Efficient Debugging with Slicing and Backtracking*

SERC-TR-80-P

Hiralal Agrawal
Richard A. DeMillo
Eugene H. Spafford

Software Engineering Research Center
Department of Computer Sciences
Purdue University
W. Lafayette, IN 47907-2004

Abstract

Programmers spend considerable time debugging code. Several tools are available to help them in this task, varying from hexadecimal dumps of program state to window- and mouse-based interactive debuggers, but the task still remains complex and difficult. Most of these conventional debuggers provide breakpoints and traces as their main debugging aids. These features have changed little in the past 15–20 years desp}te the increased complexity of software systems on which debuggers are used.

In this paper we present a prototype tool that enables users to follow their "natural" thought process while debugging. It combines dynamic program slicing and execution backtracking techniques in a novel way. With the help of this powerful combination, program bugs may be localized and removed very quickly. Examples are given to show how our debugger may be used, and how it relates to other research in the area.

Keywords: program debugging, execution backtracking, reverse program execution, program slicing, dynamic program slicing

1 Introduction

The importance of good debugging tools cannot be overemphasized. Average programmers may spend considerable amounts (possibly more than 50%) of their program development time debugging. Several tools are available to help them in this task [AS89, MH89], varying from hexadecimal dumps of program state at the time of failure to window- and mouse-based interactive debuggers using bit-mapped displays [AM86, Car86]. Most interactive debuggers provide breakpoints and traces as their main debugging aids

*This paper has been submitted to *Software—Practice & Experience*.

This work was funded by a grant from the Purdue University/University of Florida Software Engineering Research Center (SERC), and by National Science Foundation grant 8910306-CCR.

[Kat79, MB79, Dun86, Wei82]. Unfortunately, these traditional mechanisms are often inadequate for the task of isolating specific program faults.

In this paper we present the results of our preliminary experiment in integrating new approaches to software fault localization with conventional debugging techniques. This experiment has resulted in a novel, easy-to-use, interactive, window-based debugger that supports new methods of execution backtracking and static- and dynamic program slicing along with conventional techniques. We believe these mechanisms are significant additions to the mechanisms already used by programmers, and provide a natural approach to isolating and identifying software faults ("bugs").

How Does One Debug?

Given that a program has failed to produce the desired output, how does one go about finding where it went wrong? Other than the program source, the only important information usually available to the programmer is the input data and the erroneous output produced by the program. If the program is sufficiently simple, it can be analyzed manually on the given input. However, for many programs, especially lengthy ones, such analysis is much too difficult to perform. One logical way to proceed in such situations would be to *think backwards*—deduce the conditions under which the program produces the incorrect output [Sch71, Gou74, Luk80].

Consider, for example, the program in the main window panel of Figure 1.[1] This program computes the sum of the areas of N triangles. It reads the value of N, followed by the lengths of the three sides of each of these N triangles. From these values, it classifies each triangle as an equilateral, isosceles, right, or a scalene triangle. Then it computes the area of the triangle using an appropriate formula. Finally, the program prints the sum of the areas.

There is a bug in the displayed program. The C code at line 26 is doing a comparison, but the second comparison has been mistyped as using a = symbol instead of a == symbol. This is a common error when programming in C, and results in an assignment rather than a non-destructive comparison.

Suppose this program is executed for the testcase[2] when $N = 2$ and sides of the two triangles are (5, 4, 3) and (4, 4, 2) respectively.[3] If the final sum of areas printed is incorrect, how should we go about locating the bug in the program? Looking backwards from the **printf** statement on line 46, we find there are several possibilities: `sum` is not being updated properly; one or more of the formulas for computing the area of a triangle are incorrect; the triangle is being classified incorrectly; or the values for the three sides of the triangle are not being read correctly.

The statement on line 43 adds `area` to `sum`, so the first thing we may want to do is examine the state of the program at that point. We can set a breakpoint at that line and reexecute the program up to that statement to examine the values of variables `sum` and `area` at that point.

Suppose we find that the area is computed correctly during the first loop iteration, but is computed incorrectly during the second iteration. To discover why, we might wish to examine the value of `class` for the second triangle to determine which formula for computing its area was used. If we find `class` to be incorrect, we can examine the values of the three sides of the triangle to check if they are being read

[1]The figures are X Window System window dumps of our debugging tool in operation.
[2]A testcase consists of a specific set of runtime input values.
[3]We will refer to this testcase as testcase #1 in later sections.

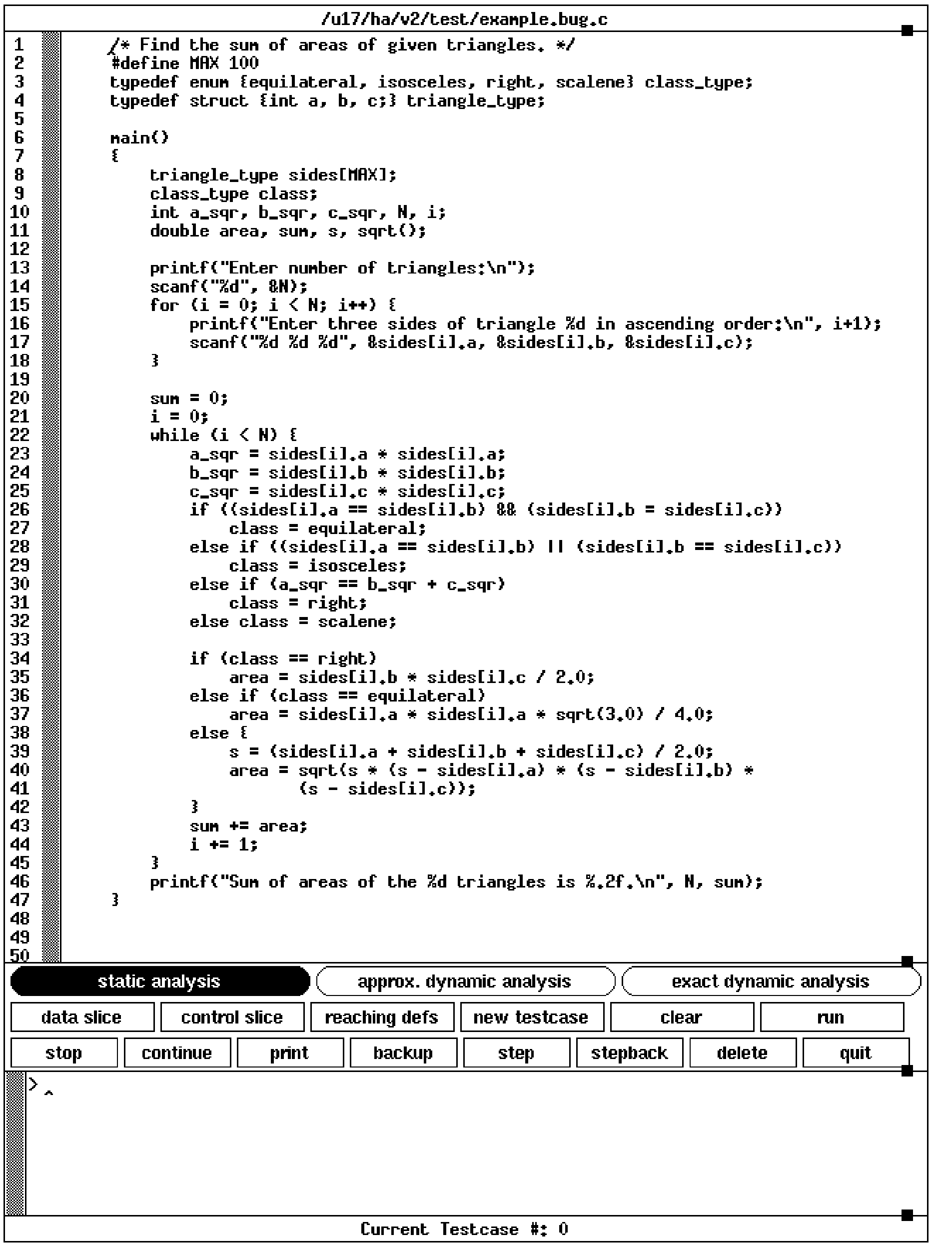

Figure 1: Tool screen with a sample C source program

correctly. If so, we should examine the statements on lines 23–32 that determine the class of the triangle.

There are three distinct tasks we performed in this analysis:

1. determining which statements in the code had an influence on the value of sum at line 46
2. selecting one (or more) of these statements at which to examine the program state
3. restoring or recreating the program state at those statements to examine specific variables

In this example, we performed the first two tasks ourselves by examining the code, without any assistance from the debugger. For the third task we had to set a breakpoint and *reexecute* the code until the control stopped at the breakpoint. Our debugging job would become much easier if the debugger provided direct assistance in performing all three of these tasks.

We have built a prototype debugger that provides the user with exactly this assistance. The first task—given a variable and a program location, determining which statements in the program affected the value of that variable at that location when the program is executed for a given testcase—is referred to as *Dynamic Program Slicing* [AH90]. Our debugger can find dynamic slices for us automatically. It can also restore the program state at any desired location by *backtracking* the program execution without having to reexecute the program from the beginning [AS88]. In this paper we discuss these two functions—slicing and backtracking—of our prototype debugging tool.

In the next section we discuss the notion of program slicing. In Section 3 we examine the usefulness of an execution backtracking facility in a debugging tool. Then in Section 4 we present an example debugging session with our tool using both slicing and backtracking facilities. In Section 5 we briefly discuss some implementation issues. Finally, in Section 6, we outline related work.

2 Program Slicing

Program Slicing is finding all those statements in a program that directly or indirectly affect the value of a given variable occurrence in the program [Wei84]. The statements that affect the value constitute the *slice* of the program with respect to the given variable occurrence (variable name and statement location). A slice has a simple meaning: it should evaluate the variable occurrence identically to the original program for *all* testcases. As the slices so obtained are independent of any runtime input values, they are referred to as *static* program slices.

For example, Figure 2 shows a static slice with respect to variable area on line 43. Statements that belong to the slice are shown in reverse "video." Note that lines 23–32 that compute class of the triangle are in the slice because the value of class is used in determining which formula is used in computing area in lines 34–42. Alternatively, if we needed the slice for the variable class on line 34, lines 34–42 are not included in the slice, because the value of area computed during one iteration does not affect the value of class during subsequent iterations. This is shown in Figure 3.

As we mentioned above, a static slice includes all statements that *could* influence the value of a variable occurrence for *all* possible inputs to the program. In debugging, however, we are concerned with examining the program behavior for a particular input that revealed the bug. For example, consider again the program in Figure 1 for testcase #1. If we find that the value of area is incorrect at the statement on line 43 for the second triangle (i.e., during the second loop iteration), we would like to know which statements in the program had an effect on the *current* value of area for the *current* testcase. The static slice for area at that location will include all three assignments to area on lines 35, 37, and 40 (along with several other statements), as shown in Figure 2, even though only one of these affects the current value of area.

The problem of finding all statements that influence the value of a variable occurrence for a given testcase is referred to as *Dynamic* Program Slicing. The particular test-case that exercises the bug helps us focus our attention to only that "cross-section" of the program that contains the bug.[4]

[4]When we say the slice contains the bug, we do not necessarily mean that the bug is textually contained in the slice; the bug

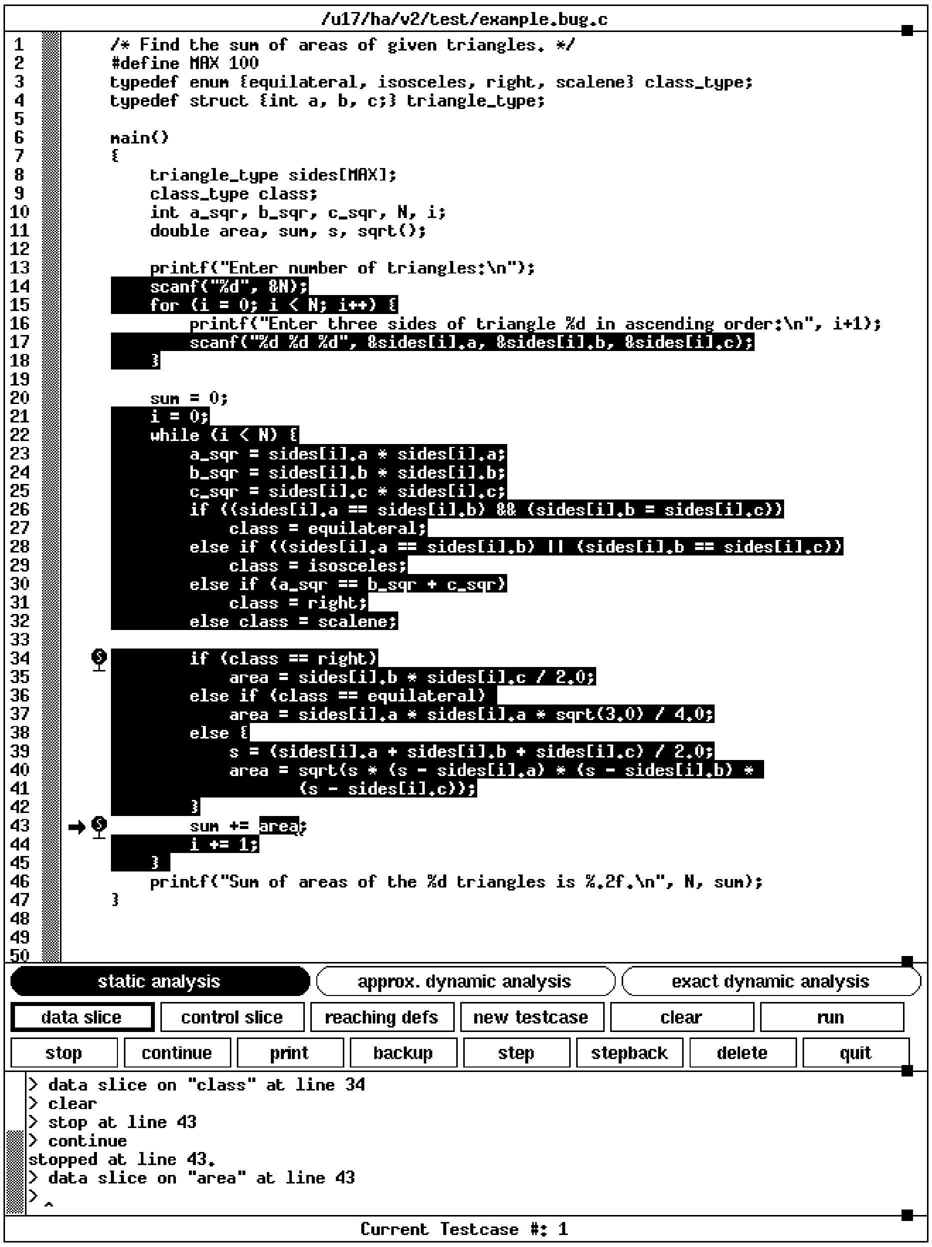

Figure 2: Static slice with respect to variable area on line 43

Figure 4 shows the dynamic slice for the variable area at line 43 during the second loop iteration for testcase #1. Note that only the assignment to area on line 37 is in this dynamic slice. Although the assignment on line 35 is executed during the first iteration, it is not included in the slice because the value of area computed during the first iteration does not affect in any way its value during the second iteration.

could correspond to the absence of something from the slice—a missing **if** statement, a statement outside the slice that should have been inside it, etc. We can discover that something is missing from the slice only after we have found the slice. In this sense, the bug still "lies in the slice."

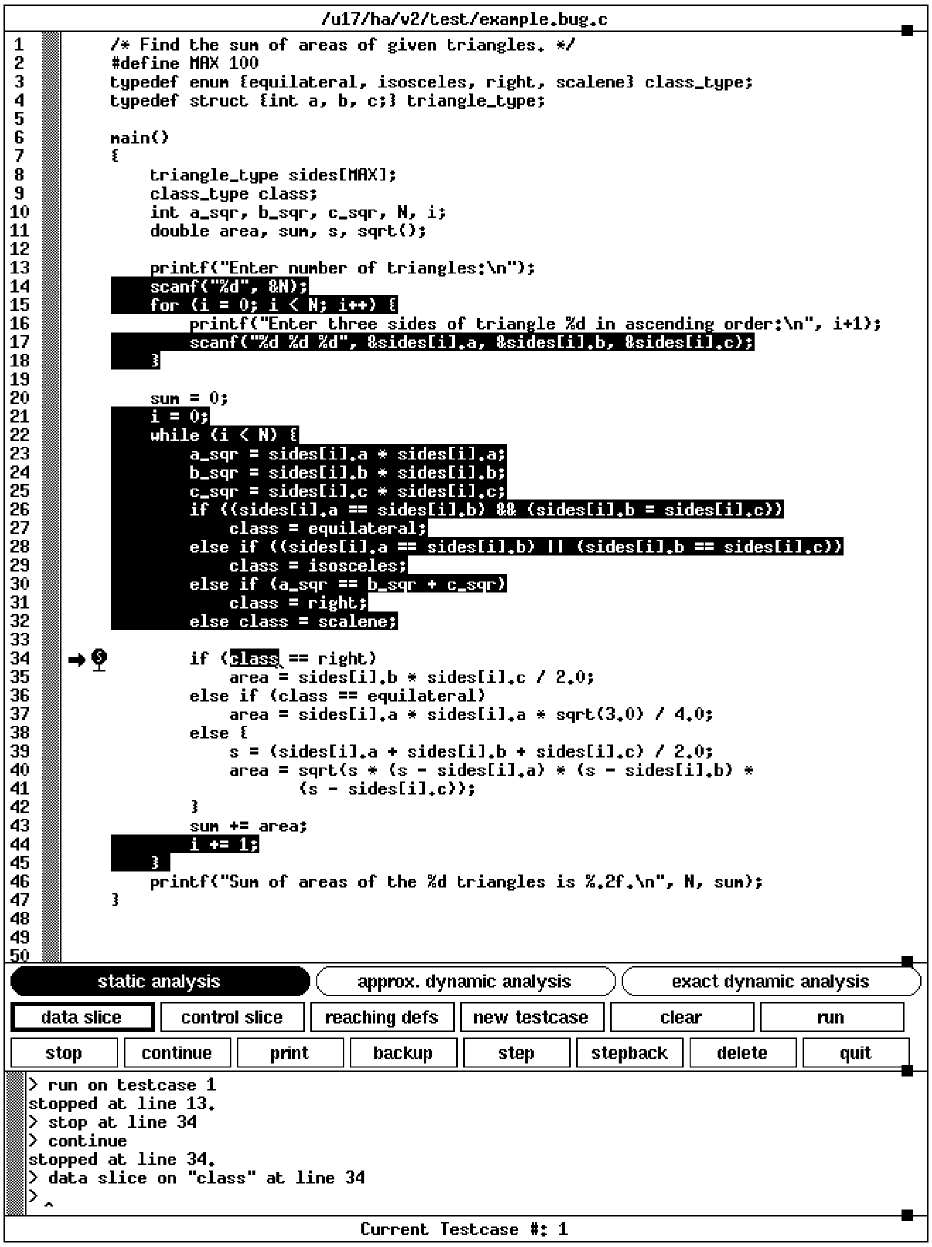

Figure 3: Static slice with respect to variable class on line 34

Similarly statements on lines 23, 24, and 25 that compute a_sqr, b_sqr, and c_sqr do not belong to the dynamic slice although they are executed during the second iteration; the values they compute do not affect the value of area in this case—their values are not even used during the second iteration.

The power of dynamic slicing becomes more apparent when the program being debugged has pointers. Consider, for example, the simple program in Figure 5. It initializes all elements of an array a and then prompts the user for values of i, j, and k. It increments the ith, jth and kth elements of the array and prints out

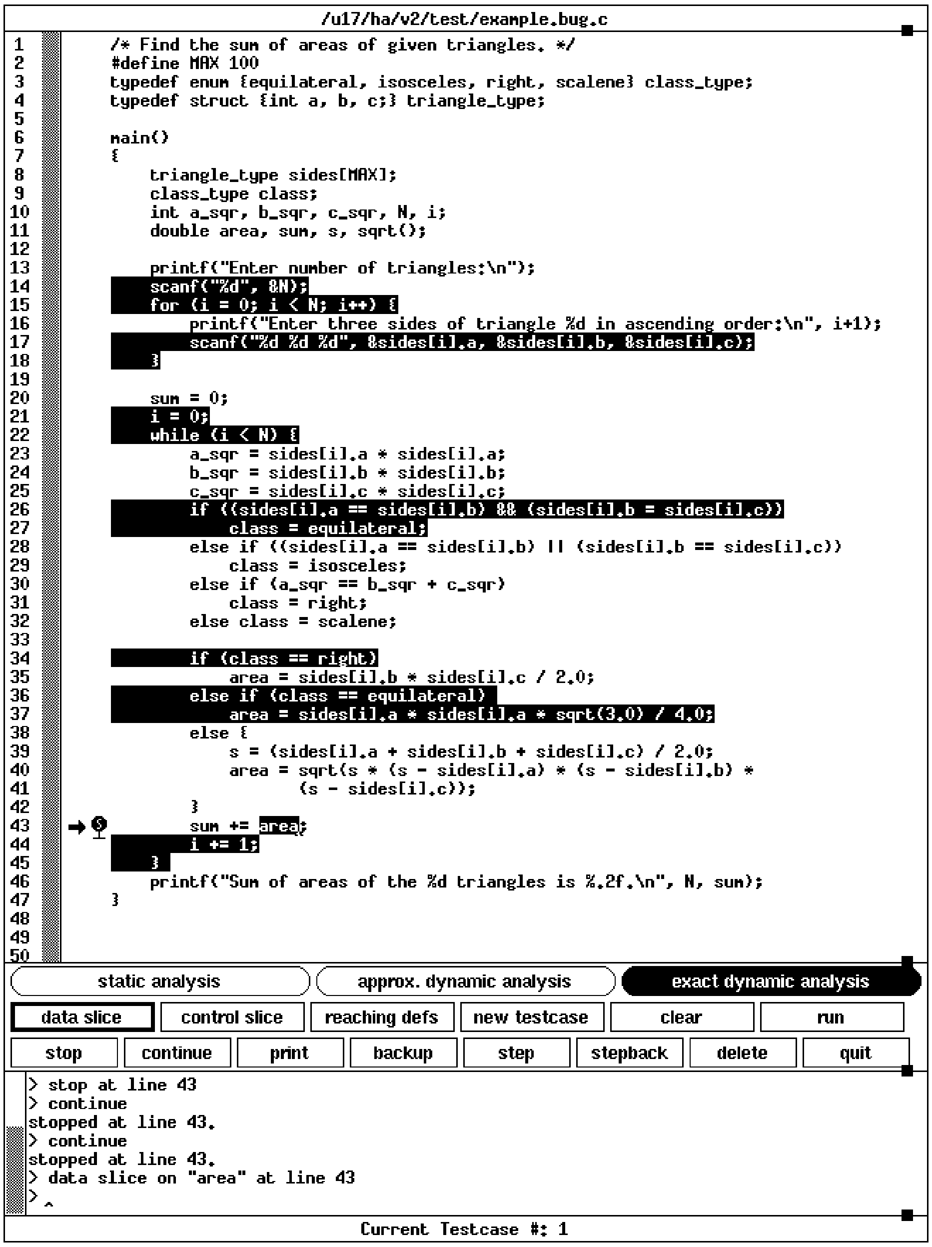

Figure 4: Dynamic slice with respect to variable area on line 43 during the second iteration of the enclosing **while** loop for the testcase #1.

the new values of these elements. p, q, and r are three pointer variables that point to the ith, jth and the kth elements of the array a respectively. Consider the testcase when this program is executed with input values $(i = 3,\ j = 6,\ k = 6)$. Figure 5 also shows the static slice with respect to a[i] on line 29. Figure 6 shows the corresponding dynamic slice. Note that the static slice contains all three indirect assignments through pointers on lines 25–27 because all three pointers, p, q, and r, can possibly be pointing at a[i]. This in turn requires that the three assignments on lines 21–23, the **scanf** statement on line 19, and all assignments on

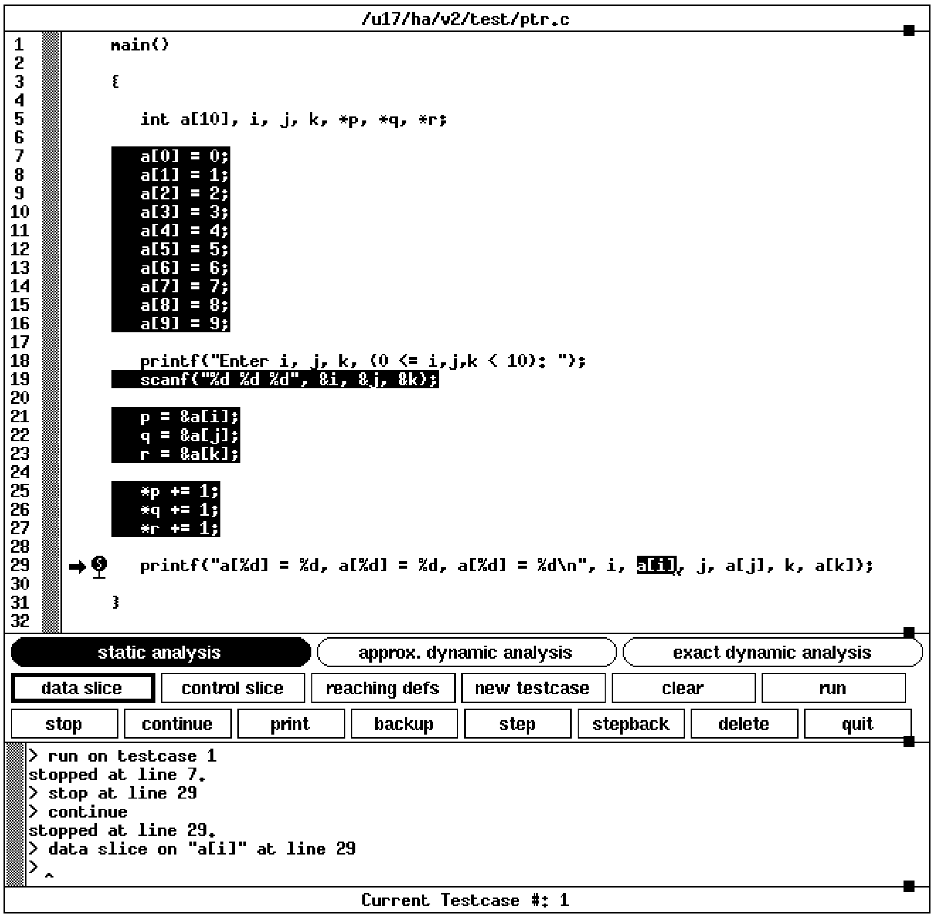

Figure 5: Static slice with respect to a[i] on line 29.

lines 7–16 also be included in the slice. The dynamic slice, on the other hand, contains only one indirect assignment through p on line 25 because, during the current testcase, q and r do not point at a[i]. This means, of the three assignments on lines 21–23, only the assignment to p on line 21 is included in the dynamic slice. Similarly, of all assignments on lines 7–16, only the assignment to the ith array element[5] is in the slice; assignments to all other elements of the array do not belong to the slice. If we obtain the dynamic slice for a[j] on line 29, both indirect assignments on lines 26 and 27 are in the dynamic slice, as shown in Figure 7. This is because, for the current testcase, values of j and k are equal, making both q and r as aliases to the same array element a[6].

As is apparent from Figures 2–7, our debugging tool provides both static and dynamic slicing capabilities. The relevant variable occurrence in the source window is selected by dragging the cursor over it. Then the button labeled *data slice* is clicked to obtain the corresponding slice. The data slice obtained is a *static, approximate dynamic* (explained below), or *exact dynamic* slice based on which of the corresponding toggle buttons is currently selected. In the case of a dynamic slice, the slice is obtained with respect to the execution history thus far. If the program is stopped at a breakpoint, the dynamic slice obtained involves only the

[5] a[3] in this case

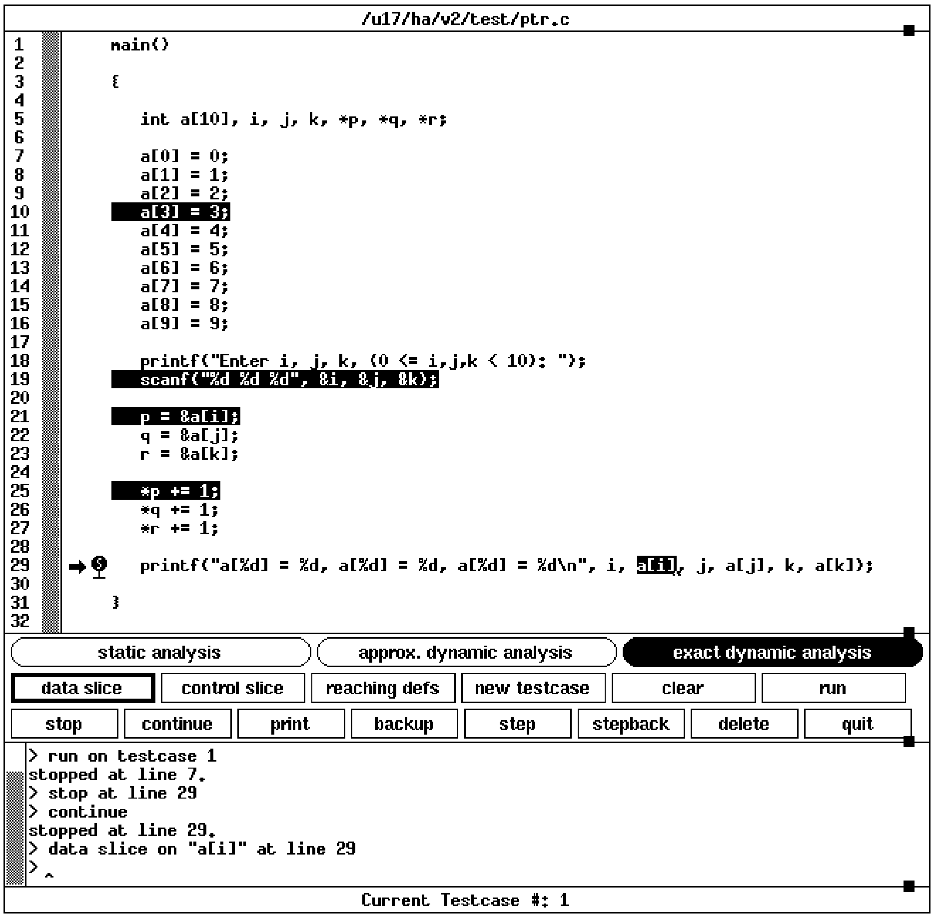

Figure 6: Dynamic slice with respect to a[i] on line 29.

dynamic dependencies that have occurred during the program execution so far.

We used the term *data* slice above because the slice was defined with respect to a data value—a variable occurrence. Sometimes, a program bug is revealed when a wrong section of the program is executed. This may become evident, for example, when a **printf** statement in that section is executed—no erroneous data values are involved. In such cases, we may define a program slice simply with respect to control reaching a certain program location. Such slices are obtained by selecting any characters on the relevant source line, and clicking on the button labeled *control slice*.[6]

Approximate Dynamic Slicing

Frequently, obtaining a static slice may be sufficient to allow the user to localize a program bug. In such situations, the overhead of obtaining dynamic slices is clearly unnecessary. In other situations, especially

[6]Both Data and Control slices are obtained by following data as well as control dependencies in the program. The difference is only in the starting criterion—the former is defined with respect to a variable as well as a statement location, the latter is defined with respect to only a statement location.

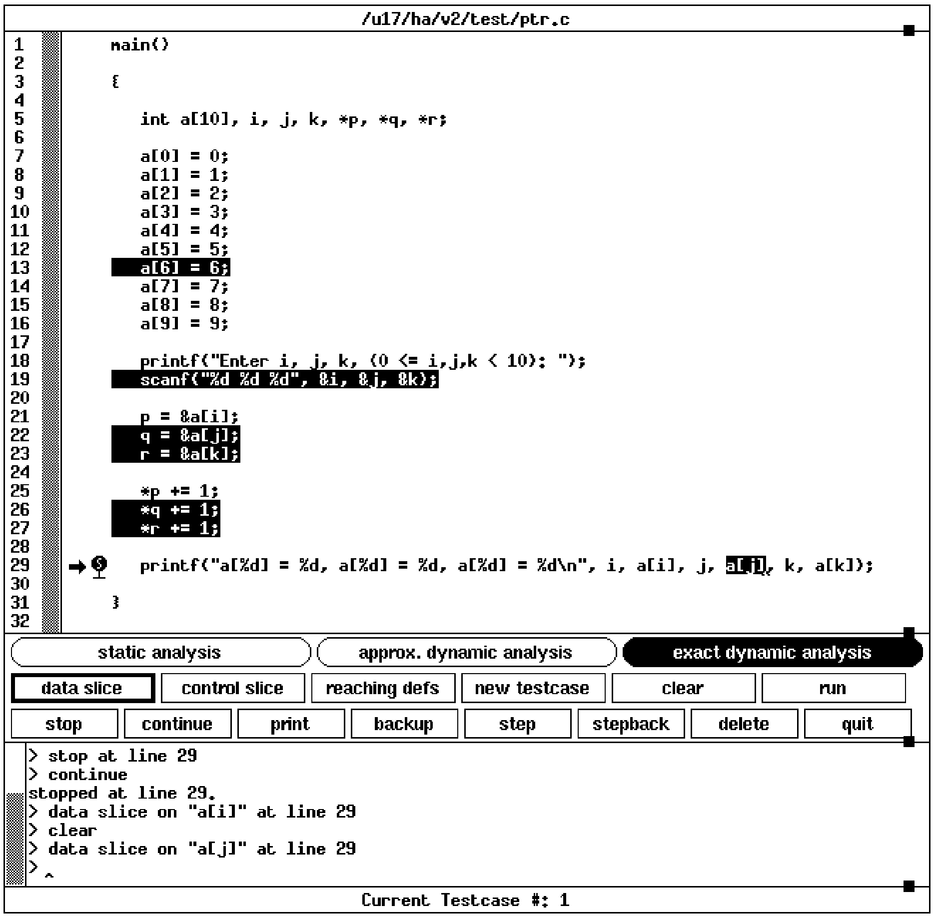

Figure 7: Dynamic slice with respect to a[j] on line 29.

when programs use data structures involving pointers, the sizes of static slices may approach that of the original program. In these situations, dynamic slices become extremely valuable. We also have a facility to provide approximate dynamic slices, which are computationally less expensive to obtain as compared to exact dynamic slices, but may be unnecessarily large compared to them. An *approximate dynamic slice* is obtained by taking the intersection of the appropriate static slice with the program execution path for the current testcase (all statements that are executed during the current test case).[7] Consider again, for example, the program in Figure 1 and the testcase #1. Figure 8 shows the approximate dynamic slice with respect to area on line 43 when the execution reaches there during the first loop iteration. The approximate slice obtained in this case is exactly the same as the corresponding exact dynamic slice. Figure 9 shows the approximate dynamic slice for the same variable occurrence but during the second loop iteration. Compare this with the corresponding exact dynamic slice shown in Figure 4 and the corresponding static slice shown in Figure 2.

It is up to the user to judiciously select the slicing criterion—static, dynamic, or approximate dynamic—

[7] Actually, the intersection is performed by doing the static slicing analysis only over the nodes in the program dependence graph that are visited during the program execution. See [AH90] for details.

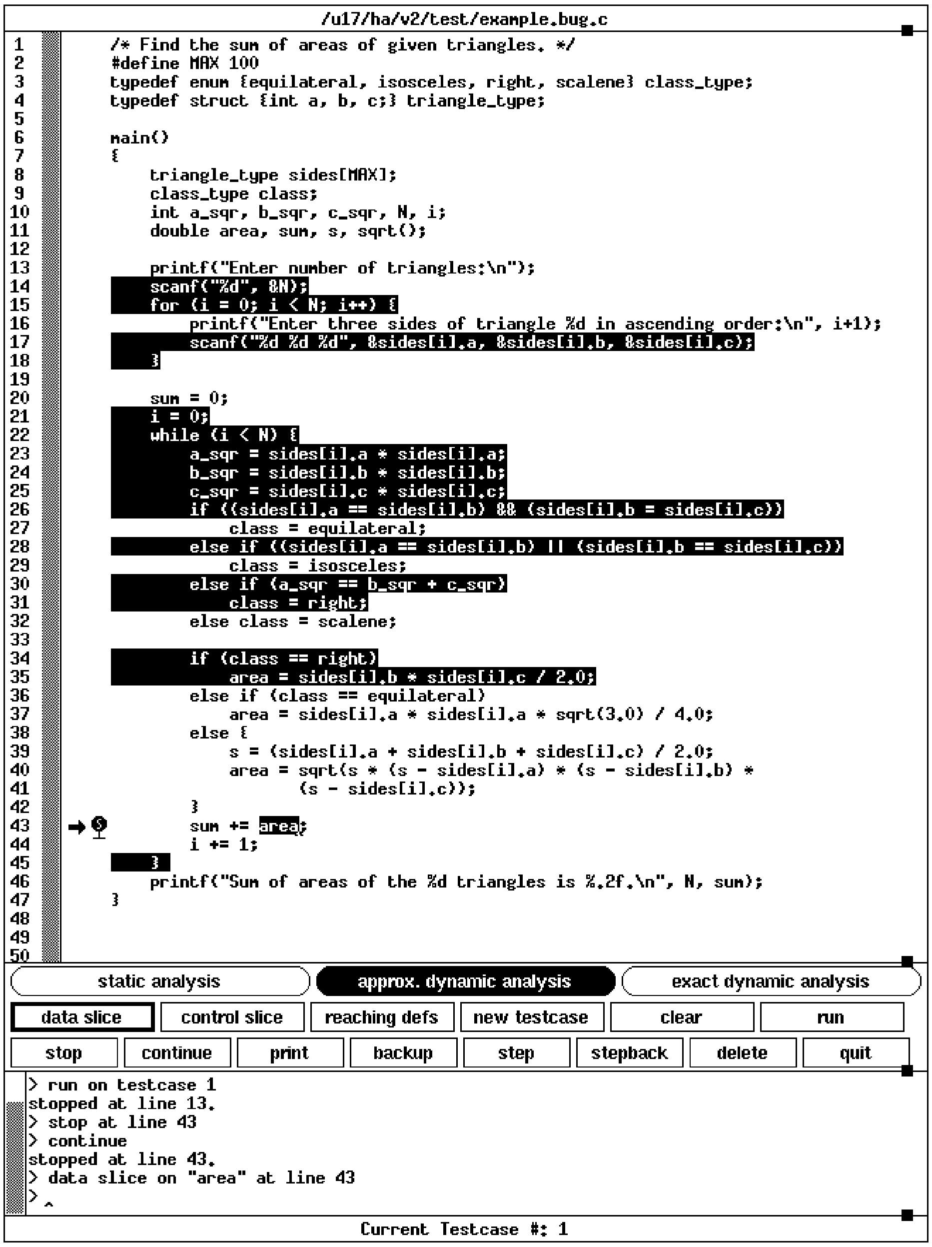

Figure 8: Approximate dynamic slice on area on line 43, first iteration.

that best suits his needs. The three alternatives provide a span of space-time-accuracy trade-offs. The selection may also be changed during the debugging session. Thus, it is possible to start with less expensive static analysis and identify a smaller program region that most likely contains the bug, and then use the more expensive dynamic analysis only within that small region. The reader is referred to [AH90] for a detailed discussion on these approaches to computing program slices and their relative costs.

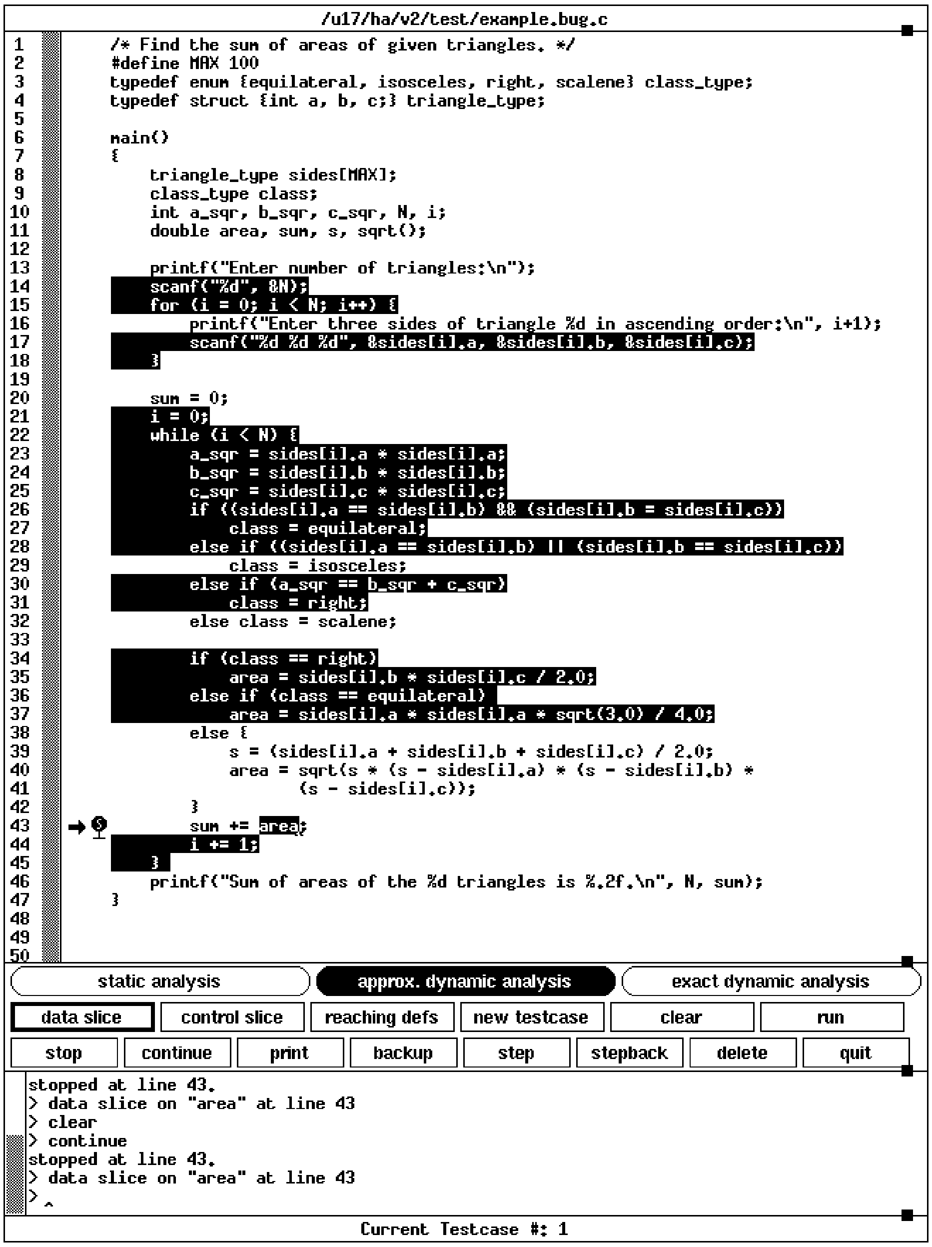

Figure 9: Approximate dynamic slice on area on line 43, second iteration.

Reaching Definitions

In some situations, looking at the whole slice at once may be overwhelming. In these situations, it may be useful to analyze the inter-statement program dependencies one at a time. For this reason, our tool also provides a function to show only the direct reaching definitions of any variable occurrence. If static analysis is currently selected, all possible reaching definitions are highlighted. If exact dynamic analysis is selected, the unique reaching definition of the variable occurrence for the current testcase is highlighted.

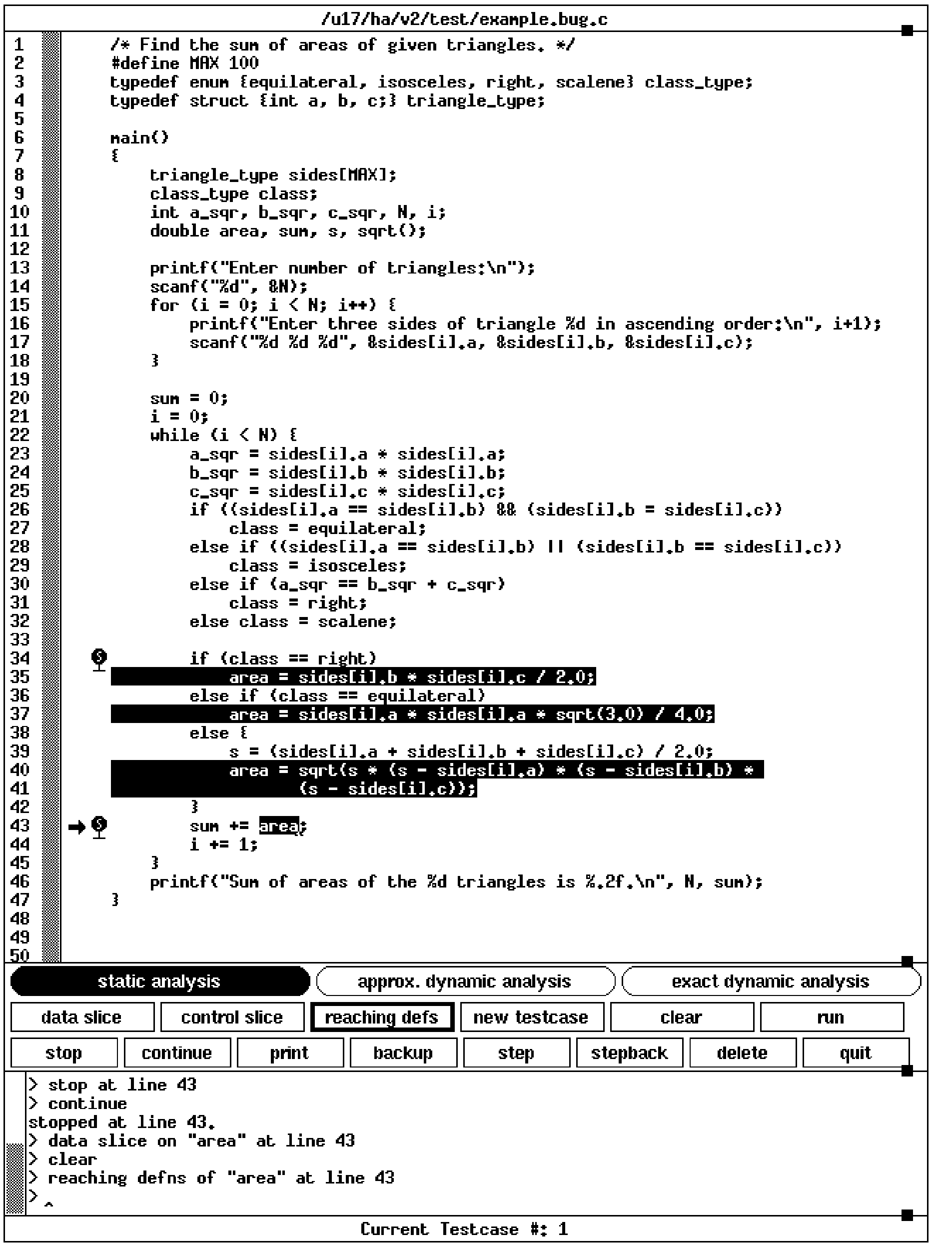

Figure 10: Static reaching definitions of variable area on line 43

For example, Figure 10 shows the static reaching definitions of variable area at line 43. Figure 11 shows the unique dynamic reaching definition for the same variable occurrence during the second iteration of the enclosing **while** loop for testcase #1.

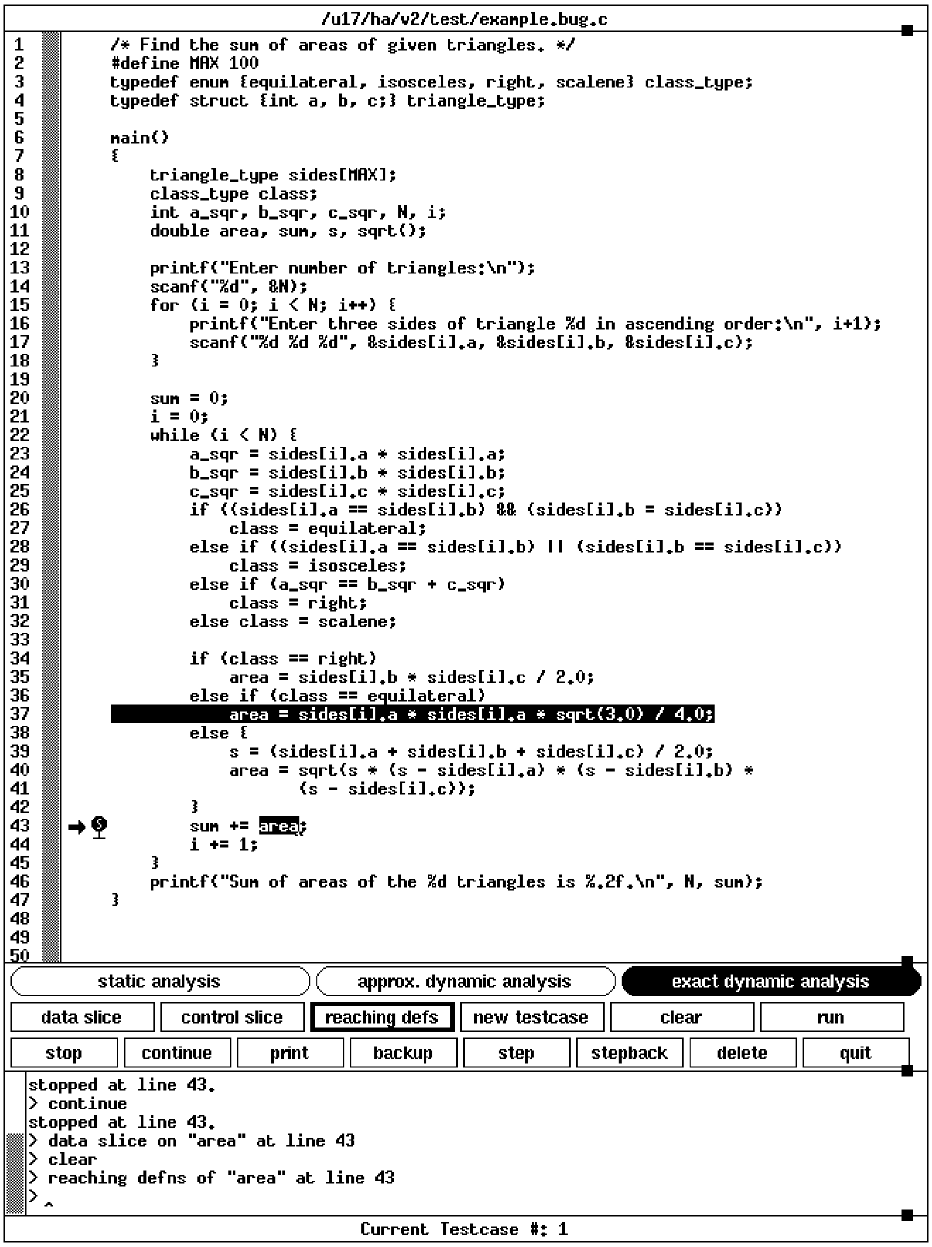

Figure 11: Dynamic reaching definitions of variable area on line 43

Handling Multiple Testcases

After a program is written, it is normally run against several input data sets designed to test specific aspects of the program behavior. If a program works correctly on one testcase but fails on another, it may be helpful to analyze the program behavior under both these testcases. The inclusion or exclusion of a statement in dynamic slices with respect to the two testcases may provide valuable debugging clues. The same idea may be generalized to that of examining dynamic program behavior under several testcases.

Our tool allows the user to save several testcases and then select any one for dynamic analysis. The current testcase selection may be changed at any time. Performing any dynamic analysis (either exact or approximate) requires that a valid testcase be currently selected. The testcase selection is performed by clicking on the button labeled *new testcase*. When this button is clicked, a dialogue window pops up prompting the user to specify which testcase to use. The specified testcase becomes the current testcase and remains so until a new testcase is selected.

3 Execution Backtracking

When debugging using conventional debuggers, one often needs to *reexecute* the program being debugged from the start. For example, when the program execution is suspended at a breakpoint, after examining some variable values we may discover that the error occurred at an earlier location. We may then decide to set another breakpoint at an earlier program statement and restart the program. When the execution stops at this new breakpoint, we may reexamine the program state. We may have to repeat this process of setting breakpoints in backward order and reexecuting the program several times. For large programs such repeated execution from the beginning may be very cumbersome.

Our debugging tool provides an execution backtracking facility with which program state can be restored at any desired earlier location without having to reexecute the entire program. Just as the normal forward program execution is suspended whenever a breakpoint is encountered, our tool can "execute" the program in the *reverse* direction and continue executing backwards until a breakpoint is reached. This way, when stopped at a breakpoint, if we find that the error occurred at an earlier location and we wish to examine the program state at that location, we simply need to set another breakpoint there and execute backwards. When the backward execution stops at that breakpoint, the tool will have restored the program state to whatever it was when the execution last reached that point. Backward execution is started simply by clicking on the button labeled *backup*. The button labeled *continue* restores the program execution in the normal forward direction.

For example, consider again the program in Figure 1 and testcase #1. If the program execution is stopped at line 46, and we discover that the value of sum is incorrect there, we may set a breakpoint on line 43 and start backward execution. The loop was iterated two times for this testcase, so the second iteration will be reached first during backward execution. When this execution stops at line 43, the program state will be exactly the same as if the execution had stopped there during normal execution during the second iteration of the loop. If we examine the value of sum there, we will get its value just before the last assignment was executed (see Figure 13). If we find this previous value of sum to be correct, we may conclude that it is the current value of area that is incorrect. If we wanted to backup to the same location during the previous iteration, we simply need to continue our backward execution from there on. As no other breakpoint is encountered during the same iteration, the backward execution is again suspended when it reaches the breakpoint at line 44 during the previous iteration.

Figure 12 shows another example of backtracking. The bottom output window shows the tool output for a sequence of debugging commands. After we select a testcase, the program execution is automatically stopped before the first executable statement (on line 7, in this case). If we examine the value of array a at this time (by selecting the variable a anywhere in the source window and clicking on the *print* button), we find that all elements 0–9 of a have the value 0. We now set a breakpoint on line 16 (by selecting line 16 and clicking on the *stop* button) and continue the program execution (by clicking on the *continue* button).

The execution stops when the breakpoint on line 16 is reached. We again examine the value of array a and find, as expected, that elements 0–8 of the array have values 0–8 respectively but element 9 still has zero because the execution stopped just *before* the assignment on line 16 is executed. We may now set a breakpoint on line 12 and start reverse execution by clicking on the button labeled *backup*. The reverse execution stops upon encountering the breakpoint after backing over the statement on line 12. If we now examine the value of array a we notice that elements 5–8 have their values restored to zero. If we continue the reverse execution (by clicking again on the *backup* button) the execution stops upon reaching the start of the program on line 7. Now the value of each element of the array is restored back to zero, their initial value. If we continue the program execution in the forward direction from here, it will again stop at the breakpoint on line 12. The values of elements 0–4 will now again be 0–4 respectively, while those of 5–9 will still be zero.

Back-Stepping

Most conventional debuggers provide facilities to step through the program execution one line at a time. While debugging we often "think backwards" from the location where a bug is manifested, so it would be helpful if the debugger also lets the user *step back* through the program execution, statement by statement. Our debugging tool provides such a back-stepping facility. When execution is stopped at a breakpoint, clicking on the button labeled *stepback* will undo the effect of the last statement executed before the program stopped at the current breakpoint. Clicking one more time on the same button undoes the effect of the last statement before that, and so on. Note that the statement next to be executed (the *current* statement) is indicated by the arrow in the left margin of the code window (see, e.g., Figure 13).

4 An Example Debugging Session

Let us illustrate our system by presenting a small debugging session using the program in Figure 1 and testcase #1 we used in our discussion above ($N = 2$, and the sides of the two triangles being (5, 4, 3) and (4, 4, 2), respectively). When the program is executed for this testcase, the final value of sum printed is 12.93 instead of the correct value 9.87 (the area of the first triangle being 6 and that of the second being 3.87).[8]

We first enable exact dynamic analysis by clicking on the toggle button labeled *exact dynamic analysis*. We then set a breakpoint on line 46 and run the program for the above testcase. When the execution stops at the breakpoint,[9] we select the variable sum using the mouse and print its value by clicking on the *print* button. The incorrect value, 12.93, is printed.

We then click on the *reaching defs* button to find the last definition of sum and find that the assignment on line 43 last assigned a value to sum, as shown in Figure 11. We set another breakpoint on line 43 and click on the *backup* button to restore program state to what it was just before the assignment on line 43 was last executed. We then print the value of sum at that location, and find that its correct value at that point, 6, is printed.

[8]We have rounded values of area and sum to the second decimal place for this discussion.

[9]Represented by the little stop sign. See Figure 13.

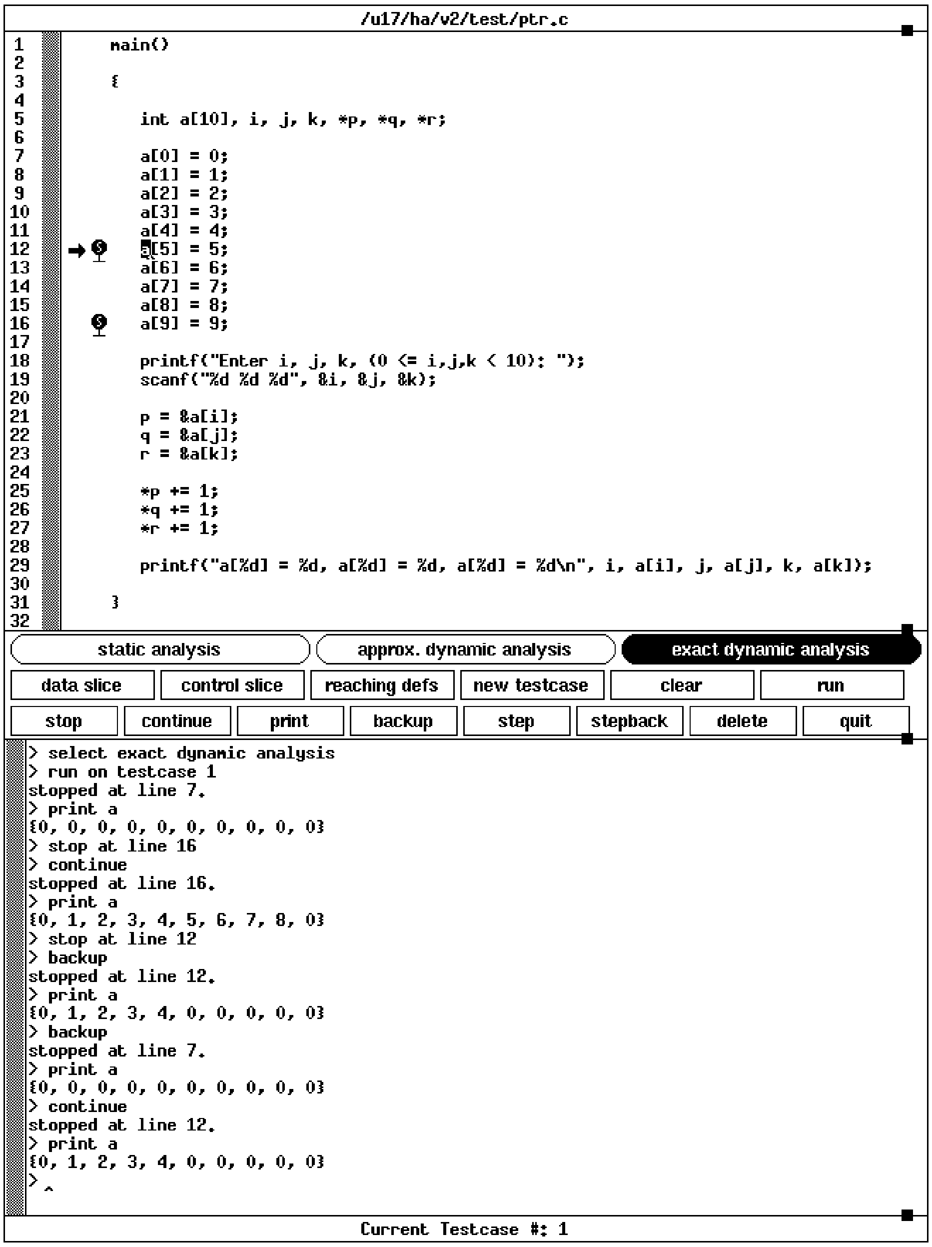

Figure 12: Execution backtracking from line 16 to line 12 to line 7

Thus far, sum contains only the area computed during the previous iteration. This implies that area was correctly computed for the first triangle and that it must be wrong for the second triangle. We print the current value of area and find that it is indeed incorrect—6.93 instead of 3.87. The tool screen at this time is shown in Figure 13. Notice the value of sum before and after the "backup" command, as displayed in the bottom (output) window.

We next click on the *data slice* button to obtain the dynamic slice for the current value of area. The

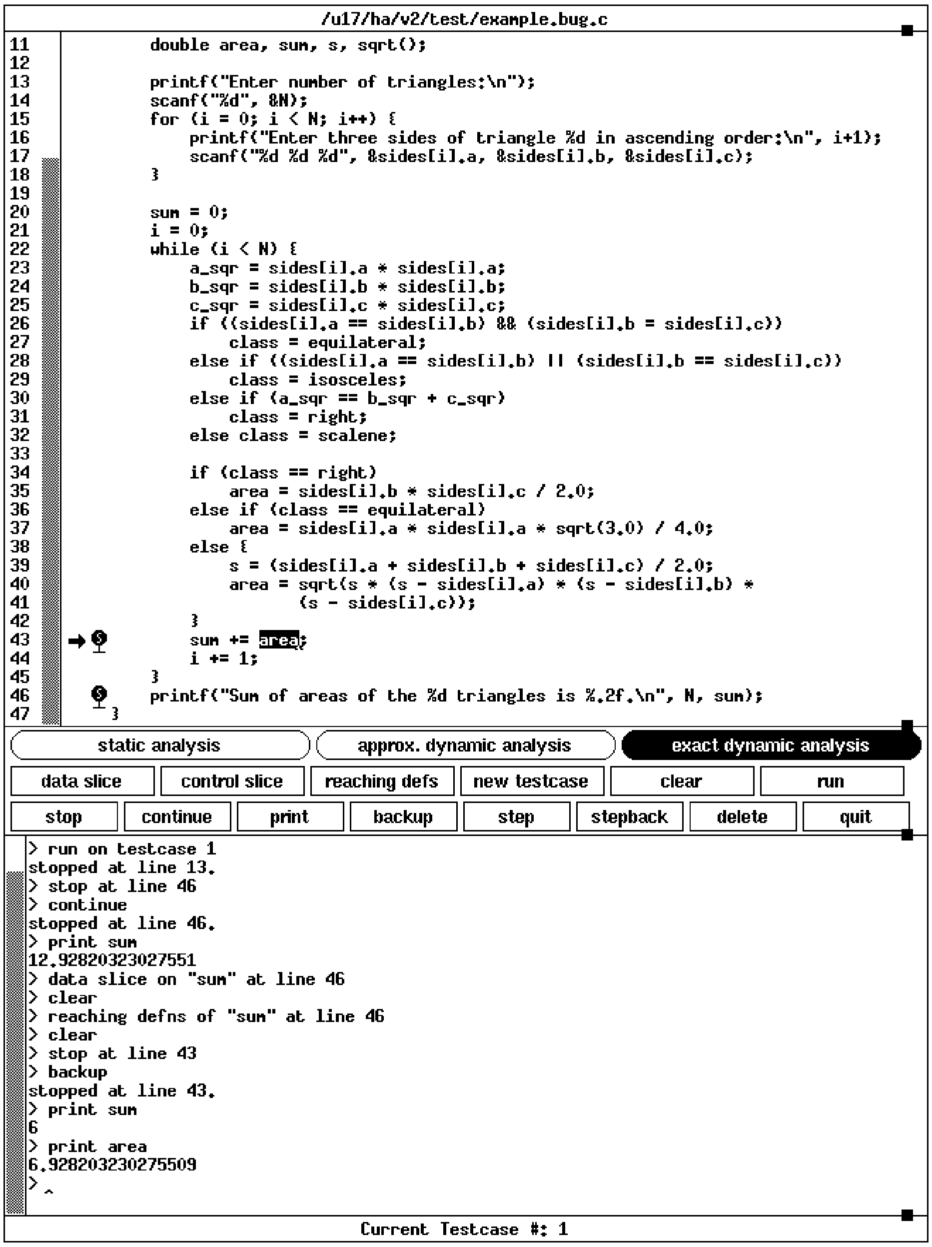

Figure 13: Tool screen after backtracking from line 46 to line 43

slice obtained is as shown in Figure 4. To our surprise, we find the assignment on line 37 that computes the area of an equilateral triangle belongs to the slice instead of the assignment on line 40 for an isosceles triangle. We also find the assignment on line 27 that assigns equilateral to class in the slice.

To check if the values of the sides of the triangle are correct, we print sides[i] and find that the current values of the three sides of the triangle are (4, 2, 2) and not (4, 4, 2)—the input values. The **if** statement on line 26 determines if the triangle is equilateral, and if so, sets the value of class. Therefore, we set another breakpoint at that line and further backup execution to that location. We again print the value of sides[i] and this time we find the values of the three sides to be correct—(4, 4, 2). The current values of the three sides are not equal and still the boolean expression on line 26 that checks if all three sides are equal evaluates to **true**! This suggests there must be something wrong with the expression. On further examination of that expression, we discover the error in the second equality subexpression within the condition. The erroneous comparison expression always returns **true** (as the length of a side is always greater than zero) and also assigns the length of side c to side b. The tool screen at this time is shown in Figure 14. Note the values of sides[i] before and after the backup is done.

5 Implementation

Our debugging system is built into versions of the GNU C compiler "gcc" and the GNU source-level debugger "gdb" [Sta89]. Our intent has not been to write a production-quality tool but to show the feasibility of the above mechanisms. We decided, therefore, to modify an existing compiler and debugger rather than write a new system. We chose the GNU tools because of their availability and their ability to run on different hardware platforms. Although this choice has led to some problems, it has allowed us to rapidly develop a prototype that will work for full ANSI C.

We have modified gcc to produce a program dependence graph [FOW87, OO84, HRB90] along with the object code of the given program. We also made several modifications to gdb:

- Modifications were made to read and use the program dependence graph generated during compilation.
- Code was added to perform necessary runtime analysis required for execution backtracking and dynamic slicing.
- Modules for obtaining static, dynamic, and approximate dynamic slices were added.
- A window and mouse-based interface was added so slices could be displayed.

As our intention was to show how slicing and backtracking can be usefully combined with standard debugging functions like breakpoints, single-stepping, examining values, etc., we only included these most common debugging functions in our windowed interface rather than provide every function that gdb provides.

The backtracking mechanism described here is implemented as a form of history-saving, with values attached to the same structures used in the slicing mechanism. This gives the user considerable power at the possible expense of the need for unbounded storage. Bugs that produce infinite loops, for instance, may exhaust the storage available to the debugger.

Alternatively, it is possible to use *structured backtracking* as described in [AS88]. This approach provides the user with almost the same capabilities, but allows the storage needed to be bounded at compile time in most cases. If we were using structured backtracking in our example in Section 4, we would not have been able to backtrack from the second iteration of the loop to the inside of the first. Rather, we would have had to backtrack to the beginning of the loop itself and execute forwards to the given statement. This can be implemented in a manner transparent to the user; a more complete explanation is available in [AS88, ADS90].

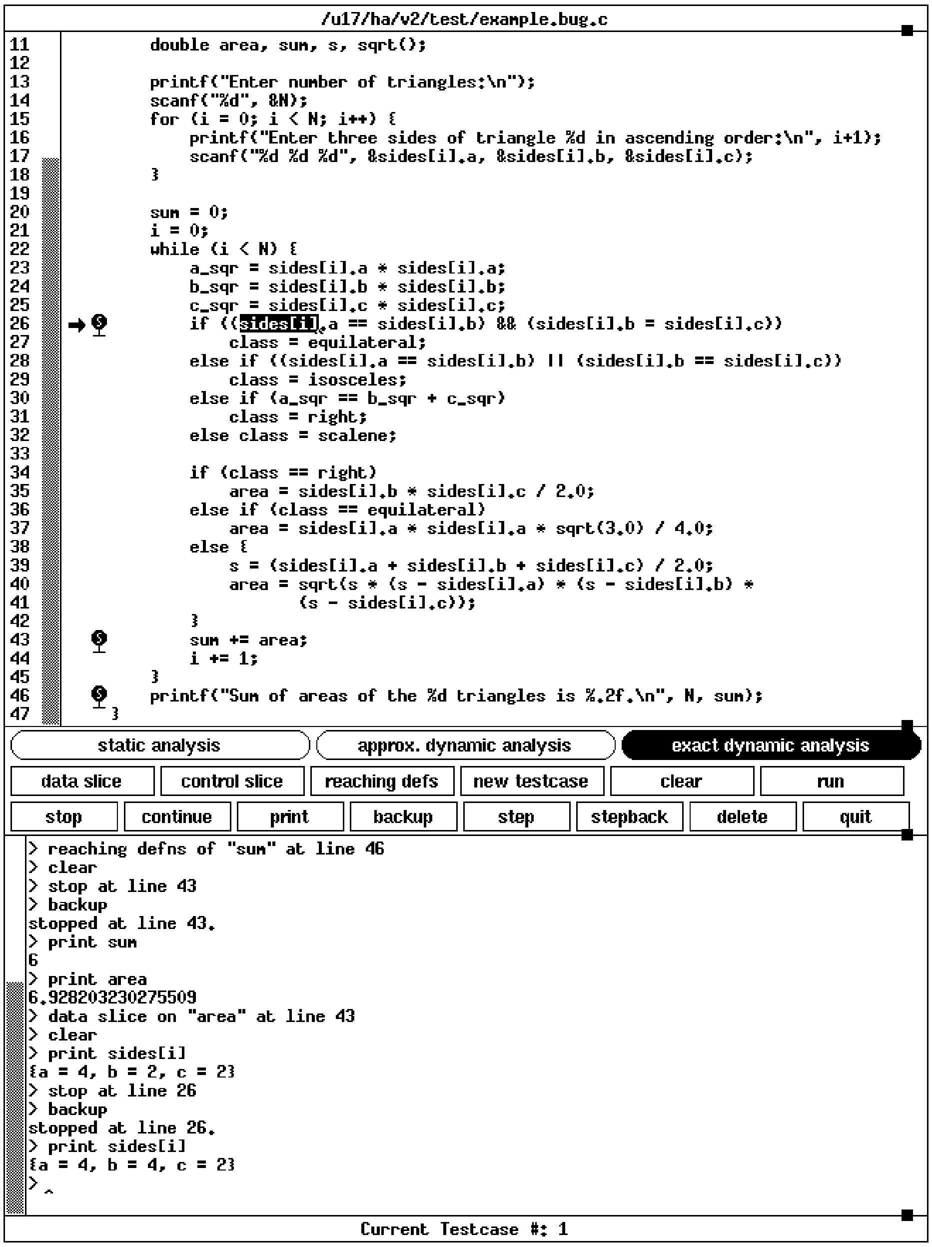

Figure 14: Tool screen after execution has been backtracked to line 26

6 Related Work

The concept of static program slicing was first proposed by Weiser [Wei84, Wei82]. He also presented an algorithm to compute static slices based on iterative solution of data-flow equations. Ottenstein and Ottenstein later presented an algorithm in terms of graph reachability in the Program Dependence Graph, but they only considered the intra-procedural case [OO84]. Horwitz, Reps, and Binkley have extended the program dependence graph representation to what they call the "system dependence graph" to find inter-procedural static slices under the same graph-reachability framework [HRB90]. Bergeretti and Carré have also defined information-flow relations somewhat similar to data- and control dependence relations, that can be used to obtain static program slices (referred to as "partial statements" by them) [BC85]. Uses of program slicing have also been suggested in many other applications, e.g., program verification, testing, maintenance, automatic parallelization of program execution, automatic integration of program versions, etc. (see, e.g., [Wei84, BC85, HPR89]).

When a program slice is defined with respect to a variable occurrence, it is assumed that control does eventually reach the corresponding program location. The issue of non-termination of program execution is not addressed under this definition. Podgurski and Clark have extended the regular notion of control dependence (which they refer to as "strong control dependence") to "weak control dependence" that includes inter-statement dependencies involving program non-termination [PC90]. To detect program faults other than infinite loops, however, strong control dependence gives much finer slices compared to weak control dependence. The definintion of data-dependence remains the same in both cases.

Korel and Laski extended Weiser's static slicing algorithms for the dynamic case [KL88]. Their definition of a dynamic slice is different from ours[AH90]. Their definition requires that if any one occurrence of a statement in the execution history is included in the slice then all other occurrences of that statement be automatically included in the slice, even when the value of the variable in question at the given location is unaffected by those other occurrences. For example, if the program in Figure 4 is executed for testcase #1, and we find the dynamic slice for the variable area on line 43 during the second iteration, their definition will also require that all statements that affected the variable area at the same location during the previous iterations be included in the slice. This means lines 23, 24, 25, 30, 31, and 35 also must be included in the dynamic slice, although the current value of area is totally unaffected by the execution of these statements. Thus, their definition may yield unnecessarily large dynamic slices. The corresponding slice produced by our tool is shown in Figure 4.

The concept of reverting program state in a debugging system is not new. EXDAMS, an interactive debugging tool for Fortran developed in the late 1960s, also provided an execution *replay* facility [Bal69]. In that system, first the complete history tape of the program being debugged for a testcase was saved. Then the program was "executed" through a "playback" of this tape. At any point, the program execution could be backtracked to an earlier location using the information saved on the history tape. However, if a program was stopped at some location it was not possible to change values of variables before executing forward again because EXDAMS simply replayed the program behavior recorded earlier.

Zelkowitz incorporated a backtracking facility within the programming language PL/1 by adding a RETRACE statement to the language [Zel71]. With this statement, execution could be backtracked over a desired number of statements, up to a statement with a given label, or until the program state matched a certain condition. This incorporation of backtracking facilities within a programming language can be useful in programming applications where several alternate paths should be tried to reach a goal. Such problems frequently arise in artificial intelligence applications, for instance. However, because the user must program the RETRACE statements into the code, this approach does not provide an *interactive* control over backtracking while debugging.

Systems like IGOR[FB88] and COPE[ACS84] perform periodic checkpointing of memory pages or file blocks modified during program execution. Using these checkpoints program execution can be restarted at arbitrary points. This approach, while suitable for undoing effects of whole programs, may not be appropriate for performing statement-level backtracking: even if a statement execution results in a small change in a file or a page block, the whole block is saved again.

The INTERLISP system, a program development environment for the LISP language, also provides recovery facilities within the language framework [Tei78]. It provides UNDO and REDO functions whose implementations are embedded within the language processor. INTERLISP provides these backtracking facilities in a functional programming environment, whereas our interest has been to provide a backtracking facility for debugging in the more common procedural environment.

Miller and Choi's PPD [MC88] performs flow-back analysis of parallel programs like EXDAMS does for sequential programs. They use a notion of incremental tracing where portions of the program state are checkpointed at the start and the end of segments of program-code called emulation-blocks. Later these emulation blocks may be reexecuted to build the corresponding segments of the dynamic dependence graph.

7 Concluding Remarks

Debugging is a complex and difficult activity. The person doing the analysis must determine the cause and the location of a program failure. The failure may be manifested far away from the fault (bug) itself—far away, both in lines of code and in execution history. Providing tools that increase the ability of the analyst to identify the location or nature of the software bug involved will lead to more efficient debugging.

In this paper, we have presented a prototype tool that integrates some conventional debugging techniques with some interesting new mechanisms. Although some of the ideas behind our new approaches are related to techniques used in previous systems, our implementation and use of these features involves novel new algorithms and presentation methods.

The result of our efforts is a window-based debugging tool that presents the user with a number of powerful, yet easy-to-understand, techniques to help identify the location of software faults. Our experiences with the tool so far have convinced us that these techniques are quite useful, and we will be conducting further experimentation on these, and other new debugging mechanisms in the future.

Acknowledgments

We would like to thank Bob Horgan for our discussions with him, Ed Krauser for his help in implementation, and Ryan Stansifer and Hsin Pan for their comments on an earlier draft of this paper.

References

[ACS84] James E. Archer, Jr., Richard Conway, and Fred B. Schneider. User recovery and reversal in interactive systems. *ACM Transactions on Programming Languages and Systems*, 6(1):1–19, January 1984.

[ADS90] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. An execution backtracking approach to program debugging. Technical Report SERC-TR-22-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, 1990. Revised from the August, 1988 report.

[AH90] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*, White Plains, New York, June 1990. ACM SIGPLAN. SIGPLAN Notices, 25(6):246–256, June 1990.

[AM86] Evan Adams and Steven S. Muchnick. Dbxtool: a window-based symbolic debugger for Sun work-stations. *Software Practice and Experience*, 16(7):653–669, July 1986.

[AS88] Hiralal Agrawal and Eugene H. Spafford. An execution backtracking approach to program debugging. In *Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference*, pages 283–299, Portland, Oregon, September 1988.

[AS89] Hiralal Agrawal and Eugene H. Spafford. A bibliography on debugging and backtracking. *ACM Software Engineering Notes*, 14(2):49–56, April 1989.

[Bal69] R. M. Balzer. Exdams: Extendible debugging and monitoring system. In *AFIPS Proceedings, Spring Joint Computer Conference*, volume 34, pages 567–580, Montvale, New Jersey, 1969. AFIPS Press.

[BC85] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, January 1985.

[Car86] Thomas A. Cargill. Pi: a case study in object-oriented programming. In *OOPSLA'86 Conference Proceedings*, Portland, OR, September 1986. ACM SIGPLAN. SIGPLAN Notices, 21(11):350–360, November 1986.

[Dun86] Kevin J. Dunlap. Debugging with Dbx. In *Unix Programmers Manual, Supplementary Documents 1*. 4.3 Berkeley Software Distribution, Computer Science Division, University of California, Berkeley, CA, April 1986.

[FB88] Stuart I. Feldman and Channing B. Brown. Igor: a system for program debugging via reversible execution. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, Madison, WI, May 1988. ACM SIGPLAN/SIGOPS. SIGPLAN Notices, 24(1):112–123, January 1989.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its uses in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[Gou74] J. D. Gould. An exploratory study of computer program debugging. *Human Factors*, 16:258–277, 1974.

[HPR89] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkeley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[Kat79] H. Katsoff. Sdb: a symbolic debugger. *Unix Programmer's Manual*, 1979.

[KL88] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29:155–163, October 1988.

[Luk80] F. J. Lukey. Understanding and debugging programs. *International Journal of Man-Machine Studies*, 12(2):189–202, February 1980.

[MB79] J. Maranzano and S. Bourne. A tutorial introduction to ADB. *Unix Programmers Manual*, 1979.

[MC88] Barton P. Miller and Jong-Deok Choi. A mechanism for efficient debugging of parallel programs. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, Atlanta, GA, June 1988. ACM SIGPLAN. SIGPLAN Notices, 23(7):135–144, July 1988.

[MH89] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–623, December 1989.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, Pittaburgh, PA, April 1984. ACM SIGSOFT/SIGPLAN. SIGPLAN Notices, 19(5):177–184, May 1984.

[PC90] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.

[Sch71] Jacob T. Schwartz. An overview of bugs. In Randall Rustin, editor, *Debugging Techniques in Large Systems*, pages 1–16. Prentice-Hall, Engelwood Cliffs, NJ, 1971.

[Sta89] Richard M. Stallman. *GDB Manual, third edition, GDB version 3.4*. Free Software Foundation, Cambridge, MA, October 1989.

[Tei78] Warren Teitelman. *Interlisp Reference Manual, Fourth Edition*. Xerox Palo Alto Research Center, Palo Alto, CA, 1978.

[Wei82] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[Zel71] M. V. Zelkowitz. *Reversible Execution As a Diagnostic Tool*. PhD thesis, Dept. of Computer Science, Cornell University, January 1971.