENTITY. P    IN       SETA    l1 & l2
ENTITY. P    NOTIN    SETA    l1 & l2
```

carry out simple tests on the occupancy of sets.

Repetition of a group of statements is carried out by the use of a **FOR** statement which can exist in two forms. The repetition instruction may be given as in a FORTRAN **DO** statement;

FOR I = n1, n2, n3

n1, n2 and n3 may be positive integers or cell names, and I will be advanced from n_1 to n_2 in steps of n3. The alternative form is

FOR I SETA

and in this case I is made equal in turn to all the class subscripts at present recorded on the set.

The range of statements over which a **FOR** operates is given by the fact of their indentation to more places from the left-hand side of the source program than the indentation of the controlling **FOR** statement. Repetition statements may be nested within each other to any depth.

DUMMY

This is a dummy statement which may be used, for example, to place a statement label at the end of a **FOR** loop.

Complex tests

The basic unit in the complex tests and compound statements is the test chain; this is a truth functional chain of tests in what is described by logicians as disjunctive normal form; for example

```
      test } group 1
      test }
OR    test } group 2
OR    test }
      test } group 3
      etc.
```

The extent of the test chain is defined by the equal indentation of all tests in the chain, omitting the **OR**'s from consideration in the indentation. The result of the consideration of such a chain is a success if all the tests in one or more of the groups between successive **OR**'s are successful; the meaning of the above chain is (test **and** test) **or** (test) **or** (test **and** test), etc.

Statements other than tests may be interspersed in a chain; they will be carried out if the previous test is encountered successfully. By this means one can set markers to indicate which path through a chain was followed during execution.

An example of a complex test is the **ALL** test;

```
ALL  I  SETA    l1 & l2
     test chain (I)
```

This is a test to determine whether the following test chain, which contains mention of cell I, is satisfied with I such that ENTITY. I is in SETA, for all the present members of the set.

```
EXISTS   (E)    I    SETA    l1 & l2
   test chain (I)
```

This test determines whether the test chain condition is satisfied by at least as many members of SETA as the present numerical value of (E).

```
UNIQUE   (E)    I    SETA    l1 & l2
   test chain (I)
```

This test requires that the test chain be satisfied for precisely as many members of the set as the present numerical value of (E).

In the **EXISTS** and **UNIQUE** statements, the test chain may be omitted as the result is a meaningful test on the set population, and the expression (E) may be omitted when the value unity is assumed.

Search statements

There are five variants of the search or **FIND** statement, of which one example is

```
FIND  I  SETA  MAX  (E)    l1 & l2
   test chain (I)
```

This statement locates that member of SETA which satisfies the test chain and which maximizes the expression (E). On exit from the statement, the class subscript of the successful member is set in I.

In the other variants of this statement, the criterion **MAX**(E) is replaced by one of the criteria **MIN**(E), **FIRST**, **LAST** or **ANY**. The test chain may be omitted in all variants, as for the **EXISTS** and **UNIQUE** complex tests. The **FIRST** and **LAST** options will select the first or last member satisfying the test chain condition, regarding the set as a queue, and the **ANY** option makes a random choice from the members who satisfy the condition.

A destination is specified in a search statement; the fail transfer will be used if it is not possible to select a member of the set who satisfies all the conditions.

Nested tests

In the description of the test chain given above, no definition was given of the type of test which may be used as a member of the chain. In fact, any test defined in CSL may be a member of any chain, including complex tests and search tests. A complex test or search test which is a member of a chain may itself involve a sub-chain, including other complex tests, and so on to any depth. This ability to use complex tests nested to any depth gives the language considerable analytical power and flexibility.

As an example in the despatching problem, for a particular long night haul, we might wish to locate the first truck (X) in the waiting queue, of capacity 3 tons or more and with more than 1,000 miles to next overhaul, provided that we can also locate any driver (Y) who is a union member, is prepared to drive overnight and is permitted to drive trucks of the tonnage of truck (X). If a truck and driver are available, take the next statement, otherwise transfer control to the statement labelled 105. The following statement paragraph performs the entire operation, assuming suitable storage allocations:

```
FIND   X   IDLETRUCKS    FIRST    & 105
   TRUCK. X (1)   GE 3
   TRUCK. X (2)   GE 1,000
   FIND    Y IDLEDRIVERS ANY
      DRIVER. Y       IN UNION
      DRIVER. Y       NOTIN DAYWORKONLY
      DRIVER. Y(1)    GE   TRUCK. X (1)
```

Program testing

Trace statements are provided to facilitate recording of the contents of specified cells during execution. The resulting output gives the cell contents and their source-language names, so that program testing can be undertaken without recourse to lower levels of language.

At the time of writing an index checkmode compilation facility is being added to the compiler. In sections of program between the words **START** and **FINISH**, extra orders will be inserted to provide dynamic checks, during execution, on the contents of all index values, to ensure that any subscript value used to access any array cell or attribute is within its allowed range. In a language which permits indirect reference to any level, this facility should be available.

Monte Carlo simulation facilities

The language contains several facilities, such as the provision of cells for holding time values associated with each entity, which are useful in writing simulation programs, and in providing a conceptal framework for the expression of such problems. The facilities are based on those in the General Simulation Program (Tocher and Owen, 1960) and in Montecode (Buxton and Kelley, 1962). No further description is given here as in this paper the authors wish to emphasize the nature of CSL as a general programming language.

Compilation techniques

CSL was produced as an experiment, to study the practical utility of some suggested language facilities based on a combination of queuing techniques with predicate logic in the solution of decision-making problems. The language is rather extensive, and in designing the compiler, emphasis was placed on speed of preparation of the compiler and ease of subsequent alteration, rather than on efficient compilation.

The CSL object computer is an IBM 7090, a machine of suitable size and power for large and complex problems: (it has core storage size of 32,768 words and a cycle time of $2 \cdot 18\, \mu$ sec.). A FORTRAN compiler is in wide use on the 7090, and it was decided to take advantage of its existence by compiling CSL into FORTRAN as a lower-level language. Sets are translated into one-dimensional FORTRAN arrays and handled as push-down lists, and a class of entities becomes a single array with one more dimension than its constituent entities. The price paid for this approach is the introduction of some source-language restrictions, of which the most obvious are the inability of a set to hold names of entities other than those of a specific class, and the inability to address sets indirectly.

The 7090 is a magnetic-tape oriented system, and for off-line work in almost all installations, 1401 computers are used. The translation of CSL into FORTRAN is done on the 1401, as this is a variable word-length machine handling alphanumeric characters, and so is ideally fitted for the string manipulations which arise in compiler work.

Translation is done as a four-stage process. The first stage translates CSL names into FORTRAN names, detects CSL syntatic words and replaces them by single-character tags which precede the statement, and performs those sections of compilation which need reference to the source language, such as trace statement compilation.

In the second phase a main read program studies the tags preceding each statement, and calls the relevant generator subroutine to analyse the statement and to write out a suitable section of FORTRAN or of simpler CSL as a replacement for this statement in the body of the program. Those generators which need test chains use a subsidiary chain read routine to translate them.

The simple generator approach is complicated by the small size of the 1401 store, which can only hold a few generator routines at a time. This difficulty arises because the 1401 is basically a commercial machine, and it is overcome by repeated scanning of the source program, which is written to and fro between two tape units with different generators in store. After each pass, the program is a little longer and contains rather more FORTRAN and rather less CSL.

With this relatively small store, the use of indentation to define ranges of **FOR** statements and test chains caused some problems. The **FOR** generator operates recursively, and when it is in the store and is compiling over a range or nested set of ranges, the identation structure of that section is deleted, even though the majority of statements in the range cannot yet be compiled due to lack of suitable generators.

The test chain read does not operate recursively; complex tests encountered in a chain are modified by the inclusion of a suitable destination phrase and are copied through, with their test chains and associated indentation structure, for later consideration.

On conclusion of this phase, the program exists in FORTRAN logic, but it may contain variables with subscripts more complex than is allowable in FORTRAN. In the third phase a syntactic analysis of subscript form is carried out, and dummy variables and suitable setting statements are inserted where needed. At present CSL is restricted so that emergent FORTRAN arrays are three-dimensional or less, but if this is found to be troublesome, the third phase will introduce address mapping functions for higher dimensions.

The fourth phase carries out some error analysis and tidies up the program into final FORTRAN form. Compilation is now completed by the FORTRAN compiler on the 7090.

Operational experience

This is limited, as yet, but the results seem promising. The compiler took about 9 man-months to write and produces a fairly efficient object program, from the point of view of both storage use and running speed. It is extremely easy to modify and extend, as the generators can be changed independently of each other. The ratio of CSL to FORTRAN statements produced, is of the order of 1 to 5, and the ratio of time spent in writing similar programs in CSL and in FORTRAN is also of the order of 1 to 5.

The most interesting point to emerge from use of the language is that several problems, which had not previously been tackled due to difficulty in formulation, have now been formulated with little trouble.

Acknowledgements

On completion of their present work on CSL, the authors became aware of the fact that very similar work had been brought independently to much the same stage by Dr. H. M. Markowitz and his colleagues, at the Rand Corporation.

In the simulation features in the language, we owe a considerable debt to the pioneer work in this field of Dr. K. D. Tocher and his colleagues at United Steel Companies Ltd.

We wish to acknowledge the advice and assistance we have received from many of our past and present colleagues, from Mr. D. F. Hartley of Cambridge University who helped us with work on the syntax of the language, and in particular from Mr. I. J. Cromar of IBM who bore a considerable part of the work in writing the compiler.

We are grateful to IBM United Kingdom Ltd. and Esso Petroleum Co. Ltd., under whose auspices this work has been carried out, for permission to publish this paper.

References

Backus *et al.* (1960). "Report on the Algorithmic Language ALGOL 60," *Numerische Mathematik*, Vol. 2, pp. 106–136.

Kelley, D. H., and Buxton, J. N. (1962). "Montecode—An interpretive program for Monte Carlo simulations," *The Computer Journal*, Vol. 5, p. 88.

Tocher, K. D., and Owen, D. G. (1960). "The Automatic Programming of Simulations," *Proceedings of the International Federations of Operational Research Societies Conference*. Aix-en-Provence, pp. 50–67.

Appendix 1

CSL test syntax

The notation developed by Backus *et al.* (Backus *et al.*, 1960) is used, extended by the following unconventional syntactic symbols:

⊁ means "end of line"
ψ means "indentation as on the previous line"
→ means "indentation to the right of the indentation on the previous line"
L means "include the following category on the first line of this structure."

The categories ⟨label⟩, ⟨empty⟩, ⟨expression⟩, ⟨entity⟩, ⟨cell⟩ and ⟨set⟩ are left undefined in this extract; the following syntax is intended for illustrative purposes rather than for formal definition, and the meaning of the undefined categories is clear enough for this purpose.

⟨destination⟩ ::= ⟨label⟩ & ⟨label⟩ | ⟨label⟩ & | & ⟨label⟩ | ⟨empty⟩

⟨relation⟩ ::= **LT** | **LE** | **EQ** | **NE** | **GE** | **GT**

⟨simple test⟩ ::= ⟨expression⟩ ⟨relation⟩ ⟨expression⟩

⟨set test⟩ ::= ⟨entity⟩ **IN** ⟨set⟩ | ⟨entity⟩ **NOTIN** ⟨set⟩

⟨set compound⟩ ::= ⟨entity⟩ **HEAD** ⟨set⟩ | ⟨entity⟩ **TAIL** ⟨set⟩ | ⟨entity⟩ **FROM** ⟨set⟩

⟨complex test⟩ ::= ⟨description line⟩ → ⟨test chain⟩ | ⟨exists line⟩ | ⟨unique line⟩

⟨find compound⟩ ::= ⟨find line⟩ → ⟨test chain⟩ | ⟨find line⟩

⟨test⟩ ::= ⟨simple test⟩ | ⟨set test⟩ | ⟨complex test⟩ | ⟨find compound⟩ | ⟨set compound⟩

⟨test chain⟩ ::= ⟨test⟩ ⊁ | ⟨test chain⟩ ψ ⟨test⟩ ⊁ | ⟨test chain⟩ **OR** ψ ⟨test⟩ ⊁⟩

⟨exists line⟩ ::= **EXISTS** (⟨expression⟩) ⟨cell⟩ ⟨set⟩ ⊁ | **EXISTS** ⟨cell⟩ ⟨set⟩ ⊁

```
<unique line>::= UNIQUE (<expression>) <cell>
                     <set> ≯ |
                 UNIQUE <cell> <set> ≯
<description line>::= ALL <cell> <set> ≯ | <exists
                     line> | <unique line>
<find line>::= FIND <cell> <set> <criterion> ≯
<criterion>::= FIRST | LAST | ANY |
               MAX ( <expression> ) |
               MIN ( <expression> )
```

```
<simple test statement>::= <simple test>
                           <destination> ≯ |
                           <set test>
                           <destination> ≯
<complex test statement>::= <complex test> L
                           <destination>
<find compound statement>::= <find compound> L
                           <destination>
<set compound statement>::= <set compound>
                           <destination> ≯
```

Appendix 2

Example

This example is based on a CSL program used by Thomas Skinner & Co. Ltd., publishers of the ABC timetables, to work out routings for the Quick Reference Section of the ABC World Airways Guide.

Initial statements in the program read in a two-dimensional data array of mileages between all airports in a given part of the world, and establish three sets which hold subgroups of airports. The first of these, AIRPORTS, holds the names of those airports between which possible transfer routings involving one change of aeroplane are required. The second set, TRANSFERPORTS, holds the names of the major airports where transfer facilities are possible. The third set, USED, is used during the program as a working-space set. A transfer routing is permissible provided that the total mileage flown does not exceed the direct mileage by more than 15%.

The following program establishes valid transfer routings. The initial statements are omitted and the output statements are stylized to avoid the introduction of detail which has not been fully described in the paper. The transfer airports for each airport pair are written out in the order of increasing total mileage.

```
    FOR A AIRPORTS
      FOR B AIRPORTS
        A LT B  & 2
        WRITE A, B
        ZERO USED
1       FIND X TRANSFERPORTS MIN
            (MILEAGE (A, X)+MILEAGE (X, B))
            & 2
          X NE A
          X NE B
          100* (MILEAGE(A, X)+MILEAGE (X, B))
            LE 115*MILEAGE (A, B)
          AIRPORT. X NOTIN USED
        WRITE X
        AIRPORT. X HEAD USED
        GO TO 1
2       DUMMY
    EXIT
```

Book Review

An Introduction to Numerical Methods, by R. BUTLER and E. KERR, 1962; x + 386 pp., 8¾ × 5½ in. (*Pitman.*) 40s.

In spite of reference in the Preface to the high-speed computer and to automatic computing, this book is entirely concerned with methods for desk computing, and even then only with the more elementary ones.

It is a long-drawn-out exposition of these elementary methods with a large number of numerical examples.

It contains a discussion of rounding-errors (in chapter one), the solution of algebraic, transcendental and simultaneous linear equations (avoiding matrices), and including accounts of synthetic division, Horner's method, etc. (chapter two), Finite Differences (chapter three), Interpolation (chapter four), Numerical Differentiation and Integration (chapter five) and the solution of differential equations (chapter six), all at a level much more elementary than, for instance, in Hartree's *Numerical Analysis*.

J. C. P. MILLER.