The debugging system AIDs

by RALPH GRISHMAN*

New York University

New York, New York

In comparison with the growth of procedural languages over the past decade, the advances in facilities for debugging compiled code have been small.† The debugging services offered today on most conversational systems have not advanced fundamentally from the design of DDT for the PDP-1.†† Batch systems have added a potpourri of other aids—in particular, systems with a machine simulator have included a variety of traces—but, in general, selective tracing and program checks of even slight complexity have been quite messy to invoke, if they were available at all.†††

The object of the AIDS project has been to provide a debugging system for FORTRAN and assembly language code on the Control Data 6600 which includes a flexible and reasonably comprehensive set of tools for program tracing and checkout, suitable for both batch and on-line use. A large variety of traces and checks can be invoked through a special “debug language” syntactically similar to FORTRAN. A system of such breadth is really practicable only on a machine with the power and memory capacity of a CDC 6600; such a large debugging system would be difficult to implement on some of the smaller machines on which the earlier interactive debugging systems were developed. At the same time, it is precisely the large, complex programs and supporting systems for machines of this size which make powerful debugging facilities so valuable.

HISTORY

The story of AIDS may be traced back to early 1965, when Prof. J. Schwartz initiated the development of a debugging system for the CDC 6600, which was soon to be delivered to New York University. This system, dubbed the WATCHR, was developed and expanded over the next two years by E. Draughon into a working debugging system.‡ As this system developed, several fundamental difficulties came to light. First, as the options proliferated, calling sequences became more complex, to the point where users not only could not possibly remember the calling sequences, but often would not attempt to invoke some of the more powerful WATCHR features. Second, although WATCHR was adapted for use on the New York University timesharing system, it was clearly not designed for interactive use. Symbols were not kept at run time, so the user had to refer to his program in terms of absolute addresses; lengthy calling sequences were particularly cumbersome at a teletype.

Thus, in 1967 development was begun on a new debugging system, designed from the outset for conversational as well as batch use, to be invoked through a special procedural language rather than subroutine calls. Design and coding lasted through mid-1968, and distribution of the program began in the spring of 1969. Several basic requirements were established for the implementation: First, to facilitate maintenance, the same program was to be usable in both batch and interactive modes. Second, to facilitate distribution, the system had to be usable without any modification to the operating system, and have a simple input-output interface adaptable to a variety of environments.

* Currently with the Department of Physics of Columbia University, New York, New York.
† The most notable exception of which the author is aware is the debugging system recently developed for TSS/360; readers are referred to this paper for a more detailed discussion of the need for more powerful debugging systems.
†† TESTRAN, the system provided with OS/360 for debugging assembly language routines, includes several features for program checks and conditional traces; however, because the debug commands are macro calls, their format is severely restricted, and consequently test conditions which do not fall into one of several predetermined forms can be quite complicated to encode.
Third, to facilitate use, simple commands had to be provided for the most common debugging requirements.

PROGRAM ORGANIZATION

AIDS, the All-purpose Interactive Debugging System, is a main program with three input files: the object code of the user's program, the listing generated by the compilation (or assembly) of the program, and a "debug file" containing the commands to AIDS for tracing and testing the user's program. AIDS may be divided into three sections corresponding to these three files: the listing reader, which extracts from the compiler and assembler listings the attributes and addresses of the identifiers in the source program; the command translator, which transforms the statements in the debug file into entries in the AIDS trap tables; and the simulator, which simulates, monitors, and traces the user's program.

The listing reader is entirely straightforward, and only one point bears mentioning, namely, that the alternative, modifying the compiler and assembler to output the needed information, was rejected for several reasons. At the time of inception of the project, new FORTRAN compilers were being issued so often by Control Data that reimplementing such a modification on each new compiler would have been a full time effort by itself. In addition, installations with their own compilers would have had to modify them in order to use AIDS, a step many installations might have been hesitant to take.

All debugging information is supplied through a special debug language; absolutely no modifications are required to the user's program to run under AIDS. This debug language will be described in some detail below, after which a few of the techniques used in implementing AIDS will be discussed.

DEBUG LANGUAGE

The three basic syntactic entities of the debug language are the tag, the expression, and the event. The tag designates a fixed location or block of memory, and may be an octal address, statement number, variable name, or subroutine name. The expression specifies a value, and is constructed according to the same rules as a FORTRAN IV expression, including full mixed modes, and logical, relational, and arithmetic operators; only function references are excluded. The event specifies a particular occurrence in the user's program, and can take one of five forms:

- `OPCODE[S] [(opcode) [TO (opcode)]]`
- `AT [(tag list)]`
- `LOAD[S] [(FROM) (tag list)]`
- `STORE[S] [(TO) (tag list)]`
- `CALLS[S] [(tag list)]`

where `(tag list) ::= = (tag) | (tag) [, (tag)] ...`

In his debug statements, a user can refer to all the identifiers of his source program: variable and subroutine names and statement numbers. Array elements can be referenced with subscripted variables, or the entire array designated by the array name alone (the latter feature is useful, for example, in tracing stores to any element of an array). All hardware registers may be used in arithmetic expressions on an equal footing with other variables. Additional variables may be created at run time for use as counters or switches, and new labels may be assigned to points in the user's program not associated with any identifier in the source text.

The principal statement in the debug language is the trap statement, which has the form

```
{WHEN ) (event), (trap sequence)
AFTER
```

where `(trap sequence) ::= = (trap command) [, (trap command)] ...` This statement directs that immediately before or after (WHEN is synonymous with BEFORE) the occurrence of the specified event* in the simulated program, the commands in the trap sequence are to be executed. The possible trap commands are:

1. Assignment statement—exactly as in FORTRAN
2. IF (logical expression) which causes subsequent trap commands to be executed only if the logical expression is true,
3. SUSPEND {OPCODE | CONTROL | LOAD | STORE | CALL} TRAPS RESUME {OPCODE | CONTROL | LOAD | STORE | CALL} TRAPS
4. GO TO (tag) which causes a transfer of control in the simulated program,
5. PRINT (print item) [, (print item)] ... where `(print item) ::= = (tag) | "(text not including")"

which prints the contents of the tag (in a format appropriate to its mode in the user's program), or the specified text, and

* `AT (tag)` denotes the execution of the first instruction at location `(tag)`. An event without any opcode specification or tag list denotes every opcode, any store, any call, etc.
The Debugging System Aids

TAGS

\( \tag:: = (\text{subscripted variable identifier}) | (\text{statement identifier}) | (\text{global identifier}) | (\text{octal identifier}) \)

\( \text{variable identifier}:: = (\text{variable name}) [($) (\text{subprogram name})] \)

\( \text{statement identifier}:: = (\text{statement number}) [($) \text{(subprogram name)}] \)

\( \text{global identifier}:: = [$] (\text{global name}) \)

\( \text{subscripted variable identifier}:: = (\text{variable identifier}) [(\text{(subscript)}, (\text{subscript}) \ldots)] \)

EVENTS

\( \text{event specifier}:: = (\text{opcode event specifier}) | (\text{control event specifier}) | (\text{load event specifier}) | (\text{store event specifier}) | (\text{call event specifier}) \)

\( \text{opcode event specifier}:: = \text{OPCODE}[S] [(\text{opcode}) [\text{TO (opcode)}]] \)

\( \text{control event specifier}:: = \text{AT} [(\text{tag list})] \)

\( \text{load event specifier}:: = \text{LOAD}[S] [(\text{FROM (tag list)})] \)

\( \text{store event specifier}:: = \text{STORE}[S] [(\text{TO (tag list)})] \)

\( \text{call event specifier}:: = \text{CALL}[S] [(\text{tag list})] \)

\( \text{tag list}:: = (\text{tag}) [, (\text{tag}) \ldots] \)

TRAP STATEMENT

\( \text{WHEN } \text{event}, \text{trap sequence} \)

\( \text{BREAK BEFORE } \text{event}, \text{trap sequence} \)

\( \text{BREAK AFTER } \text{event}, \text{trap sequence} \)

\( \text{trap sequence}:: = (\text{trap command}) [, (\text{trap command}) \ldots] \)

TRAP COMMANDS

\( \text{left side}:: = (\text{arithmetic expression}) \)

\( \text{right side}:: = (\text{global identifier}) | (\text{variable identifier}) | (\text{octal identifier}) | (\text{arithmetic expression}) \)

\( \text{IF (logical expression)} \)

\( \text{SUSPEND (trap type)} [(\text{trap word})] \)

\( \text{RESUME (trap type)} [(\text{trap word})] \)

\( \text{where (trap type)}:: = \text{OPCODE} | \text{CONTROL} | \text{LOAD} | \text{STORE} | \text{CALL} \)

\( \text{trap word}:: = \text{OPCODE}[S] | \text{TRACE}[S] \)

PRINT (print element) [, (print element)] ...

where (print element):: = (tag) | "(text not containing")" 

COMMANDS NOT VALID IN TRAP SEQUENCES

WHAT [IS] (tag)
WHERE [IS] (tag)
$ (subprogram name)

RETREAT (decimal integer)

LABEL (octal identifier)

MAP

STEP \{ (decimal integer) \}

BREAK OUT[AT (tag)]

BREAK IN AT (tag)

* establishes new local (default) subprogram for identifiers

** finds nearest symbolic label

*** defines (assigns an address to) a symbol

Figure 1—The syntax of the AIDS debug language

(6) TRACE

which prints a line describing the event which caused the trap. Since the statement

\( \text{WHEN (event), TRACE} \)

is one of the most often used, the natural abbreviation

\( \text{TRACE (event)} \)

has been allowed. A few sample trap statements:

\( \text{TRACE STORES TO A WHEN CALL TEST, IF (I**3+J . GT . 27)} \)

\( \text{PRINT "INVALID ARGUMENTS TO TEST," I, J WHEN AT 10S, I=I+1, IF (I . GT . 100) PRINT "LOOP EXECUTED 100 TIMES, EXIT FORCED"; GO TO 100S} \)

(the S after the numbers in the last statement indicate that they are statement numbers rather than integers).

The user has at his disposal quite a few other control and informational commands; these are enumerated in Figure 1. A few of these deserve special mention:

The MAP feature provides a simple means of tracing the flow of control in his program. The MAP TRACE command makes AIDS print out pairs of addresses between which instructions were executed without any transfers. If the user does not want a continuous map, he can still get the last 25 such pairs printed at any time by typing MAP.

The user can step forward through his program, a fixed number of instructions at a time, with the command

\( \text{STEP (integer)} \)

More interestingly, with the command

\( \text{RETREAT (integer)} \)

he can step backwards through his program by a fixed number of instructions; his program is restored to exactly the same status it had earlier. Although this process is limited to a few thousand instructions, it is
The program to be debugged:

```plaintext
PROGRAM PRIME (INPUT, OUTPUT)
C PROGRAM DETERMINES IF A NUMBER IS PRIME
10 PRINT 20
20 FORMAT(*YOUR NUMBER, PLEASE-*)
READ 40, NUM
40 FORMAT(I4)
70 IF (NUM. LE. 0)
80 CALL EXIT
ISQRT = NUM ** (0.5) + .1
70 DO 90 J = 2, ISQRT
80 IF (NUM/J*J. EQ. NUM) GO TO 130
90 CONTINUE
PRINT 110
110 FORMAT(* NUMBER IS PRIME*)
GO TO 10
130 PRINT 140
140 FORMAT(* NUMBER IS NOT PRIME*)
GO TO 10
END
```

A log of the debug session:

```
TYPE PROGRAM NAME-Igo
BEHEST-when at ls, pause
BEHEST-go
PAUSE.
BEHEST-go
YOUR NUMBER, PLEASE-9
NUMBER IS PRIME
PAUSE.
BEHEST-trace stores to j
BEHEST-go
STORE TO J = 2
why didn't it try J = 3?
NUMBER IS PRIME
PAUSE.
BEHEST-what is isqrt?
check limit of do loop
1
BEHEST-after store to isqrt,
fix it
fnum = num, isqrt = fnum**
0.5 + .1
BEHEST-go
YOUR NUMBER, PLEASE-9
STORE TO J = 2
STORE TO J = 3
NUMBER IS NOT PRIME
PAUSE.
BEHEST-go
try one more
YOUR NUMBER, PLEASE-5
STORE TO J = 2
NUMBER IS PRIME
PAUSE.
BEHEST-quit
NOTE: Input from user appears above in lower case; output from AIDS appears in standard upper case; output from the user's program appears in italicized upper case. The comments on the right would not be a part of an actual debugging session.
```

Note: From the collection of the Computer History Museum (www.computerhistory.org)
the source language level, such as subscript in range and agreement in type of formal and actual parameters. However, several considerations dictated development of a system running from the object code. First, a large fraction of users have assembly language subroutines in their FORTRAN programs; running such programs interpretively would in effect mean assembling the source code and then simulating the machine instructions. Second, some of the most elusive bugs are due to compiler and system routine errors; such bugs can clearly only be found by a system which runs from the compiled code. (Interestingly enough, some of the first bugs found by AIDS were in the compiler, loader, FORTRAN coded output routine, and the routine which generates FORTRAN execution-time error messages.)

The next choice to be made is whether to simulate or execute the object code. In contrast to most debugging systems, AIDS offers the user the ability to do either. Simulation provides a far richer set of traces and checks than could a system which executes the object code; in particular, it provides a simple solution to what appears to be the most common plight of the desperate user, "What part of my program stored that?" On the other hand, when a particular routine can be isolated as the source of a program error, only that routine need be simulated, with the rest of the code executed; in this case, the program can run at nearly normal speed.

The trap system is entirely straightforward, using for each type of trap (load, store, etc.) a list of addresses which is checked regularly during simulation. To avoid possible "side-effects" (e.g., instruction modification at a location where a breakpoint is stored) absolutely no modifications are made to the user's program during simulation. Trap commands are checked syntactically and translated into an internal form on input, and are interpreted whenever a trap occurs.

Whenever a store is performed by the simulated program, the old contents of the referenced memory location are saved in a circular buffer. At two points in the circuit of the circular buffer, the contents of all the simulated hardware registers are saved. When a RETREAT is requested, the user's program is first reset to its status at one of these two earlier points; memory is restored by working backwards through the circular buffer from its current position to the earlier point. The program is then stepped forward to the point to which the user wanted to retreat in the first place.

AIDS consists of about 6000 source cards, and occupies a minimum of 44000 words of memory. With the exception of the simulation routine, which was coded in assembly language for efficiency, the entire system was written in FORTRAN. This was no doubt a factor in getting the system coded and largely debugged in less than one man-year of programming effort.

CONCLUSION

In evaluating the results of the AIDS project, it is necessary to ask two separate questions: Is such a powerful debugging system worthwhile? and Has this implementation been successful, in particular with respect to the three points mentioned towards the beginning of this paper?

The latter question I believe can be answered in the affirmative; as regards the three specific points:

1. The identical program has been used for both batch and conversational debugging. In general the system appears to be flexible enough to satisfy the debugging styles of both types of user: the selective traces and automatic program checks required by the batch user and the conditional trapping desired by the time-sharing user.*

2. In large part because most of AIDS is coded in FORTRAN, it has been converted for use under two batch and three conversational systems with relative ease. In addition, the modular design has made it possible for the author of a subsequent CDC 6600 debugging system to incorporate major sections from AIDS.\(^5\)

3. The only "abbreviation" included is the TRACE command (in place of WHEN ..., TRACE). Short of a general redesign of the command structure to reduce the amount of typing required, no other particular sequences of commands seemed to be frequent enough to merit abbreviation.

The more general question, whether such a powerful debugging system is worth the cost, is more difficult to answer. There is, of course, the increased cost in processor time and memory space, but these items generally represent only a small part of the cost of debugging; as these costs decrease further, it is safe to assume that nearly any significant saving of a programmer's time at the expense of computer time will represent a net savings.

Thus the fundamental question is, does AIDS save the programmer time in debugging? In one aspect it clearly does not: since it is such a large system, it takes quite a while to learn all its capabilities. Indeed,

* It has been suggested that the ability to jump around within the deck of commands to AIDS may be desirable to give the batch user even greater control over the debugging process; such a facility may soon be added.

From the collection of the Computer History Museum (www.computerhistory.org)
potential one-time or occasional users have been dissuaded by the thought of reading a 23-page manual. As a result, most AIDS users until now have been systems programmers or user consultants. One user has suggested, however, that it is precisely these experienced users who are most in need of such a system and for whom the system should be designed; in this case the time required to become familiar with the system is not such a critical factor.

So, finally: Does AIDS save time in the actual task of debugging, in comparison with simpler debugging systems? In the batch mode, where the primary object is to collect as much useful information as possible from each run, I am confident that the answer is yes. In conversational debugging, on the other hand, brevity and ease of typing are important factors; these aspects clearly favor the simple debugging systems, where considerable effort has been expended in this area, over the syntactically complex AIDS. It is the author's impression, however, that the few most difficult program bugs—those in which a very powerful system like AIDS can be expected to be the most help—are the ones which consume most of a programmer's time and cause most of his ulcers. In any event, a good deal more experience with the on-line use of AIDS and similar debugging systems will be required to find the best balance of brevity, simplicity, and power.

REFERENCES

1 T G EVANS D L DARLEY
   *On-line debugging techniques: A survey*
   FJCC Proceedings 1966

2 W A BERNSTEIN J T OWENS
   *Debugging in a time-sharing environment*
   FJCC Proceedings 1968

3 System/360 operating system TESTRAN
   IBM Form No C28-6648-1

4 E DRAUGHON
   *WATCHR III—A program analyzing and debugging system for the CDC 6600, user's manual*
   AEC Research and Development Report NYO-1480-58

5 H E KULSRUD
   *Helper—An interactive extensible debugging system*
   IDA—Communications Research Division Working Paper No 258

6 P T BRADY
   *Writing an on-line debugging program for the experienced user*
   CACM Vol 11 No 6 p 423 June 1968