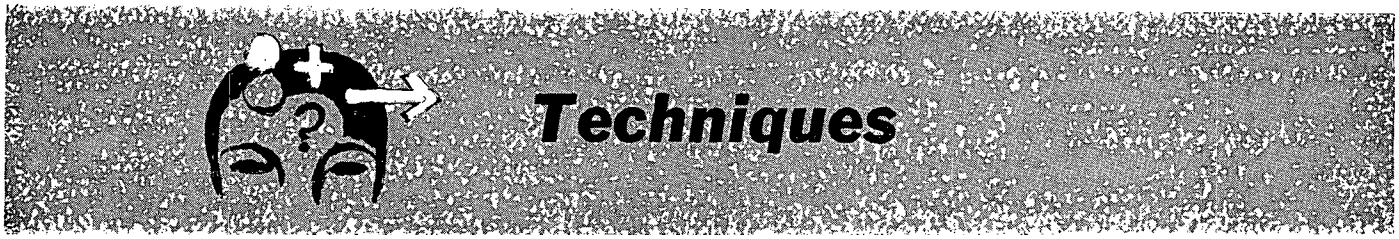

C. L. McCARTY, JR., Editor

Debugging Systems at the Source Language Level*

H. EARL FERGUSON AND ELIZABETH BERNER
University of California, San Diego
La Jolla, California

Introduction

One of the most important and time-consuming phases of programming is the testing of a program to insure that it works properly. It is therefore worthwhile to consider systems for facilitating this phase. In this paper the design of systems for mechanizing these tests is considered. A system for mechanizing debugging at the problem-oriented language (POL) level is described. This system facilitates program testing and minimizes the possibilities of introducing errors in the testing process. The system is well suited to organizations that need debugging facilities that are not provided by the compiler being used.

The debugging system has been implemented for FORTRAN as the source language and could be easily adapted to other problem-oriented languages. The implementation of the processor for mechanized debugging depends only on the POL syntax and is independent of the compiler. The syntax table approach of Irons [1] is used which provides that changes in the processor may be easily made to accommodate changes in the source language.

Considerations of System Design

The problems involved in debugging programs may be approached at several different levels and in several different ways. In the case of a compiler program, they may be approached either at the level of the absolute object code as loaded into the machine, or at the level of the relocatable object code before loading, or at the level of some intermediate language if provided by the compiler or at the level of the compiler source language. They may be approached by providing check-point information for comparison with intermediate results; they may merely provide intermediate results to the programmer; they may be approached by postmortem or snap dumps or by tracing sections of the program.

The check-point method [2] compares the intermediate results at some point with information provided by the programmer. If the comparison is not satisfactory, diagnostic information is provided and the program is then abandoned or may be continued by supplying the correct values to the check-point variables. If the comparison is satisfactory, the program will continue with perhaps a comment that the check-point was satisfied.

In determining which technique or techniques should be used at any particular installation, several factors should be considered. Some of these are the programming system or operating system used, the source languages used and the ease of knowing what portions of memory are occupied by the program. The mechanics of the installation and the operations system are not the only considerations; others, for example, are the sophistication of programmers using the system and the work capacity of the installation.

Some of these factors will now be considered to see what conclusions can be drawn. It is assumed that a large-scale high-speed computer is in use. Therefore to make efficient use of this costly device, it becomes necessary to use an automatic operating system (or monitor). Consistent with this efficiency consideration it is expected that this operating system will allow the programmer to compile and execute his program (if no fatal errors occur in the compilation) in a single computer pass. If this is done, it becomes difficult to use debugging aids which require knowledge of the program's ultimate location in memory, its relocated object program or the intermediate languages which might be used. It seems desirable to have some debugging scheme which uses the source language, whatever it may be.

The authors feel that the most severe restriction placed on debugging procedures is the lack of sophistication of programmers using the system. It appears that many programmers of large-scale computers today do not understand and often do not even wish to understand symbolic machine language. This puts severe restrictions on the types of debugging procedures that may be used, and, in fact, restricts them to debugging at the compiler language level.

The approach often taken is for the programmer to insert printouts in his compiler program at strategic points to obtain intermediate results which may reveal possible problems. This often leads to unnecessary trouble in restoring the program once it is checked and also may take several runs to debug the inserted printouts.

At some installations the amount of work on the computer may require the programmer to obtain the maximum

* This work was supported in part by Grant No. GP-536 from the National Science Foundation. Computations were performed on the CDC 1604 computer operated under Contract AT (11-1) Gen 10, Project Agreement 9.

amount of information out of a minimum number of runs because of long turn-around times. This implies the ability to make changes in the source program and to take a maximum of debugging information from a single run.

From the preceding discussion, it seems obvious that a source language debugging system is desirable at the POL level.

In considering the types of debugging that should be done, we discard the check-point method as a general debugging aid because nearly exact intermediate results are required which cannot be practically obtained in most cases.

The other types of debugging are normally applied at the relocatable or absolute object code level; however, this need not be the case.

The postmortem dump is not really applied at any particular level, but must be analyzed by examining the relocatable or absolute object code.

The trace, as usually utilized to trace object codes, can be expensive and cumbersome to use because of the great quantity of output produced, a majority of which is usually not pertinent. Therefore, it would seem to be advantageous to restrict tracing to three classes: (1) trace the flow of a subprogram, (2) trace entries to a subprogram and (3) trace particular variables in a subprogram.

Generally the current approach to the use of snap dumps is by reference to relocatable or absolute locations. This can be done in a compiler source language by referencing statement labels (or numbers) and variable names. In this manner no excess printout is necessary to obtain the desired information, and it is not necessary to have a compiled symbolic or absolute listing to determine where the snaps should be placed.

At the University of California, San Diego, a compiler source language debugging system for FORTRAN has been implemented. The debugging system is called BUGTRAN. The system includes variable trace, flow trace, trace of program entries and exits, snap dump, conditional termination of the program and an option to print the comments of the source program. It also includes the ability to delimit those portions of the program in which debugging is to be done. The system was particularly designed so that it would not affect the operation of the program and so that it would provide a convenient way of removing the debugging statements from the program. Another design criterion was that the system should not make any requirements on the compiler such as providing symbol tables [3].

BUGTRAN Debugging System

BUGTRAN is a tape-to-tape prepass debugging aid for use with FORTRAN. The FORTRAN symbolic source deck is preceded by a number of BUGTRAN control statements which specify the type of checking desired and give restrictions as to when this checking should take place. These restrictions include the subroutine to be checked, the area of program as delimited by statement numbers, the level of subroutine calls which will cause checking and a conditional statement (IF clause) which must be true for checking to occur.

This IF clause (IF(AE)) is perhaps the most powerful part of BUGTRAN debugging. The IF clause provides some of the power of the checkpoint method. In fact, the programmer may suppress diagnostic printouts except when special conditions or undesirable conditions arise that need attention, and may allow a program to execute normally until such time as these conditions do arise.

The format for BUGTRAN control statements follows (FORTRAN card format is used):

BN IN *NAME* (FROM S_1 TO S_2,J), IF (*AE*), *TYPE*, *LIST*.

in which

BN = BUGTRAN control statement number (optional)
NAME = subroutine or program name
S_1 and S_2 = statement numbers in the program; a single statement number (S_1) may replace (FROM S_1 TO S_2) or it may be left out entirely
J = level at which checking will occur; this is the level at which this subprogram will be executed, where the main program is executed at level 0; this may be omitted
AE = an arithmetic expression which is true (= 0) or false (≠ 0); in Fortran compilers which allow logical expressions it may also be a logical expression in which case true and false will be interpreted according to the compiler's convention; this may be omitted
TYPE = type of debugging desired (see options for checking below)
LIST = list of variables to be checked; this may be omitted.

An example of a BUGTRAN control statement is:

75 IN *BUGGY* (FROM 10 TO 20,3), IF (*X* − 4.5), *CHECK VARIABLES, A, B, INDEX.*

After BUGTRAN processes the control statements, it will read in the FORTRAN symbolic deck and make certain changes to it according to the specifications of the control statements—generally the addition to the FORTRAN source deck of various call statements to the BUGTRAN output routine.

Calls to the BUGTRAN output routine will generate at execution time a printout of the following general form:

BUGTRAN *BN(f)*LEVEL SN MESSAGE VAR = floating or fixed value,* AND *the octal value*

in which

BN is the BUGTRAN statement number that caused the output
f is the count of the number of times this point in the program has been reached
LEVEL is the level at which this subprogram was executed
SN is the Fortran statement number at which the checking was done
MESSAGE is an indicative message of the executed FORTRAN statement
VAR is the name of the variable whose value is listed (in two forms).

For example, the FORTRAN statement

10 IF(*A*) 100,300,500

might generate a call to the BUGTRAN output routine which would at execution time produce the following information:

BUGTRAN 75(0)* 3 10 *CONDITIONAL* GO TO 500 ON *A* = 1.520000E001,2004746314631463.

The BUGTRAN processor automatically makes the following modifications in every program that is to be checked:

1. Every DO loop will have the first statement and the terminal statement modified in order that a BUGTRAN call can be inserted before the loop termination and a transfer to the terminal statement can also be made:

$$\text{DO } nI = m_1, m_2, m_3$$

becomes

$$\text{DO } n'I = m_1, m_2, m_3$$

where $n' > 30000_{10}$ and the original terminal statement of the DO

$$n \cdots$$

is followed by

$$n' \text{ CONTINUE.}$$

2. Insertions are made for the following options for checking:

a. CHECK VARIABLES (variable trace). A list of variables is specified in the BUGTRAN control statement. If one of these variables appears on the left-hand side of an arithmetic substitution statement or as the index in a DO statement it will generate a call to the BUGTRAN output routine following the statement. For example, the statement

$$n \quad X = A$$

will have inserted as the next statement[1]

CALL BUGVAR (*X*, *2HX=*, *kHn*, *J,C,F*, *l*HBN)

and the statement

$$n \quad \text{DO } m \; I = n_1, n_2, n_3$$

generates as the next statement

CALL BUGLOOP (*I*, *2HI=*, *kHn*, *J,C,F*, *lHBN*)

in which BUGVAR and BUGLOOP are entries to the BUGTRAN output routine.

b. CHECK FLOW (flow trace). All GO TOs, ASSIGNs and IF statements generate calls to the BUGTRAN output routine.

c. DUMP (snap dump). A statement number is specified on the BUGTRAN control card with the variables to be dumped at this statement. A call to the output routine is generated so as to be executed immediately prior to the execution of the indicated statement.

d. CHECK ENTRIES. Everytime a SUBROUTINE, FUNCTION, END or RETURN statement is encountered, a call is generated to the output routine.

e. PRINT COMMENTS. All COMMENTs generate a call to the BUGTRAN output routine, causing the COMMENT to appear in the output.

f. TERMINATE. A statement number is specified on the BUGTRAN control card with an IF clause condition. The condition is checked immediately prior to the execution of the indicated statement and the program is terminated if the condition is met (terminates unconditionally if the IF clause is deleted).

[1] The *k* and *l* designate the integer value that is the number of characters to the right of the symbol "*H*" which follows them and to the left of the delimiters "," or ")".

Certain other options for debugging programs have been considered and temporarily tabled because of implementation problems. These are the ability to dump certain variables and arrays by name at the time a termination is made by the TERMINATE statement and a frequency clause added to most types which would allow the indicated type of debugging to take place only if the indicated frequency conditions are met. The latter addition can be at least partially handled by use of the IF clause.

Syntax Table Approach

A syntax table approach is used in interpreting the BUGTRAN and FORTRAN statements. This follows closely the methods described by Irons [1]. The BUGTRAN control statements are first interpreted using one syntax table. Then, depending upon which type of modifications are specified by the control cards, another syntax table is generated to process the FORTRAN statements. Provisions are made in the syntax table to interpret only those FORTRAN statements which will be processed; the other statements are not recognized and therefore not processed.

The advantages of the syntax table approach are that it is possible to use the same scanner to recognize various types of statements by merely changing the syntax tables and that changes in the source language require only modification of the syntax tables.

Acknowledgments. The authors wish to acknowledge the work of R. W. Mitchell, who initially conceived this project and was instrumental in formulating the specifications for the design of the BUGTRAN processor; Neal Friedman, who did much of the initial work in the design and planning of the implementation; and the many helpful suggestions of Dr. C. L. Perry and others in the design of the processor and writing of this article.

REFERENCES

1. IRONS, EDGAR T. A syntax directed compiler for ALGOL 60. *Comm. ACM 4* (Jan. 1961), 51–55.
2. JACOBY, K., AND LAYTON, H. Automation of program debugging. Philco Corp., PC No. 859, 8 Sept. 1961.
3. International Business Machines Corporation, 32K 709/7090 FORTRAN: Source Language debugging at object time. IBM 700/7000 Series, Data Processing Systems Bulletin, No. J28-6133, August 1961.
4. MILLER, JOAN C., AND MALONEY, CLIFFORD J. Systematic mistake analysis of digital computer programs. *Comm. ACM 6* (Feb. 1963), 58–63.