UNRAVEL—a programming language to put intelligence into dumps

P. J. Brown

Computing Laboratory, University of Kent at Canterbury, Cornwallis Building, The University, Canterbury, Kent

The need to resort to using dumps of core storage when debugging programs has been a continuing feature of computing. These dumps are like vermin: attempts to eradicate them from some areas have succeeded but they continually reappear in other areas. Each new machine or operating system brings new problems. This paper presents a new weapon, which the user can employ to keep dumps in check.

(Received January 1972)

UNRAVEL is a programming language for printing out information from core store. There already exist several dumping programs that do this, so it is best to start by describing how UNRAVEL differs from them.

There are two main problems with traditional dumping programs. Firstly information is printed out in a uniform format, to a uniform base (e.g. octal) and without any interpretation or annotation. The unfortunate reader of the dump has to go through a mass of information to extract what he needs. Often he has to perform tortuous conversions, e.g. from octal to decimal, character, or program address.

The second problem with conventional dumping programs is that it is often impossible to extract information that is indirectly addressed, for example: given that a certain address points at a 40 word table, print the table, or given the start of a linked list, print all the items on the list.

The purpose of the UNRAVEL system is to surmount these problems by providing the user with a programming language to specify how a dump is to be made. In this way UNRAVEL can be used to put 'intelligence' into dumps. When a dump is required, the UNRAVEL processor is brought into action, and the UNRAVEL program supplied by the user is executed.

An UNRAVEL program can be made to interpret the material to be dumped to save the reader the trouble of doing so. For example assume that a 24-bit word of information in a table describes the usage of an I/O device in the following way

First two bits :	state, e.g. free, busy.
Next seven bits :	priority level.
Next fifteen bits :	pointer to name of user

and assume further that the table contains 30 entries, corresponding to devices 0 to 29. An UNRAVEL program could be made to interpret the table accordingly and print out information in a form such as

```
DEVICE 0 BUSY AT PRIORITY LEVEL 6.
                              USER IS CU/RO99
DEVICE 1 FREE
DEVICE 2 . . .
```

The object machine

UNRAVEL can be implemented on almost any machine, whether a byte machine, a character machine or a word machine. It is also independent of the base to which the contents of core store are naturally interpreted, for example octal or hexadecimal. In this paper this base is called the *machine base*.

At present UNRAVEL has been implemented on two machines, the ICL 4130 and the PDP-11. In the former case there are two separate implementations, one for the on-line system and one for the manufacturer's batch system.

Typical actions of UNRAVEL programs

The most common use of dumps is in debugging, and this, therefore, is the most popular usage of UNRAVEL. The program that is being debugged is called the *spotlighted program*. At the end of each run of the spotlighted program, the user calls UNRAVEL to give him a dump, supplying a program in the UNRAVEL language to specify how the dump is to be made. Information that might be put into this dump includes the following

(*a*) the values of all the variables in the spotlighted program, with the name of the corresponding variable given against each value.

(*b*) information about the operating environment of the spotlighted program, for example the status of I/O devices, the contents of buffers, the location of workspace areas, the running time, the reason for failure, etc.

(*c*) workspace areas, printed in appropriate formats, for example dictionaries, stacks, lists.

This information might be supplemented by a conventional dump, which could be used as a last resort. However, the ideal should be to make the UNRAVEL program print out all the information that the user would have to extract from a conventional dump.

System variables

There may be some difficulty in finding out all the information described above. Difficulty varies according to the machine and, more particularly, the operating system. To help the user, UNRAVEL provides some fixed variables, called *system variables*, which are preset to point at information that may be of interest to the user. System variables vary between implementations. The following list, which describes the system variables for the ICL 4130 implementation, may serve as an example

(*a*) pointers to spotlighted program. On the ICL 4130, variables and constants are separated from code, and system variables are set to point at the start of each of the two parts of the program and to show how large each part is. It is possible for several programs to reside in core simultaneously, and if the user wishes to examine a different program he can use the UNRAVEL statement

PROG *program name*

This will cause the system variables to be reset to describe the new program.

(*b*) location of operating system table giving state of current program.

(*c*) important base addresses used by operating system, e.g. base of current slave.

Assuming that variables in the spotlighted program are in contiguous storage, or can be made to be so, a system variable pointing to the first such variable will be enough to let the user write an UNRAVEL program to give a complete dump of his variables. In some implementations it may not be so easy to find the first variable and the user may need to modify the spotlighted program to provide this information. The spotlighted program could, for example, plant in some fixed place a pointer to the first variable. A solution on the ICL 4130, where the variables follow the constants but only the start of the latter is known, is to make the first constant point at the first variable.

More will be said later about printing variables in the spotlighted program, but before this, the UNRAVEL language will be described.

The UNRAVEL language

The UNRAVEL language is a simple high-level language. The main features will be outlined here, and the reader who is interested in further details is referred to the User's Manual (Brown, 1971).

Variables are represented by identifiers. Names beginning with the letter '*Z*' are reserved for system variables. Variables have no data type (as in BCPL (Richards, 1969)), but are interpreted simply as a *word* of information. (The concept of a *word* can be defined for each implementation, though for most implementations there will be an obvious interpretation.) If arithmetic operations are performed on variables, the variables are treated as integers.

Constants are specified in decimal or in the machine base. In the latter case they must start with the digit zero. It is assumed in examples in this paper that the machine base is octal. Sample constants are 20 and 0177.

Expressions are written in the way used in most high-level languages. The available binary operators are add (+), subtract (−), multiply (*), divide (/), logical 'and' (&), logical shift (↑), and 'field' (represented by a comma). The field operator is used to extract a field from a word and is a short way of masking and shifting. Sample expressions are

$$(X + 3) * Y$$
$$(\text{STATUS}\&(7 * 8)) \uparrow - 3$$

There are two unary operators: unary minus (−) and indirect addressing (represented by a colon). The indirect addressing operator is, of course, vital as it is the only way of looking at the areas of core to be dumped. All indirection is performed relative to a system variable called ZBASE. The system will initialise ZBASE to some appropriate value, but the user can change it. For example if ZBASE points at the first variable in a program then

```
: 12
```

gives the value of the twelfth variable. (Strictly speaking, the address taken is twelve storage units beyond where ZBASE points. On a machine where the storage unit is a byte, a variable might occupy several storage units, so it might only be, say, the third variable rather than the twelfth that is at offset 12 from ZBASE. Indirect addressing always selects a word of information, but the user can, of course, easily mask off individual bytes.)

Statements

Statements in UNRAVEL are terminated by the end of a line or by a semicolon. There are no special rules about statement format. Where appropriate, UNRAVEL statements have been based on the BASIC programming language. The assignment (LET), comment (REM), GOTO and subroutine calling statements are almost identical to those in BASIC. Statements may optionally be labelled by numerical labels. Any statement may be preceded by one or more IF clauses, though the syntax and semantics of these are not quite the same as in BASIC. In particular if an IF clause does not hold, control passes to the next *line* of the program.

The only specialised statements are those for printing. Here the philosophy is that the printing format is entirely under the control of the user. The system, therefore, never adds any extra spaces, tabs or newlines unless the user asks for them. The following are some of the available printing statements

D *expression*	print value of *expression* in decimal.
M *expression*	print value of *expression* in the machine base.
C *expression*	print value of *expression* as a character string.
NL	print a newline.
NL *expression*	print a number of newlines.
TAB	print a tabulate character.
TAB *expression*	print a number of tabulate characters.
'*string*'	print a string.

Thus for example

```
'STACK SIZE IS'; D:12 − :16; NL
```

might print

```
STACK SIZE IS 29
```

Sample programs

Two examples of complete UNRAVEL programs follow.

The first program corresponds to the example mentioned earlier, concerning the table of uses of I/O devices. It is assumed that this table is pointed at by location 41 relative to the current base. One feature of this example, which has been found to be valuable in many UNRAVEL programs, is that the program contains checks for invalid data. In this case it is assumed that the value of the priority of a device should be less than ten; if it is not, a message is printed. This is an example of 'second-level' intelligence—not only does the program present information in a readable form but it also tries to suggest where the errors are. The program is as follows

```
    LET TABLE = :41
    REM LOOP, X GOING FROM 0 TO 29
    LET X = 0
10  'DEVICE '; D X; ' '
    LET STATE = :(TABLE + X)&060000000
    IF STATE = 0 'FREE'; GOTO 20
    IF STATE = 1 'BUSY'
    IF STATE = 2 'WAITING'
    LET PRIORITY = (:(TABLE + X)&017700000) ↑ −15
    IF PRIORITY > 10 NL 2; 'BEWARE: PRIORITY
                                    TOO HIGH'; NL 2
    'AT PRIORITY LEVEL '; D PRIORITY
    REM ASSUME POINTER TO USER POINTS AT
                          NAME PACKED INTO 2 WORDS
    LET USER = :(TABLE + X)&077777
    '.    USER IS '; C :USER; C :(USER + 1)
20  NL
    IF X < 29 LET X = X + 1; GOTO 10
```

The output would consist of thirty lines of a form such as

```
DEVICE 0 FREE
DEVICE 1 BUSY AT PRIORITY LEVEL 3. USER IS
          CU/RO99
DEVICE 2 . . .
```

The second example shows how a stack might be printed out. It is assumed that the stack contains decimal numbers and is pointed at by three variables in the spotlighted program as follows

STACKPT	points at the first item on the stack.
TOPPT	points at the top item on the stack.
LASTBL	points at some intermediate point.

These variables are at offsets 8, 12 and 16 respectively from the current base. It is desired to print the stack out in a diagramatic form, which might look like this

```
STACKPT  ⟶ 3
             7
             1
LASTBL   ⟶ 2
             2
             1
             4
TOPPT    ⟶ 0
```

The program to accomplish this is as follows

```
    LET X = :8
    'STACKPT ⟶'
10  D:X; NL
    LET X = X + 1
    IF X = :16 'LASTBL ⟶'; GOTO 10
    IF X NE: 12 TAB 2; GOTO 10
    'TOPPT ⟶'; D:X:NL
```

Finding variable names

One of the most important uses of UNRAVEL programs is to print out the names of the variables in the spotlighted program with their corresponding values. Hence the UNRAVEL program needs to be told what the names of the variables are. There are basically three methods of doing this. In explaining these methods it will be assumed, for the sake of example, that the spotlighted program is written in assembly language (or passes through assembly language at some stage of its compilation process).

The first method is for the assembler to communicate its symbol table to UNRAVEL. This method is used in the DDT debugging system (Digital Equipment Corporation, 1969), available on some PDP computers. If it can be achieved it is the best solution.

The second method is to write a simple preprocessor that will scan an assembly language program and calculate the relative addresses of all the variables. It is best if such a preprocessor outputs a series of UNRAVEL statements. For example if PIG is a variable at offset 17 from the first variable then the UNRAVEL statement

```
LET PIG = 17
```

would be generated. UNRAVEL is made to take the preprocessor output and compile it with the user's program. The user can then write statements such as

```
D :PIG
```

to print the value of his variable PIG. Such preprocessors should be easy to write using a text processing language or a general-purpose macro processor. A preprocessor for the ICL 4130 assembly language, for example, has been written in about forty lines using the ML/I macro processor.

The third, and least attractive, method is for the user himself to have to specify, within his UNRAVEL program, the relative offsets of the variables he wishes to examine.

Robustness

Any dumping program should be robust. It is possible, of course, for the user to make logical errors in an UNRAVEL program or for the data being examined to be so corrupt as to upset the way it is being interpreted. For example a linked list might link back on itself so that an UNRAVEL program that was printing the list would get into an endless loop.

Hence UNRAVEL needs to be implemented in such a way that all errors are detected within the UNRAVEL system rather than by the operating system under which it is running; errors can then be made fail-soft. Some ways of achieving this are to check all indirect addresses before they are accessed and to detect likely endless loops. The way the latter is done is to count backward jumps and to check this against a limit which is set by the system (but can be overridden by the user). This is not, of course, a perfect method but seems effective.

Uses

Some of the uses of UNRAVEL have already been shown. In its basic use as a dumping program it is probably at its most useful in helping to find the types of bugs that arise in established software or software that is in the final stages of development, as information is then in a fairly stable form and full use can be made of all the facilities of UNRAVEL for interpreting data structures. UNRAVEL should be a useful tool for those engaged in software maintenance.

When used in an interactive environment UNRAVEL has further uses. UNRAVEL programs can be written to print out the current state of the system, for example who the users are and what they are doing, and these programs can be kept on disc to be used when the need arises. Furthermore UNRAVEL programs can be written to monitor the system, for example to print out the contents of a storage location every time it changes. (In this case, assuming the program is invoked by a user at a console rather than by the operating system itself, scheduling problems may arise. The storage location being monitored might change more frequently than the monitoring program is scheduled. There are, therefore, limitations in its use.)

Implementation

To make UNRAVEL relatively easy to implement, its logic has been encoded as a series of macro calls that can be mapped, using any suitable macro processor, into the assembly language of any desired machine. There are, however, a number of implementation-dependent features that need to be coded by hand.

This general technique is described by Brown (1972). The language used for encoding the logic is based on the LOWL language, which, being set at a low level, is relatively easy to map.

In the current implementations, UNRAVEL compiles its program to a reverse Polish form and then executes this.

One of the main problems in each implementation of UNRAVEL has been to prevent it destroying some of the information it is supposed to be dumping. One solution has been to load UNRAVEL and compile its program before the spotlighted program is entered, leaving UNRAVEL set up and poised to give a dump when next requested.

References

Brown, P. J. (1971). *UNRAVEL user's manual for the ICL 4130*, Computing Laboratory, University of Kent at Canterbury.

Brown, P. J. (1972). Levels of language for portable software, to be published in *CACM*.

Digital Equipment Corporation. (1969). *PDP-10 reference manual*, Maynard, Mass., pp. 537-582.

Richards, M. (1969). BCPL: a tool for compiler writing and system programming, *Proc. AFIPS Spring Joint Computer Conference*, 34, pp. 557-566.