Vol. 34 No. 3 Transactions of Information Processing Society of Japan Mar. 1993

Regular Paper

Bug Localization Based on Error-Cause-Chasing Methods

TAKAO SHIMOMURA †

In program debugging, when programmers find errors during program execution, they hypothesize the error causes and then verify their hypotheses. By repeating this process, they trace error causes one by one and eventually find a bug, that is, the fundamental cause of an error. This paper first classifies errors commonly found during program execution into five types: variable-value errors, allocation errors, control-flow errors, omission errors and functional errors. It then presents error-cause-chasing methods that chase the causes of each type of error. These methods help programmers to examine error causes and eventually locate bugs. The paper also describes a bug-locating assistant system CHASE that has been developed based on these error-cause-chasing methods. This system identifies the locations at which these errors are caused and restores the program state there. It can further localize the error causes by analyzing the common cause of multiple variable-value errors, and for omission errors, by chasing the conditional statements that might have caused the errors through the use of path analysis.

1. Introduction

(1) Survey of Program Debugging Techniques

Various kinds of program debugging techniques have been developed, and they can be classified into the following five categories.

Breakpoint Method. With symbolic debuggers, programmers first set breakpoints at the locations that they think are related to errors, then execute the program and examine the program state when the execution is interrupted. They locate a bug by repeating this process. OMEGA[1] can easily set breakpoints that have complicated conditions by using a query language as used in relational database systems. Path Rules[2] can detect erroneous function invocation sequences based on specified assertions with respect to execution paths. Dalek and the like[3)-5)] can easily detect the occurrences of specific events. Dbxtool[6] displays the source text of a program being debugged and indicates the current execution line; it also displays the values of specified variables every time execution is interrupted. VIPS[7),8)] further facilitates debugging by displaying complicated data structures as figures.

Algorithmic Debugging. Programmers judge the correctness of executed functions one by one. They judge whether a function was executed correctly or not according to whether its output values are correct or not for its input values. If a function is judged to be incorrect, programmers examine the functions that it invoked. If a function is judged to be correct, they examine that function's sibling functions. In **Fig. 1**, when function Q2 is incorrect and its child functions R1 and R2 are both correct, we can see that there must be an error in the calling sequence with which function Q2 invoked its child functions. With algorithmic debugging, programmers do not need to determine which function to examine next. The system optimally orders the sequence in which programmers are shown the function to be examined and its input and output values. To locate a bug, programmers have only to repeatedly judge the correctness of each function. This technique is applied to functional languages that do not have side effects[9),10)].

Program Dependency Analysis. This method can localize the part of a program that should be examined by extracting the part of the source program that might be related to errors. There are two different types to this method, one which analyzes programs statically and one which analyzes them dynamically. Static slicing[11),12)] extracts the statements that might affect the value of a variable at a certain statement, and dynamic slicing[13)-15)] extracts the executed statements that might have affected the value of a variable at a certain statement. In **Fig. 2**, variable A is defined at statements S1 and S3, and its value is used at statement S6. Statement S2 is a conditional statement like if-then-else. In this case,

† NTT Software Laboratories

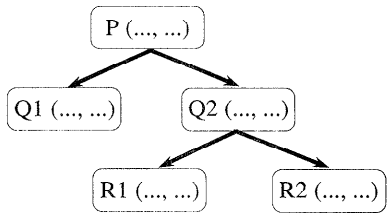


Fig. 1 Algorithmic debugging.

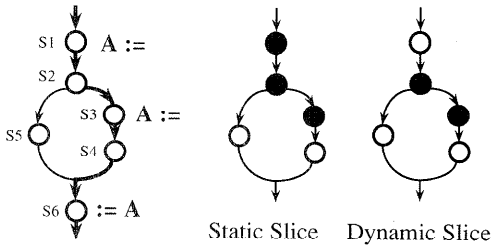


Fig. 2 Program-dependency analysis.

static slicing extracts all the statements (S1 and S3) that define variable A and the conditional statements (S2) that determine whether those defining statements are executed. If we assume that the program was executed along the path represented by thick arrows, dynamic slicing extracts only the statement (S3) that defined variable A such that its defined value was used at statement S6. Thus, statement S1 is not included in the dynamic slice. With program dependency analysis, programmers locate a bug by examining these localized statements.

Execution Replay. By recording the results of executing each statement, when an error is found, this method can examine previous program states without rerunning a program. This replay method has a problem, however, in that the size of the execution history increases in proportion to the number of the dynamic steps. EXDAMS[16] does not require the statement numbers of executed statements to be output to the execution history file. When it traverses an execution path, it determines which statement is to be traversed next by referring to both the control structure of the program and the results of predicate statements. Backtracking[17] can reduce the execution history by maintaining only the result of the last execution of each statement regardless of how many times it may be executed.

Anomaly Detection. This method discloses some errors by statically or dynamically analyzing a program. AIDA[18] executes a program and automatically detects data-flow errors like "reference before setting." Dynamic analysis also allows analysis of the data flow for array elements. By static analysis of a program, Cecil[19] can detect anomalies the kinds of which are specified in advance.

(2) Problems with Current Debugging Techniques

Program dependency analysis tells us which part of a program should be examined. In algorithmic debugging, it is only necessary to tell the system whether or not the execution of a given function was correct for bugs to be detected. Symbolic debuggers like VIPS can help us find any difficult bug. However, as programs become bigger and bigger, they impose a heavier load on programmers. With program dependency analysis, if a program is very large, the part of the program that should be examined will still be large. Algorithmic debugging cannot be applied to large programs because they usually have side effects. If a function refers to some data other than its parameters, it is impossible to judge the correctness of the function only from is parameter values. With symbolic debuggers, programmers have to set breakpoints at the locations that they think are related to errors, but it becomes very difficult to find these locations if the program is large. Although anomaly detection can reveal various kinds of internal contradictions, it never helps us find bugs, that is, the fundamental causes of such errors. Execution replay can automatically restore previous program states. However, existing systems using this method do not have a strategy for locating bugs.

When programmers find a failure in the execution of a program, they hypothesize the causes of the failure and verify these hypotheses[20]. By repeating this process, they trace error causes one by one and eventually find the bug that is the basic cause. Experienced programmers may be able to notice error causes close to the bug merely by glacing at an error state (See **Fig. 3**). However, the bigger programs become, the longer the procedure to trace error causes to locate a bug will be. To reduce this procedure, this paper presents error-cause-chasing methods that help programmers to trace error causes by auto-

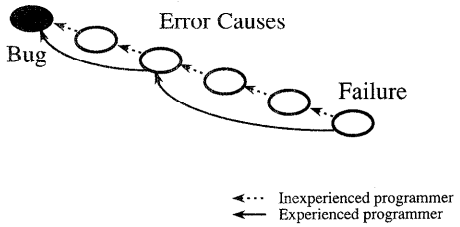


Fig. 3 Approach.

Table 1 Common errors.

Errors	Types
Variable-value Errors	D
Allocation Errors	D
Control-flow Errors	C
Omission Errors	C
Functional Errors	D, C

D : Data errors, C : Control errors.

matically identifying the locations at which errors are caused and restoring the program state there. Errors commonly found during program execution can be classified into five types: variable-value errors, allocation errors, control-flow errors, omission errors and functional errors. The error-cause-chasing methods presented here can chase the causes of each type of error, and can further localize the error causes by analyzing the common cause of multiple variable-value errors, and for omission errors, by chasing the conditional statements that might have caused the errors through the use of path analysis. This paper also describes a bug-locating assistant system CHASE that has been developed based on these error-cause-chasing methods.

2. Error-Cause-Chasing Methods

(1) Classification of Errors

Let us first consider what kinds of errors are found during program debugging. Many common errors concerning program states can be roughly classified as data errors or control errors (See **Table 1**). *Variable-value errors* are those in which a variable has an erroneous value. There are also *allocation errors* concerning dynamically allocated data. These errors include cases in which data that should not have been created is created, and cases in which data of the same type is duplicated and consequently the amount of data is greater than expected. We often notice such allocation errors when we use the visual debugger VIPS[7),8)], which can display linked lists as figures. In **Fig. 4**, the linked list displayed in the upper list window is correct, but the linked list actually created is the one displayed in the lower list window, which contains two superfluous nodes.

Control-flow errors refer to cases in which the flow of control is erroneous. One specific case of such control errors is *omission errors*, in which a certain input or output process that should have been executed was not executed. Examples are when a message has not been displayed, or a result has not been output. In addition to data and control errors, *functional errors* are often found during program bebugging. Functional errors refer to cases in which a certain process was executed incorrectly. These errors are caused by a combination of data errors and control errors. Examples are when a correct message is not displayed or the location of a displayed item is erroneous.

Data errors and control errors cause failures during program execution. These errors are caused by other data errors and control errors, which are ultimately caused by a bug, e. g., an erroneously written statement. The kinds of errors that can be found during program execution depend on the techniques that display program execution. For example, allocation errors can be found easily by VIPS because it has a facility that displays linked lists. For another example, when a value is assigned to a variable, VIPS shows an arrow from referenced variables to updated variables. This makes it easier to notice when some variables are referenced or updated erroneously. Although these kinds of errors do not cover everything, when we debug a program, we find that they occur very often. The bug-locating assistant system CHASE provides a facility to identify the locations at which these kinds of errors are caused and automatically restore the program state there.

(2) Bug-Locating Assistant Facilities

There are three separate bug-locating assistant facilities. *The error-cause-chasing facility* identifies the location at which an error is caused; *the program-state-display facility* displays the program state there, i. e., variable values at that point and function call/return

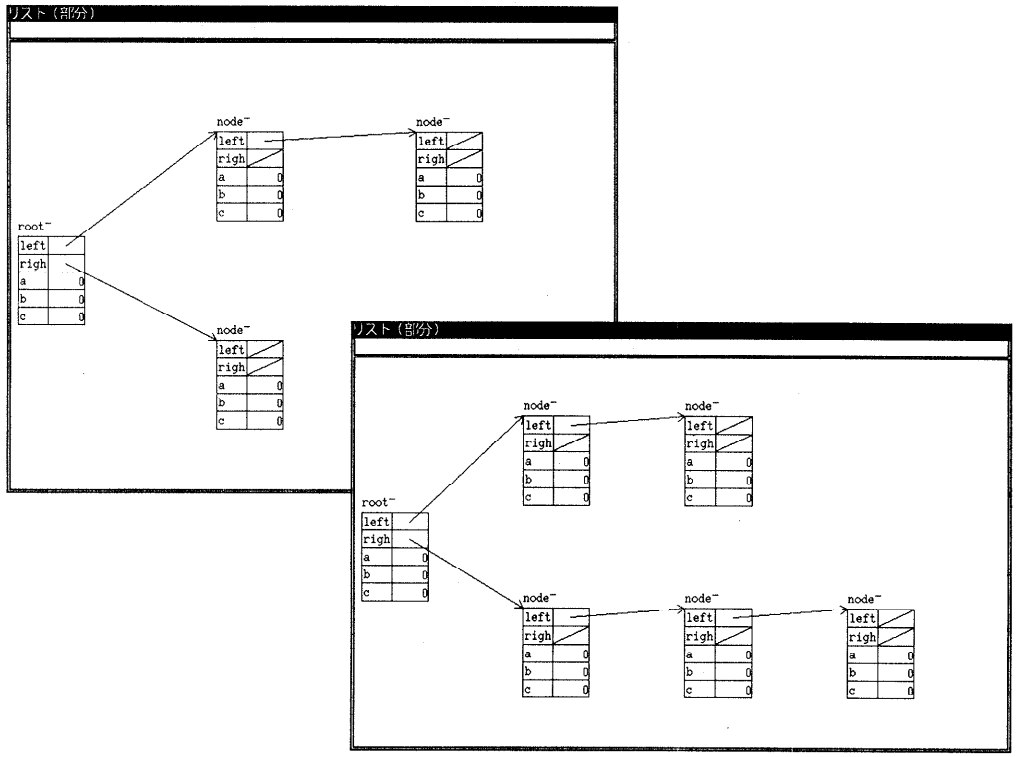


Fig. 4 Example of allocation error.

sequences or changes in variable values in its neighborhood; and *the execution-point-move facility* moves the current execution point forward or backward by statements or functions. This paper mainly describes the error-cause-chasing facility; the details of the other facilities have previously been described in a separate paper[21]. The error-cause-chasing facility consists of:

• Variable-Value-Error Chasing—When a variable has an erroneous value, this facility locates the statement that created the erroneous value. As a part of this facility, if multiple variables have erroneous values, it locates the common cause of the multiple variable-value errors. We call this *the variable-value common-error-chasing facility*.

First, let us consider the data flow related to the creation of an erroneous value. In **Fig. 5**, when the value of variable V1 is incorrect, part P1 is the data flow that created this erroneous value. Any statement in this data flow might include a bug. In actual debugging, when we examine a program state at a certain execution

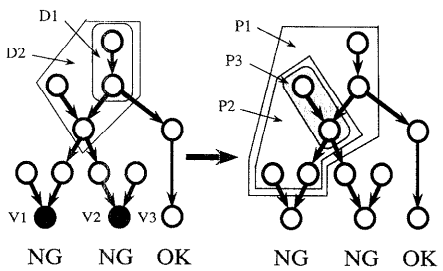


Fig. 5 Variable-value-error chasing.

point, we often notice some variables have incorrect values but some others have correct values. In such a case, we can further localize the part of the data flow that might have affected the erroneous value. Sub-data flow D1 that created correct values can be considered to have a low probability of having affected the erroneous value. If we remove that part, the part of the data flow that should be examined will be reduced to part P2. Sub-data flow D2 that created multiple incorrect values can be considered to have a high probability of including an error. If we extract such a

part, the part of the data flow that should be examined will finally be reduced only to part P3.

• Allocation-Error Chasing—This facility identifies the location at which data was allocated erroneously and restores the program state there. In the example of Fig. 4, the system identifies the location at which the superfluous nodes were allocated and automatically restores the program state there.

• Control-Flow-Error Chasing—When the control flow of a program is erroneous, this facility lets programmers trace backward, in turn, the locations of the conditional statements or function invocation statements that caused the current statement to be executed. In **Fig. 6**, we can examine one by one the shaded conditional and function-invocation statements that caused the current statement to be executed.

• Omission-Error Chasing—We assume that programmers find an omission error by identifying a normal input or output immediately before the omitted input or output. When they specify the input or output statement (or the function-invocation statement that executes input or output) that control was not transferred to, this facility finds the locations at which conditional statements were executed such that those conditional statements might have prevented control from transferring to the omitted statement. Exactly speaking, it shows the last execution point at which a conditional statement was executed such that there exists an unexecuted path not including any input or output from that conditional statement to the omitted statement.

For example, in **Fig. 7**, there are two statements, C1 and C2, that invoke the omitted part. We can see that the two conditional statements S1 and S2 prevented control from transferring to these two function invocation statements. In this example, we can further reduce the two conditional statements that should be examined to one. That is, we need only consider the conditional statement whose path to the omitted part does not include any inputs or outputs. This is because if the path to the omitted part included inputs or outputs, these inputs or outputs would also be omitted. In this case, therefore, the conditional statement S2 could never have caused the omission errors in question. Finally, we can see that conditional statement S1 has a high probability of including an error.

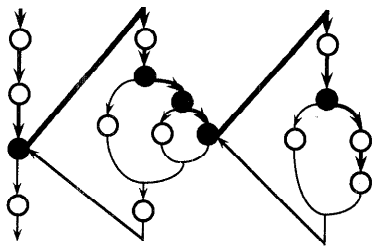


Fig. 6 Control-flow-error chasing.

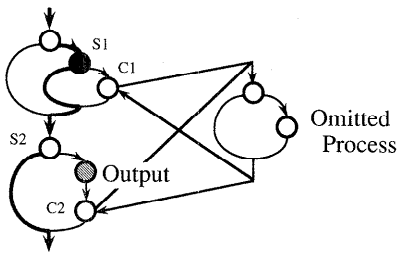


Fig. 7 Omission-error chasing.

• Functional-Error Chasing—When a certain process was executed incorrectly, this facility identifies the location at which that process was executed and restores the program state there. The process to be examined is specified in one of three ways: by function names, line numbers, or event numbers. Event numbers identify the events, such as function invocations or changes in variable values, displayed by the program-state-display facility.

(3) Cooperation between CHASE and programmers

During a debugging process, it is programmers who recognize what type of error occurs at the current execution point and determine which error-cause-chasing facility to be used to examine its cause based on the error state. CHASE executes the error-cause-chasing facility that the programmers selected, finds the location at which that errors is caused, and shows the program state (the executed statement, variable values, function-invocation sequence, and so on) there.

For example, consider a case in which two variables A and B have incorrect values and an output X is omitted. First, a programmer notices that these errors occurred. Then, based on this error state, he or she 1) assumes that it is due to the erroneous value of variable A that

this error state was caused and decides first of all to examine the cause of the variable-value error concerning A; or 2) tries to examine the common cause of the variable-value errors concerning A and B; or 3) first examines the cause of the omitted output X. A fourth possibility is that, 4) if the programmer is very familiar with how the program works, he or she might intuitively think this error state is due to a function P and might try to directly examine the process of function P using the functional-error-chasing facility.

3. Implementation

(1) System Configuration

Figure 8 shows the system configuration. The bug-locating assistant system consists of two subsystems: VIPS[7] and CHASE. VIPS records execution history while executing an Ada program. CHASE localizes error causes by analyzing the execution history.

Figure 9 shows an example of a CHASE screen. The CHASE system has a program-text window, a panel window, and a monitor window. The program-text window displays the current execution point, the panel window allows programmers to enter commands through menus or buttons, and the monitor window displays variable values and various events.

(2) Execution History

The execution history this system requires consists of the following items concerning the execution of each statement.

- Executed statements—line number and statement type,
- Function invocations—function identifiers,
- Assignments — variable identifiers and defined values, and
- Dynamic allocations—allocation addresses and type identifiers of the allocated data.

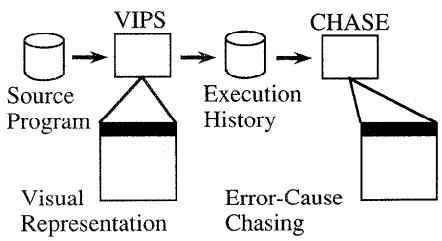


Fig. 8 System configuration.

4. Error-Cause-Chasing Algorithms

Algorithms for implementing some of the main error-cause-chasing facilities are described below.

(1) Program-State Restoring

Because the purpose of this system is to chase error causes, unlike Backtracking[22], it does not need to restore the whole program state by backtracking the execution step by step. By analyzing only the necessary part of the execution history, it obtains execution points that caused an error and variable values at those points.

The execution point where the value of a variable was last updated can be acquired in the following way.

Step 1. Acquire the IDs of the variable and the block in which the variable is declared.

Step 1.1 Acquire the active block ID from the execution history based on the current execution point.

Step 1.2 Acquire the IDs of the variable, its

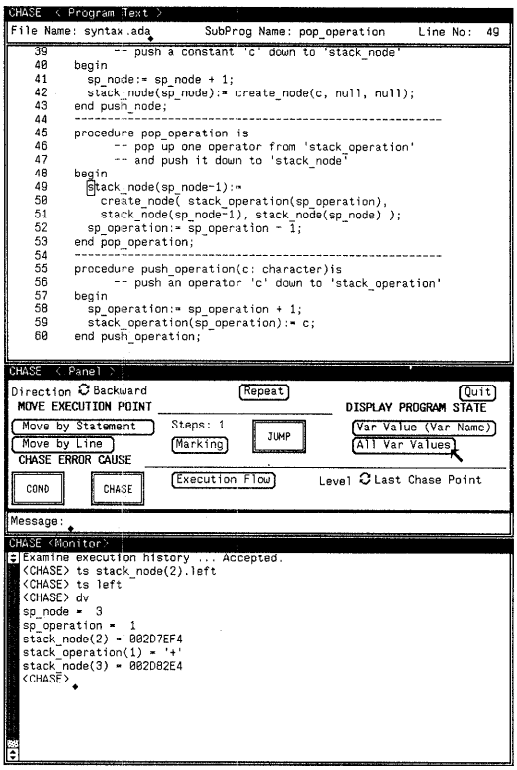


Fig. 9 CHASE screen; Error-cause chasing (4).

type, and the block in which it is declared using the active block ID and variable name. If the specified variable is pointer qualified, the value of the pointer will be acquired as the variable ID.

Step 1.3 If the variable is an element of either a record or an array, acquire its element ID from its type definition.

Step 2. Acquire the execution point at which the variable was last updated (that is, at which the variable was included in the updated variable) by examining backward the updated data sequence in the execution history as long as the block is active. If such an execution point is not found, the value of the variable will be undefined.

The value of a variable at the current execution point can be acquired in the following way.

Step 1. Acquire the IDs of the variable, its type, and the block in which it is declared.

Step 2. Acquire the execution point at which the variable was updated (that is, at which the variable was included by the updated variable).

Step 3. By comparing the variable ID with the updated variable ID, acquire the offset of the variable value from the top of the updated variable value.

Step 4. Using the offset, acquire the variable value from the updated variable value and then convert this to a literal expression using its type definition.

In the example shown in **Fig. 10**, we will obtain the value of the element "Q.a", that is, element "a" pointed to by pointer "Q" at line 100. The system first obtains the currently executed function from the current execution point and determines which variable name Q refers to. It then obtains the value of pointer "Q". It will find the location at which pointer Q" was defined by analyzing the execution history backward and then obtain the value of "Q", "addressl". Next it obtains the value of element "a" of the data allocated at addressl. Similarly, it will find the location at which the data at addressl was defined and finally obtain the value of "Q.a".

(2) Common-Error Chasing

Consider the case in which at an execution point t, variables X and Y have incorrect values and a variable Z has a correct value. Appendix 1 shows the algorithm for acquiring the last point among the execution points that might be the common cause of these errors.

Depend(j, v) denotes a set of the execution points on which execution point j is control- or data-dependent with respect to variable v. Execution point j is control-dependent on an execution point i if the statement executed at i is a conditional statement and the execution of that statement caused control to transfer to execution point j. Execution point j is data-dependent on an execution point i with respect to variable v if variable v was used at j and that used value was defined at i. Depend(j) denotes Depend$(j$, Use$(j))$, where Use(j) is a set of variables used at j. Last(P) denotes the last executed point among a set of execution points P. Execution points are numbered 1, 2, 3, ⋯ in the order they were executed. If P is empty, last (P) denotes 0. Get-last(P) takes the last executed point out of a set of execution points P and returns it.

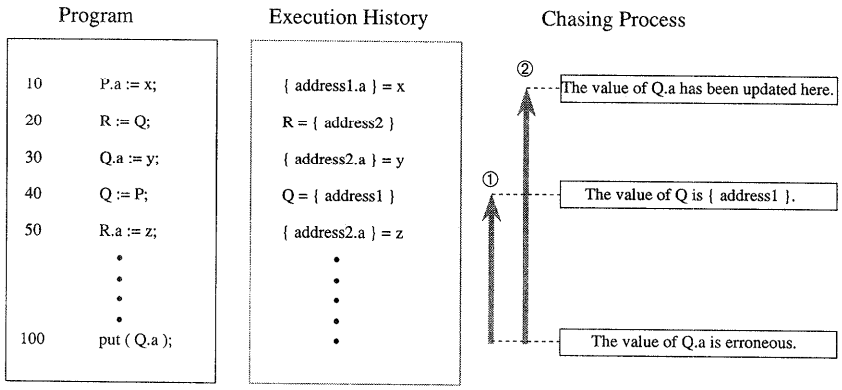


Fig. 10 Algorithm for obtaining variable values.

(3) Omission-Error Chasing

Appendix 2 shows the algorithm for examining whether or not there exists a path not including any input or output from a given conditional statement to an omitted statement s in a function Q. For a given block B (which means a function, each branch of a conditional statement, or statements inside a loop statement), ProcessBlk (B) returns false if there exists a path not including any input or output from the entry of B to the exit of B. StartStm(B) returns the first statement in block B. NextStm(c) returns the statement next to statement c.

For each function P, the value of ProcessBlk (P) can be obtained in advance. In addition, for each branch B of a conditional statement, a set of the functions that can be invoked inside B before executing input or output can also be obtained in advance. Therefore, we have only to examine whether or not there exists a path not including any input or output from the entry of function Q (or from a conditional statement of interest if it is included in Q) to statement s.

5. Examples

Figures 11 to 15 show an example of tracing a data flow to locate a bug. We use the program *interpreter* as an example program. Interpreter accepts a numerical expression, and then creates a syntax tree representing the expression. It finally calculates the expression's value based on the syntax tree.

1) An expression "1+2*3−4/5" is entered and its syntax tree is created. VIPS shows the linked list of the syntax tree, but it is incorrect because the left pointer of the addition operator is connected with the multiplication operator instead of the number "1" (Fig. 11).

2) We run the CHASE system, and from its program-text window, we can see that the subprogram "parser" has just been executed (Fig. 12).

3) We first chase the execution point where the left pointer of the addition operator was updated. The left pointer is found to have been updated at line 33 in the subprogram "create_node" (Fig. 13). We further chase the variable "left".

4) The variable "left" is found to have been set at line 49 in the subprogram "pop_operation". We display the values of all variables referred to

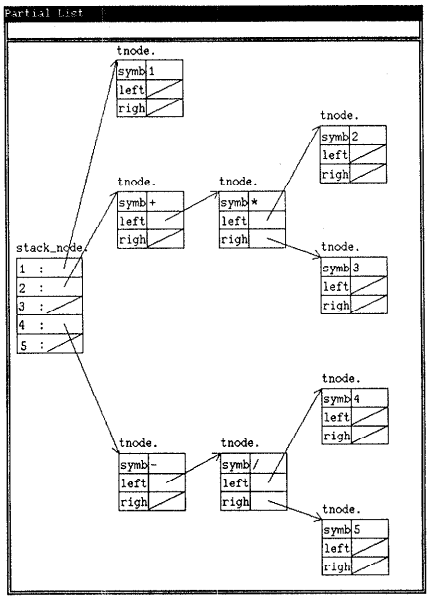


Fig. 11 Error-cause chasing (1).

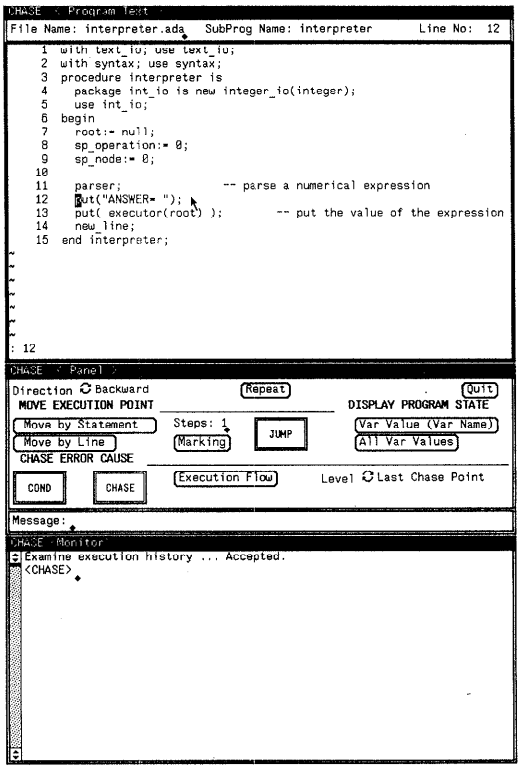


Fig. 12 Error-cause chasing (2).

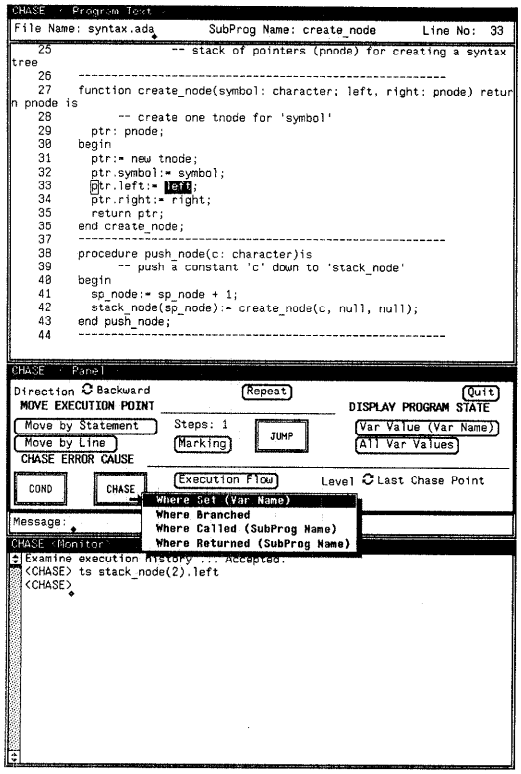


Fig. 13 Error-cause chasing (3).

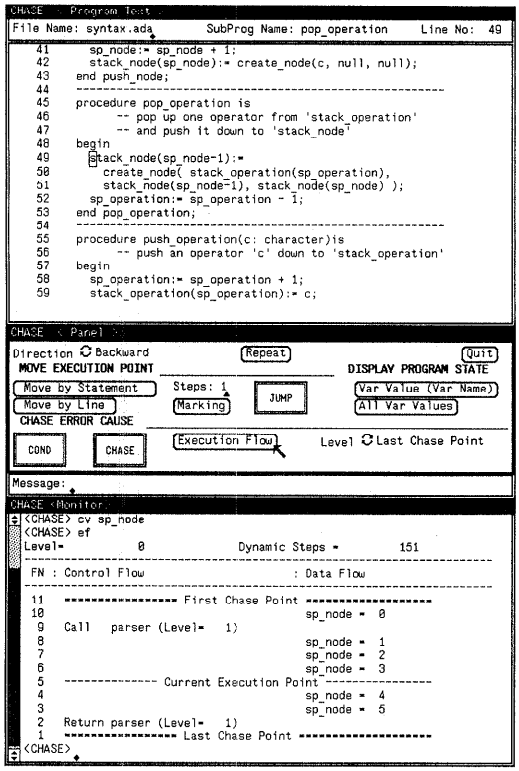


Fig. 14 Error-cause chasing (5).

in this statement (Fig. 9). As a result, a programmer will notice that the value of a stack pointer is erroneous and that this might have caused the error.

5) We will then examine the values of the stack pointer in the neighborhood of the current execution point. As expected, we find that the value of the stack pointer has never been decreased (Fig. 14).

6. Observation

To test the effectiveness of the error-cause-chasing method, we performed a debugging experiment. We used CHASE and a.db[23], which is a screen-based symbolic debugger for Ada programs with a source-text-display facility like dbxtool[6] for C programs. Two kinds of programs were prepared. Both programs are written in Ada and include about 200 lines of code. One analyzes numerical expressions by creating a syntax tree. The other handles a binary tree to manage a stock of goods. Several bugs were buried, one by one, in each program.

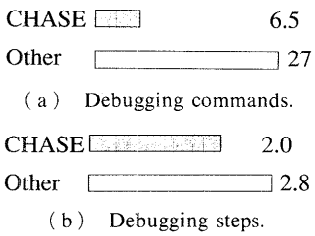


Fig. 15 Result of experiment.

Figure 15 shows the result of this debugging experiment. To detect a bug with CHASE required only about one-fourth as many debugging commands. In debugging, programmers repeatedly hypothesize error causes and then verify them by examining program states. If we regard this hypothesize-verify process as one debugging step, the CHASE system also reduced the number of such debugging steps, though not as dramatically. The author thinks that these improvements are due to the CHASE system saving the procedure of finding the location at which an error is caused and then executing the

program to that location.

7. Conclusion

A bug-locating assistant system CHASE and its error-cause-chasing methods have been described. CHASE facilitates program debugging by automatically chasing the program part that caused an error found during execution, and by displaying the program state there for investigation. A preliminary experiment shows that this system reduces the number of debugging commands by about three-fourths and the number of debugging steps by about one-third.

Because the current version of CHASE is a prototype and records almost all execution history, it takes rather much time to record and analyze this history. As a consequence, this system can presently be applied only to small programs. The author, however, feels that it is still an effective way to automate the process of chasing error causes. The next version of CHASE will implement controlled recording, whereby the recording of both variables and sampling points will be controlled.

Acknowledgements The author is grateful to Dr. Alan M. Davis at the University of Colorado and Dr. Stewart N. Weiss at Hunter College of the City University of New York for their kind advice. He would also like to thank Dr. Seishiro Tsuruho, Executive Manager of NTT Software Laboratories, for his continuous encouragement.

References

1) Powell, M. L. and Linton, M. A.: A Database Model of Debugging, *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pp. 67-70 (Mar. 1983).
2) Bruegge, B. and Hibbard, P.: Generalized Path Expressions: A High-Level Debugging Mechanism, *The Journal of Systems and Software*, Vol. 3, pp. 265-276 (1983).
3) Lazzerini, B. and Lopriore, L.: Abstraction Mechanisms for Event Control in Program Degugging, *IEEE Trans. Softw. Eng.*, Vol. 15, No. 7, pp. 890-901 (1989).
4) Olsson, R. A., Crawford, R. H. and Ho, W. W.: A Dataflow Approach to Event-based Debugging, *Softw. Pract. Exper.*, Vol. 21, No. 2, pp. 209-229 (1991).
5) Olsson, R. A., Crawford, R. H., Ho, W. W. and Wee, C. E.: Sequential Debugging at a High Level of Abstraction, *IEEE Software*, Vol. 8, No. 3, pp. 27-36 (May 1991).
6) Adams, E. and Muchnick, S. S.: Dbxtool: A Window-Based Symbolic Debugger for Sun Workstations, *Softw. Pract. Exper.*, Vol. 16, No. 17, pp. 653-669 (July 1986).
7) Isoda, S., Shimomura, T. and Ono, Y.: VIPS: A Visual Debugger, *IEEE Software*, Vol. 4, No. 3, pp. 8-19 (1987).
8) Shimomura, T. and Isoda, S.: Linked-List Visualization for Debugging, *IEEE Software*, Vol. 8, No. 3, pp. 44-51 (May 1991).
9) Shapiro, E. Y.: *Algorithmic Program Debugging*, The MIT Press (1982).
10) Takahashi, N. and Ono, S.: DDS: A Declarative Debugging System for Functional Programs, *The Transactions of the Institute of Electronics, Information and Communication Engineers, Japan*, Vol. J72-D-I, No. 11, pp. 779-788 (1989).
11) Weiser, M.: Program Slicing, *IEEE Trans. Softw. Eng.*, Vol. SE-10, No. 4, pp. 352-357 (1984).
12) Horwitz, S., Reps, T. and Binkley D.: Interprocedural Slicing Using Dependence Graphs, *ACM Trans. Prog. Lang. Syst.*, Vol. 12, No. 1, pp. 26-60 (1990).
13) Korel, B. and Laski, J.: Dynamic Program Slicing, *Inf. Process. Lett.*, Vol. 29, No. 10, pp. 155-163 (1988).
14) Korel, B. and Laski, J.: Dynamic Slicing of Computer Programs, *J. Syst. Softw.*, Vol. 13, pp. 187-195 (1990).
15) Agrawal, H. and Horgan, J. R.: Dynamic Program Slicing, *ACM SIGPLAN Notices*, Vol. 25, No. 6, pp. 246-256 (1990).
16) Balzer, R. M.: EXDAMS — Extendable Debugging and Monitoring System, *AFIPS Proceedings, Spring Joint Computer Conference*, pp. 567-580 (1969).
17) Agrawal, H. and Spafford, E. H.: An Execution Backtracking Approach to Program Debugging, *Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference*, pp. 283-299 (Sep. 1988).
18) Chan, F. T. and Chen, T. Y.: AIDA—A Dynamic Data Flow Anomaly Detection System for Pascal Programs, *Softw. Pract. Exper.*, Vol. 17, No. 3, pp. 227-239 (Mar. 1987).
19) Olender, K. M. and Osterweil, L. J.: Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation, *IEEE Trans. Softw. Eng.*, Vol. 16, No. 3, pp. 268-280 (1990).
20) Araki, K., Furukawa, Z. and Cheng, J.: A

General Framework for Debugging, *IEEE Software*, pp. 14-20 (May 1991).
21) Shimomura, T. and Isoda, S.: CHASE: A Bug-Locating Assistant System, *COMPSAC'91*, pp. 412-417 (Sep. 1991).
22) Agrawal, H., DeMillo, R. A. and Spafford, E. H.: An Execution-Backtracking Approach to Debugging, *IEEE Software*, pp. 21-26 (May 1991).
23) VERDIX Ada Development System: Debugger Manual, VERDIX Corporation, Virginia (1989).

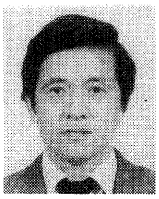
Appendix 1 Common-Error Chasing

```
Old X:= Depend(t, X);Old Y:= Depend(t, Y);
Old Z;=Depend(t, Z);
Old X:=Old X−Old Z;Old Z:=Old Y−Old Z;
New X:= Old X; New Y:= Old Y; New Z:=
  Old Z;
while (Old X ≠ φ and Old Y ≠ φ) loop
  Com:=Old X ∩ Old Y;   —the common cause
                          of the variable-
                          value errors con-
                          cerning X and Y
  if Com ≠ φ then
    Remove XY:=Com;
    New Z:= Depend(New Z);Old Z:= Old Z
      ∪New Z;
    while Com ≠ φ loop
      s:=get_last(Com);
      Correct:=false;
      Next Z:=Old Z;
      while s ≦ last(Next Z) loop
        if s∈Next Z then   —s is included in
                            the correct data-
                            and/or control-
                            flow
          Correct:−true;
          exit;
        end if;
        New Z:= Depend(New Z);Old Z:=
          Old Z∪New Z;
        Next Z:=New Z;
      end loop;
      if not Correct then
        return s;          —Found
      end if;
    end loop;
    Old X:= Old X − Remove XY;Old Y:=
      Old Y − Remove XY;
    New X:= New X − Remove XY; New Y:=
      New Y − Remove XY;
  end if;
  New X:=Depend(New X);New Y:=
    Depend(New Y);
  Remove X:={r∈ Old X | r > last(New Y)};
    Remove Y:={r∈Old Y | r > last(New X)};
  Old X:= (Old X − Remove X) ∪New X;
    Old Y:= (Old Y − Remove Y) ∪New Y;
  Remove Z:={r∈Old Z | r >
    max(last(New X), last(New Y))};
  Old Z:=Old Z − Remove Z;
  Old X:= Old X − Old Z;Old Y;= Old Y
    −Old Z;
end loop;
return 0;
                          —Not found
```

Appendix 2 Omission-Error Chasing

```
function ProcessBlk(B: block) return boolean is
  IOexist: boolean;
  c: statement;
begin
  if block B has been recursively invoked then
    return true;
  end if;
  IOexist:=false;
  c:=StartStm(B);
  while c≠null and IOexist=false loop
    if c=s then
      There exists a path not including any
        input or output fom entry of B to s.
    elsif c=assignment statement then
      null;
    elsif c=input or output statement then
      IOexist:=true;
    elsif c=conditional statement then
      IOexist:−ProcessBlk(then branch) and
        ProcessBlk(else branch);
    elsif c=while-loop statement then
      ProcessBlk(while-loop block);
    elsif c=function-invocation statement then
      IOexist:=ProcessBlk(invoked function);
    end if;
    c:=NextStm(c);
  end loop;
  return IOexist;
end;
```

(Received September 18, 1992)
(Accepted January 19, 1993)



Takao Shimomura is a senior research engineer at NTT Software Laboratories. Since April, 1992, he has been a guest associate professor at the Graduate School of Information Systems of the University of Electro-Communications, His research interests include software design automation, program visualization, and automated debugging. He was born in Shizuoka, Japan in 1949. He received a BS in mathematics from Kyoto University in 1973 and an MS from Tohoku University in 1975. He is a member of IPS Japan, IEICE Japan and ACM.