TOWARDS AUTOMATIC DEBUGGING OF COMPUTER PROGRAMS

A Thesis
Submitted to the Faculty

of

Purdue University

by

Hiralal Agrawal

In Partial Fulfillment of the
Requirements for the Degree

of

Doctor of Philosophy

August 1991

ACKNOWLEDGMENTS

First of all I would like to thank my advisor, Rich DeMillo, who gave me the initial impetus to look into the problem of software debugging and helped me develop many ideas presented in this dissertation. I am also grateful to my co-advisor, Gene Spafford, whose expert advise, particularly on implementation issues, was always indispensable. I am also thankful to my office mate, Ed Krauser, who collaborated with me in the early stages of the development of our prototype debugging tool. I would also like to express my thanks to Ahmed Elmagarmid, Bob Horgan, Aditya Mathur, Piyush Mehrotra, Hsin Pan, Ryan Stansifer, Guda Venkatesh, Nok Viravan, Michal Young, and many others with whom I had fruitful discussions from time to time.

If there is one person I am most indebted to for everything I have been able to achieve, he is my father Ghanshyam Das Agrawal, who fostered my love for learning and who, despite adversities, supported me in all my academic endeavors. I would also like to thank my wife, Sweta, who has always had words of encouragement for me, and who has patiently tolerated my absence during the many long hours I spent preparing this dissertation.

Support for this research was provided, in part, by a grant from the Software Engineering Research Center at Purdue University, a National Science Foundation Industry/University Cooperative Research Center (NSF Grant ECD–8913133), by National Science Foundation Grant CCR–8910306, and by a summer internship at Bellcore in 1989.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

Page

LIST OF FIGURES

Figure Page

Figure Page

ABSTRACT

Agrawal, Hiralal. Ph.D., Purdue University, August 1991. Towards Automatic Debugging of Computer Programs. Major Professors: Richard A. DeMillo and Eugene H. Spafford.

Programmers spend considerable time debugging code. Symbolic debuggers provide some help but the task still remains complex and difficult. Other than breakpoints and tracing, these tools provide little high level help. Programmers must perform many tasks manually that the tools could perform automatically, such as finding which statements in the program affect the value of an output variable under a given testcase, what was the value of a given variable when the control last reached a given program location, and what does the program do differently under one testcase that it does not do under another. If the debugging tools provided explicit support for such tasks, the whole debugging process would be automated to a large extent.

In this dissertation, we propose a new debugging paradigm that easily lends itself to automation. Two tasks in this paradigm translate into techniques called dynamic program slicing and execution backtracking. We discuss what these techniques are and how they can be automated. We present ways to obtain accurate dynamic slices of programs that may involve unconstrained pointers and composite variables. Dynamic slicing algorithms spanning a range of time-space-accuracy trade-offs are presented. We also propose ways in which multiple dynamic slices may be combined to provide further fault localization information. A new space-efficient approach to execution backtracking called "structured backtracking" is also proposed. Our experiment with the above techniques has also resulted in development of a prototype tool, SPYDER, that explicitly supports them.

1. INTRODUCTION

> The presence of bugs in programs can be regarded as a fundamental phenomenon; the bug-free program is an abstract theoretical concept like the absolute zero of thermodynamics, which can be envisaged but never attained.
>
> Jacob Schwartz, in [Sch71].

Such pessimism is not unfounded. A computer program is a complex automaton. As the size of the program grows, the complexity of the underlying automaton soon starts exploding. It becomes increasingly more and more difficult for the human mind to perceive, and keep track of, all possible state transitions of this complex state machine. Slight inattention on the programmer's part may result in serious faults. The difficulty is further compounded when programmers work with insufficient understanding of the problem and its solutions (which is unfortunately often the case).

Because of the presence of faults, the automaton may fail on certain inputs. The incorrect output produced by a faulty automaton is referred to as a *failure*, and faults in the automaton that cause the failure are referred to as *bugs* [ANS83]. In other words, a fault is a cause and a failure is a symptom. Faults get introduced in a program because of *errors* committed by programmers while translating specifications into implementations or because of errors in specifications themselves. *Testing* is the problem of finding inputs to the automaton that cause it to fail, and *debugging* is the problem of finding the faults once the failure has been detected.

Anyone who has ever engaged in serious programming knows that software testing and debugging are hard problems. As much as twenty-five to fifty percent of the total system development cost and time may be spent on these activities alone [Zel78, Boe81]. It has been suggested that the best solution to the problem of bugs

is to ensure that they never get into the program in the first place. To this effect, it has been argued that program verification or "proving programs correct" would eliminate the need for testing and debugging [Dij76, Hoa69, NR69]. Program verification requires that the program behavior be expressed as assertions on its input and output. Then one has to prove that the output assertion holds whenever the program is executed on an input that satisfies the input assertion. While this approach works well in theory, several problems arise in practice [DLP79]: Firstly, for many real-life programs, characterizing their behavior mathematically may not always be feasible, thus precluding the use of any mathematical reasoning on them. Further, constructing program-proofs can be an arduous task. Proofs are generally much more difficult to construct and understand than the programs themselves. Thus, proofs themselves can be bug-prone! Also, proofs are constructed with respect to program specifications. But specifications themselves can be "buggy." As Shapiro explains in [Sha83]:

> No matter what language we use to convey [the specifications], we are bound to make mistakes. Not because we are sloppy and undisciplined, as advocates of some program methodologies may say, but because of a much more fundamental reason: we cannot know, at any finite point in time, all the consequences of our current assumptions. A program is a collection of assumptions, which can be arbitrarily complex; its behavior is a consequence of these assumptions; therefore we cannot, in general, anticipate all possible behaviors of a given program. This principle manifests itself in numerous undecidability results, that cover most interesting aspects of program behavior for any non-trivial programming system [HR67].

More recently, in [Fet88], Fetzer goes as far as suggesting that "the success of program verification as a generally applicable and completely reliable method for guaranteeing program performance is not even a theoretical possibility."

Structured programming is another solution often cited to be the cure for bugs. While structured-programming certainly helps improve program comprehensibility (and thus, reliability), it would be naive to think that it reduces the need for software

testing and debugging. As Parnas points out in [Par85]: "Even in highly structured systems, surprise and unreliability occur because the human mind is not able to fully comprehend the many conditions that can arise because of the interactions of the components."

Thus one thing is evident: there is no escape from testing and debugging of programs if we are to have any confidence in them. That is precisely what these activities achieve — increase our confidence in a program by furnishing evidence that the program works correctly on testcases we supplied, and by extrapolation, on many other "similar" cases, if the testcases are selected judiciously. Much research has gone into answering the question of how to select testcases judiciously (see e.g. [DMMP87, How87, AB82]). But, surprisingly, much less work has been done on the equally important topic of how to localize bugs revealed by these testcases. Few tools or techniques are available to help programmers debug their programs. The tools that are available all basically provide breakpoints and traces as their main debugging aids [Kat79, MB79, Dun86, Wei82]. Unfortunately, these traditional mechanisms are often inadequate for the task of quickly isolating program faults.

In this dissertation, we present a new paradigm for debugging based on dynamic program slicing, fault guessing, and execution backtracking techniques. The importance of this paradigm lies in that each step in the paradigm can be automated individually thus automating the whole process and thus taking a lot of tedium out of the debugging process. It also provides a systematic approach to debugging, and thus attempts to introduce an element of science into it, which otherwise has largely been viewed as an art. Our experiment with these techniques has also resulted in development of a prototype tool that explicitly supports this paradigm.

1.1 A New Paradigm for Debugging

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 itself, 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, Gou75, Luk80].

Consider, for example, the program in the main window panel of Figure 1.1.[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 assignment on line 24 mistakenly computes b_sqr as sides[i].b $*$ sides[i].c instead of sides[i].b $*$ sides[i].b.

Suppose this program is executed for the testcase[2] when $N = 2$ and the sides of the two triangles are (3, 3, 3) and (6, 5, 4), respectively.[3] The sum of the areas of the two triangles for this testcase is incorrectly printed as 13.90 instead of 13.82. How should we go about locating the bug? Looking backwards from the **printf** statement on line 46, we find that 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 lengths of 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 to 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.

[1] The figures are X Window System window dumps of our prototype debugging tool, SPYDER, 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 chapters.

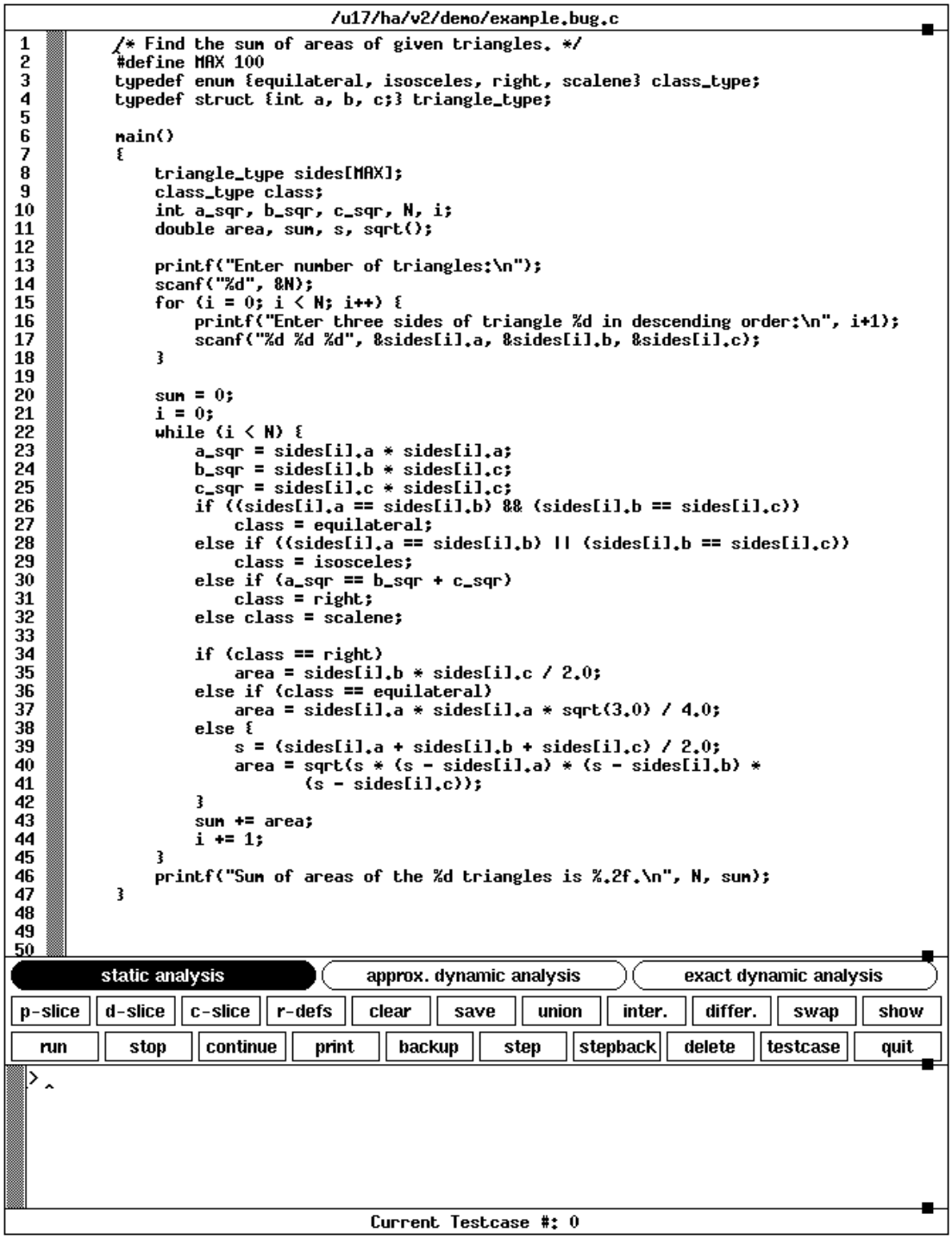

Figure 1.1 SPYDER screen with a sample C source program

To discover why, we may decide 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 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 repeatedly performed in this analysis:

1. Determine which statements in the code had an influence on the value of a given variable observed at a given location.

2. Select one of these statements at which to examine the program state.

3. Recreate the program state at the selected statement 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 that breakpoint. Our debugging job would become much easier if the debugger provided direct assistance in performing all three of these tasks. With explicit support for these activities, we will be able to pursue software debugging in a systematic fashion. Our prototype debugger, SPYDER, provides the user with exactly this assistance. The first task—given a variable and a program location, determining which statements in the program affect the value of that variable at that location when the program is executed for a given testcase—is referred to as *Dynamic Program Slicing*. SPYDER can find dynamic slices for us automatically. It also provides mechanisms to help programmers select appropriate statements for further examination. 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.

Figure 1.2 depicts the proposed systematic debugging paradigm. The first step upon detection of a program failure is to translate the external symptoms of the program failure into the corresponding internal symptoms in terms of data or control

problems in the program. Then one of the internal symptoms is selected as the slicing criterion and the corresponding dynamic slice is obtained. After examining the slice, a statement is selected at which to examine the program state, and the program state is restored to that when control last reached that statement. Examining values of some variables in the restored state may reveal the fault, otherwise the user may choose to further examine the restored state, or guess a new fault, or select a new slicing criterion, and repeat the cycle until the fault is localized.

1.2 Scope and Goals of this Research

In this dissertation we are primarily concerned with how to automate each step in the debugging paradigm outlined above for programs written in sequential and procedural programming languages such as Pascal and C. Issues concerning debugging of parallel or distributed programs, programs written in functional or logic programming languages, or debugging optimized programs, are not addressed.

An underlying goal of this research has been not only to enhance the state-of-the-art in software debugging but also to enhance its state-of-the-practice. Thus an important focus of this research has also been to develop techniques that are practicable—techniques that are not only sound in theory but also practical to implement. An equally important goal has been to develop techniques that can be applied to programs written in realistic procedural languages such as Pascal and C. For this reason, we focussed on techniques that would work for programs that involved arrays, pointers, records, unions, procedures, etc., for it is hard to imagine real programs written in a procedural language that do not use these language features. To show that the techniques we proposed are indeed useful as well as practical, we also wanted to implemented a prototype tool that explicitly supported them. Our goal was not to build a production quality tool but to demonstrate the usefulness and feasibility of automating significant steps in the debugging paradigm proposed.

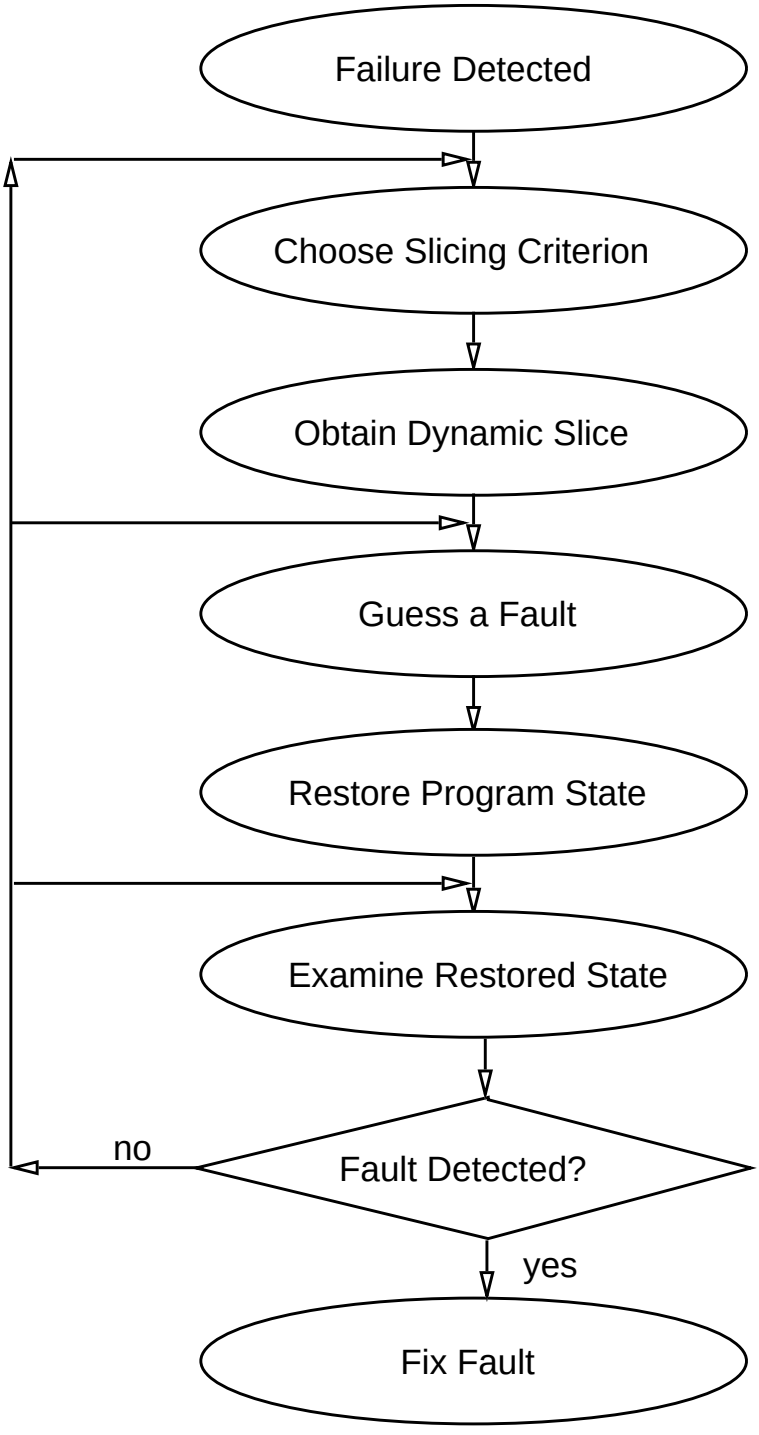

Figure 1.2 A Debugging Paradigm

1.3 Contributions of this Research

Following are the main contributions of this research:

- *Slicing–Guessing–Backtracking Paradigm:* We observed that the think–hypothesize–verify cycle that programmers follow during debugging can be translated into a dynamic slicing–fault guessing–execution backtracking paradigm. The significance of this observation, as mentioned earlier, lies in the fact that each step in the paradigm can be automated, thus making it possible to largely automate the whole debugging process.

- *Dynamic Program Slicing:* We developed techniques to perform dynamic program slicing of programs that may involve unconstrained pointers, arrays, records, unions, and procedures. We designed a number of algorithms spanning a range of time–space–accuracy trade-offs. We also proposed a new space-efficient data-structure called the reduced-dynamic-dependence-graph used in obtaining dynamic program slices.

- *Data and Control Slices:* Oftentimes observing complete program slices may be overwhelming, so we introduced the notions of data- and control-slices that are smaller than program slices and much easier to follow.

- *Combining Dynamic Slices:* A dynamic slice is obtained with respect to four arguments: a program, a variable, a location, and a testcase. We described several ways in which we can fix any three of these arguments and vary the remaining argument to generate multiple related dynamic slices, which may then be combined in several ways to provide further fault localization information.

- *Structured Backtracking:* We also proposed a new approach to execution backtracking called structured-backtracking that is both space- and time-efficient compared to the execution history approach.

- SPYDER*: A Prototype Implementation:* We also implemented a prototype debugging tool, SPYDER, that explicitly supports the slice-guess-backtrack paradigm. The tool works for the C programming language including pointers, arrays, structures, and procedures. It is interactive, easy to use, and has an X Window System based user interface.

1.4 Organization of this Dissertation

The rest of this dissertation is organized as follows: In the next chapter we present a brief survey of the state-of-the-art-and-practice in software debugging and describe how our work relates to that of others. The three steps in the debugging paradigm mentioned above are discussed in Chapters 3–7. Chapter 3 discusses several approaches to obtaining dynamic slices for the simple case when the program involves only scalar variables. Chapter 4 extends this discussion to handling composite and pointer variables. In Chapter 5, we introduce the notions of data and control slices. Chapter 6 examines how multiple dynamic slices may be usefully combined to provide further fault localization information. In Chapter 7, we discuss two approaches to implementing execution backtracking. Then in Chapter 8, we describe our prototype debugging tool, SPYDER, that explicitly supports the debugging paradigm outlined above. Finally, Chapter 9 concludes the dissertation with a discussion of lessons learned from this research and outlining some ideas on future research.

2. DEBUGGING: STATE-OF-THE-ART-AND-PRACTICE

If the progress made in a field is measured by the rate at which new ideas are developed, perfected, and absorbed in the society for wide spread use, then, quite disappointingly, the area of software debugging has not made much progress. Essentially the same techniques developed thirty years ago for debugging assembly language programs are being used today. That is quite surprising given the very high cost (in terms of programmer time) associated with debugging. The basic notion of user-controlled breakpoints, upon which almost all commonly used debuggers today are based, was introduced by assembly language debuggers FLIT [SD60] and DDT [SW65] developed in the late 50's. We have not come much farther from there. In this section we present a brief survey of the state-of-the-art-and-practice in software debugging and outline how our work relates to it.

2.1 Traditional Debugging

Though the use of core-images to analyze faulty program behavior has somewhat faded away, use of print statements still seems to be the most prevalent debugging technique. Print statements are placed at "strategic" locations in the program, to display intermediate values of important variables and to indicate the flow of control. In this way users get visual feedback about the internal workings of the program that helps them localize the fault. This technique is simple and effective in most cases. Besides, it comes free with the language. It becomes tedious to use, however, as the program has to be recompiled every time new print statements are included. Further, these print statements usually need to be deleted (or commented out) from the program once the bug has been detected and fixed.

Interactive symbolic debuggers allow runtime control over display of debugging information without requiring the user to modify the source code [Bea83, Dun86, Kat79]. They provide the capability of setting *breakpoints* in the program as their basic debugging facility. The user can specify one or more statements as breakpoints, so whenever the control reaches any of these locations the program execution is suspended and control is passed to the debugger. The user can then inspect the program state by displaying current values of variables. Most debuggers also allow the control stack to be displayed. Many also allow program state to be explicitly modified by the user during execution. After examining (and possibly changing) values, the user can set new breakpoints and continue execution.

Tracing is another common facility provided by most symbolic debuggers. The user can specify tracepoints and the corresponding trace information to be displayed at these points. Trace information may include a simple message about control reaching the tracepoint, or it may include values of certain variables at these points. Whenever control reaches a tracepoint, the corresponding trace information is displayed and the execution is continued. Note that a tracepoint is a special case of a breakpoint where the specified trace information is displayed and the execution continued automatically.

Some symbolic debuggers also allow certain boolean conditions to be associated with tracepoints. Whenever control reaches such a tracepoint the corresponding condition is evaluated. If the condition is true the specified trace information is displayed and the execution continued, otherwise the execution is resumed without displaying anything. Similarly, conditions can also be associated with breakpoints in many debuggers.

Another common facility provided by most interactive debuggers is *single-stepping*. The program is executed one statement at a time and control returned to the debugger after every statement. Note that single-stepping is also a special case of breakpoints where breakpoints are set after consecutive source statements.

Most symbolic debuggers provide limited control over what actions can be performed when a breakpoint is reached. Dalek [OCHW91], a symbolic debugger built on top of the Gnu Debugger Gdb, provides a full programming language including conditionals, loops, blocks, procedures, functions, and variables, to program higher-level actions to be performed at breakpoints. It also provides support for events to form high level abstractions during program executions.

Interpreted environments, where the program source is interpreted instead of being compiled, provide a much greater degree of flexibility for debugging purposes. Integrated Programming Environments such as INTERLISP [Tei78], The Programmer's Assistant [Tei72], The Cornell Program Synthesizer [TR81], and Saber C provide integrated support for program editing, execution, and debugging. If an error is encountered during the program execution, it is possible to correct the erroneous function immediately and continue execution from there on. The usual debugging cycle for compiled programs — edit, compile, execute, and debug — is considerably shortened in such systems.

Window- and mouse-based symbolic debuggers using bitmapped displays like Dbxtool [AM86], Pi [Car83, Car86], and Saber C bring about a significant improvement over traditional command-driven debuggers from the user-interface perspective. With such debuggers instead of having to type the debugging commands the user can simply select them using the mouse and button-clicks. Source code is simultaneously displayed in one or more windows, and the current control location is highlighted, e.g., with an arrow pointing at the next statement to be executed. In Dbxtool [AM86] and Saber C, the user can select any variables or expressions in the source-window with the help of the mouse and request their values to be displayed. Pi [Car86] provides context dependent pop-up menus of variable names, visible in the current scope, for selecting and displaying their values.

2.2 Pictorial Debugging

Some debuggers like VIPS [ISO87, SI91] and PROVIDE [Moh88] go a step further and attempt to present both the control-flow and the data-structures pictorially to the user. The graphical representations on the screen are updated dynamically as the program execution proceeds. For example, a stack of integers may be represented by a sequence of rectangles containing numbers and the stack-top may be represented by an arrow pointing at the top rectangle in the stack. The arrow moves up or down (and new rectangles added or deleted) as numbers are pushed on or popped off the stack. Such debuggers are still in their infancy, and presently are able to handle only small programs with simple data-structures. Research in the area of algorithm animation [BS85, Bro88, LD85] has much to offer in this respect.

2.3 Debugging with Specifications

A debugging technique called "two-dimensional pinpointing" [LST91] aims at locating inconsistencies in software that is structured in levels. It requires that formal specifications defining the program's desired behavior at each structural level be provided. The technique works by first checking the actual execution behavior of the program under a test sequence against the top level specifications. If an inconsistency is observed, new top level specifications may be added and the process repeated to further narrow down the search to a specific program unit. When no new specifications may be added, the same process is repeated at the next lower level but only for the program unit whose specification was violated at the top level. This process continues at successive lower levels until a fault is detected.

2.4 Algorithmic Debugging

Shapiro's "Divide & Query" interactive fault-diagnosis approach [Sha83] uses a kind of binary search on the computation tree of the program to localize bugs. It relies upon the user to verify the correctness of intermediate function calls based on

their input and output values. It is based on the premise that if the computation performed by a subprogram, *proc*, is correct then computations performed by each of the subprograms invoked by *proc* are also correct. On the other hand, if the computation performed by *proc* is incorrect, then at least one of the subprograms it invokes is incorrect. The diagnosis algorithm works by first selecting a node in the computation tree that divides the tree into roughly two equal parts. Then the user is presented with the input and output values of the function call and asked to check if the function computation is correct. If it is incorrect, the algorithm is recursively applied to the subtree rooted at that node, otherwise that subtree is removed from further consideration and the same algorithm is applied to the rest of the tree. This cycle is repeated until the fault is located.

The above approach works particularly well for side-effect free languages like the logic programming language Prolog. But it can also be applied to imperative languages such as Pascal at the procedure-call level [SKF90]. Using this approach the fault can be localized to a procedure that contains the fault. Then other debugging techniques may be used to further localize the bug within that procedure. Korel and Laski's STAD (A System for Testing and Debugging) provides a similar fault-diagnosis approach at the intra-procedure level for a subset of Pascal [KL91, KL88b], except that instead of asking users to verify the input-output correctness of procedure calls, it asks them questions like if a given assignment is the correct reaching definition of a variable at a given location, or if the control has correctly reached a given location, etc. Their approach is targeted to find simple operator and operand faults in the program.

2.5 Knowledge-Based Debugging

Attempts have also been made to apply artificial intelligence techniques to the debugging problem. In particular, many experimental systems have been developed using the knowledge-based approach (see, e.g., [Sev87, DE88]). However, all these

systems can presently handle only a restricted class of bugs in small programs. We discuss a few representative knowledge-based debugging systems below.

PUDSY (Program Understanding and Debugging System) [Luk80] maintains a knowledge-base of simple programming schemas for solving simple programming tasks, e.g., swapping two values, finding the maximum element of an array, etc. For each such schema an assertion describing the function performed by the schema is also stored. PUDSY takes a program and an assertion describing its specifications, and searches for code patterns in the program, matching those stored in the knowledge-base. Then it constructs an assertion describing the program behavior by combining the assertions fetched from the knowledge-base. The constructed assertion is matched with the given assertion, and if the two differ the process is traced back to suggest possible bugs.

Proust [JS85] uses the opposite approach. It also takes a program and a formal specification of the program as inputs, but it tries to "synthesize" a program from the specifications, while trying to match the synthesized program with the given program. The synthesis is performed with the help of a knowledge-base of "programming plans" for solving simple "goals". If the synthesis fails to produce a program matching the given program, the discrepancies are analyzed, and possible faults are suggested.

The Programmer's Apprentice [RW88] also uses a similar approach. Given a "plan" for the intended program, it infers the existence of bugs in the program by performing a kind of pattern-matching between the plan and the program itself. Note that all three systems above perform static program analysis irrespective of any runtime inputs.

Falosy [STJ83], on the other hand, expects both the actual and the expected output of the program on the failed testcase to be supplied. It compares the two outputs and tries to infer the fault from the differences between the two, with the help of a knowledge-base of error-cause heuristics. The knowledge-base stores a list of fault-symptoms (output discrepancies), and a list of possible faults for each symptom. Faults are described in terms of prototype (faulty) code schemas. The system searches

for faults in the program corresponding to the output discrepancies observed. It performs pattern-matching over code-schemas, as in Proust, to locate the faults.

2.6 Program Slicing

The static slicing approach to program debugging was proposed by Weiser [Wei84, Wei82]. A program slice is helpful in debugging because it presents the user with only that subset of the program that may have some effect on the value of an erroneous variable. It expedites debugging by narrowing the user's attention to only the code segments that are relevant to the fault. As slicing constitutes a major component of our debugging paradigm, it is addressed again in more detail in Chapter 3.

Weiser's algorithm to compute slices was based on iterative solutions of data-flow equations. Ottenstein and Ottenstein presented an algorithm in terms of graph reachability in the program dependence graph, but they only considered the intra-procedural case [OO84] (we describe this algorithm in Chapter 3 in the context of static slicing). Horwitz, Reps, and Binkley 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 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, software metrics, etc. (see, e.g., [Wei84, BC85, HPR89b, GL89, OT89, LOS86]).

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 definition of data-dependence remains the same in both cases.

Korel and Laski extended Weiser's static slicing algorithms for the dynamic case [KL90]. Their definition of a dynamic slice is different from ours (see Chapter 3). They require that if any one occurrence of a statement in the execution history is included in the slice then all other occurrences of that statement are automatically included in the slice, even when the value of the variable in question at the given location is unaffected by those other occurrences. We examine the consequences of this requirement in Chapter 3 (Section 3.4.4.1).

Several people have also investigated the semantic basis of program slicing [Ven91, RY88, CF89, Sel89].

2.7 Program Dicing

Slicing uses the information that the value of a variable is incorrect to narrow down the search for the fault. It does not, however, use the information that values of many other variables may be correct. Lyle and Weiser proposed the notion of program dicing [LW87] which attempts to further narrow down the search for the fault using information gained during testing about which variables in the program are observed to have incorrect values and which have correct values when the program is executed on various testcases. The search for the fault is narrowed down by removing statements that belong to slices with respect to correct variables from the slice with respect to an incorrect variable. We discuss this approach again in Chapter 6.

2.8 Anomaly Detection

In the static anomaly detection approach, a program is analyzed using data-flow analysis to detect certain conditions in the program that are generally indicative of errors or inefficiencies [OF76, FO76]. For example, data-flow analysis can detect if an

assignment statement references a variable before that variable is assigned a value, or if there are two assignments, A_1 and A_2, such that both assign a value to the same variable, *var*, and control always flows from A_1 to A_2 without encountering any reference to *var*.

As it is not always possible with static analysis to determine if a given path in the program is feasible, the above approach may also find "anomalies" in the program that cannot arise during execution. Dynamic anomaly detection techniques, on the other hand, detect anomalous conditions that arise during actual program executions [Hua79, CC87a]. These techniques instrument the program with extra code that checks for any anomalous behavior, such as the conditions mentioned above, to occur during the program execution. The anomalies detected this way are almost always indicative of errors or inefficient program behavior. At the same time, as program instrumentation can only analyze actual program executions it does not guarantee detection of all possible anomalies in the program.

2.9 Execution Backtracking

Many debugging systems in the past have also supported execution backtracking facilities. EXDAMS, an interactive debugging tool for Fortran developed in the late 1960s, 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.

Miller and Choi's PPD [MC88] also performs flow-back analysis like EXDAMS but it uses 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 generate the corresponding segments of the execution history.

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. INTERLISP [Tei78] and the Cornell Program Synthesizer [TR81] also provide facilities to undo operations. All these systems maintain a fixed-length history list of side-effects caused by operations. As new events occur, the existing events on the list are aged, with oldest events "forgotten." Thus returning to points arbitrarily far back in the execution may not be possible. In Chapter 7 we present a structured backtracking approach to execution backtracking which attempts to overcome this problem.

IGOR [FB88] and COPE [ACS84] also provide execution backtracking by performing periodic checkpointing of memory pages or file blocks modified during program execution. This approach, while suitable for undoing effects of whole programs, may be inefficient for performing statement-level backtracking.

2.10 Summary

The gap between state-of-the-art and state-of-the-practice in software debugging is a very wide one. Most commonly available debuggers today restrict themselves to providing facilities described in Section 2.1 (Traditional Debugging). They basically provide some variants of the breakpoint mechanism coupled with a display facility. Their use generally results in a substantial saving in debugging time as compared to

deciphering core-images or using print statements. When debugging large programs, however, decisions about where to set a breakpoint, or to determine the right balance between trace information and trace frequency, etc., may not be easy ones to make. In this dissertation, we describe how some of the tasks commonly performed during debugging may be automated to make software debugging less arduous.

As the problem of debugging parallel, logic, functional, or optimized programs is not in the scope of this dissertation, we have not included related work in these areas here, though research is also being carried out in these areas (see, e.g., [MH89, AS89, Hen82, Zel84, Cou88]).

3. SIMPLE DYNAMIC SLICING

Often during debugging, value of a variable, *var*, at a program statement, S, is observed to be incorrect. The program slice with respect to *var* at S gives the set of program statements that directly or indirectly affect the value of *var* as observed at S [Wei82]. But this notion of a program slice does not make any use of the particular inputs that revealed the error. It is concerned with finding all statements that *could* influence the value of the variable occurrence for *any* inputs, not all statements that *did* affect its value for the *current* inputs. Unfortunately, the size of a slice so defined may approach that of the original program, and the usefulness of a slice in debugging tends to diminish as the size of the slice increases. Therefore, here we examine a narrower notion of "slice," consisting only of statements that influence the value of a variable occurrence for specific program inputs.[1] This problem is referred to as *Dynamic Program Slicing* to distinguish it from the original problem of *Static Program Slicing*.

Conceptually a program may be thought of as a collection of *threads*, each computing a value of a program variable. Several threads may compute values of the same variable. Portions of these threads may overlap one-another. The more complex the control structure of the program, the more complex the intermingling of these threads. Static program slicing isolates all possible threads computing a particular variable. Dynamic slicing, on the other hand, isolates the unique thread computing the variable for the given inputs.

During debugging programmers generally analyze the program behavior under the testcase that revealed the error, not under any generic testcase. The concrete testcase

[1] A slice with respect to a set of variables may be obtained by taking the union of slices with respect to individual variables in the set.

that exercised the bug helps them focus their attention to the particular "cross-section" of the program that contains the bug.[2] Consider the following scenario: A friend while using a program discovers an error. He finds that the value of a variable printed by a statement in the program is incorrect. After spending some time trying to find the cause without luck, he comes to you for help. Probably the first thing you would demand from him is the testcase that revealed the bug. If he only gave you the print-statement and the variable with the incorrect value, and didn't disclose the particular inputs that triggered the error, your debugging task would clearly be more difficult. This suggests that while debugging a program we probably try to find the *dynamic* slice of the program in our minds. This simple observation also highlights the value of the automatically determining dynamic program slices. The distinction between static and dynamic slicing and the advantages of the latter over the former are further discussed in Section 3.4.

In this chapter we examine several approaches to compute dynamic program slices. We first discuss a program representation called the Program Dependence Graph used in computing static program slices, and present the static slicing algorithm. Then we present two simple extensions to the static slicing algorithm to compute dynamic slices. But these algorithms compute overlarge slices—they may include extra statements in the dynamic slice that shouldn't be there. We then present a data-structure called the Dynamic Dependence Graph and an algorithm that uses it to compute accurate dynamic slices. Size of a Dynamic Dependence Graph depends on the length of the program execution, and thus, in general, it is unbounded; so we introduce a mechanism to construct what we call a Reduced Dynamic Dependence Graph which requires limited space, proportional to the number of distinct dynamic slices that arise during the current program execution, not to the length of the execution. Finally, we present an efficient way to construct a Reduced Dynamic Dependence Graph. The

[2]When we say the slice contains the bug, we do not necessarily mean that the bug is textually contained in the slice; the bug 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. This is discussed in Chapter 5.

four approaches to dynamic slicing presented span a range of solutions with varying space-time-accuracy trade-offs.

For simplicity in exposition, we restrict our attention to the following language with if-then-else, while-do, composition, assignment, read, and write statements.

$$
\begin{array}{lcl}
\textit{Program} & \longrightarrow & \textit{Declarations}\ \textbf{begin}\ \textit{Stmt\_list}\ \textbf{end}. \\
\textit{Stmt\_list} & \longrightarrow & \textit{Stmt};\ \textit{Stmt\_list} \mid \epsilon \\
\textit{Stmt} & \longrightarrow & \textit{Simple\_stmt} \mid \textit{If\_stmt} \mid \textit{While\_stmt} \\
\textit{Simple\_stmt} & \longrightarrow & \textit{Assgn\_stmt} \mid \textit{Read\_stmt} \mid \textit{Write\_stmt} \\
\textit{Assgn\_stmt} & \longrightarrow & \textit{Var} := \textit{Exp} \\
\textit{Read\_stmt} & \longrightarrow & \textbf{read}(\textit{Var}) \\
\textit{Write\_stmt} & \longrightarrow & \textbf{write}(\textit{Var}) \\
\textit{If\_stmt} & \longrightarrow & \textbf{if}\ (\textit{Predicate\_exp})\ \textbf{then}\ \textit{Stmt\_list}\ \textbf{else}\ \textit{Stmt\_list}\ \textbf{end\_if} \\
\textit{While\_stmt} & \longrightarrow & \textbf{while}\ (\textit{Predicate\_exp})\ \textbf{do}\ \textit{Stmt\_list}\ \textbf{end\_while} \\
\textit{Predicate\_exp} & \longrightarrow & \textit{Exp} \\
\textit{Exp} & \longrightarrow & \textit{Exp}\ \textit{Binary\_op}\ \textit{Exp} \mid \textit{Unary\_op}\ \textit{Exp} \mid \textit{Var} \mid \textit{Const}
\end{array}
$$

Our discussion here is easily extensible to other statement types like do-while, for, case, etc., and to expressions with side-effects (e.g., expressions with pre- and post-increment or decrement operators as in C). Techniques for handling pointers, arrays, structures, and procedures are discussed in the next chapter.

3.1 Notation

In the following sections we use a *let-in* construct (adapted from a similar construct in the programming language ML [MTH90]). Consider the following generic use of *let*:

let <*declarations*> *in* <*expression*>

Here, <*declarations*> consists of a sequence of name bindings that may be used inside <*expression*>. The scope of these bindings is limited to <*expression*>. The

result of evaluating <*expression*> is returned as the value of the *let* construct. For example, the following expression evaluates to 5.

$$let\ a = 2,\ b = 3\ in\ a + b$$

Names may also be bound using "pattern matching" between two sides of the symbol =. For example, if the complex number $X + Yi$ is represented by the tuple (X, Y), then the sum of two complex numbers $complex_1$ and $complex_2$ may be defined as follows:

$sum(complex_1, complex_2) =$

$$let\ complex_1 = (real_1, imaginary_1),\ complex_2 = (real_2, imaginary_2)$$
$$in\ (real_1 + real_2, imaginary_1 + imaginary_2)$$

In the above expression, $real_1$, $imaginary_1$, $real_2$, and $imaginary_2$ were all defined using pattern matching.

We also use $\bigcup$ notation to denote set unions. For example, if S= $\{x_1, x_2, \ldots, x_n\}$, then we have:

$$\bigcup_{x \in S} f(x) \equiv f(x_1) \cup f(x_2) \cup \ldots \cup f(x_n)$$

$\bigcup$'s may also be composed. For example, if S_1= $\{x_1, x_2\}$ and S_2= $\{y_1, y_2\}$, then we have:

$$\bigcup_{x \in S_1} \bigcup_{y \in S_2} g(x, y) \equiv g(x_1, y_1) \cup g(x_1, y_2) \cup g(x_2, y_1) \cup g(x_2, y_2)$$

Or, we write the same thing as:

$$\bigcup_{\substack{x \in S_1 \\ y \in S_2}} g(x, y) \equiv g(x_1, y_1) \cup g(x_1, y_2) \cup g(x_2, y_1) \cup g(x_2, y_2)$$

3.2 Preliminary Definitions

3.2.1 Flow Graph

Flow-graph *Flow* of a program P is a four-tuple (V, A, En, Ex) where V is the set of vertices corresponding to simple statements and predicate expressions (that

correspond to non-terminals *Simple_stmt* and *Predicate_exp* in the grammar above),[3] A is the set of directed edges between vertices, and En and Ex are the distinguished *entry* and *exit* nodes in V respectively. If there is an arc from node v_i to node v_j (i.e., $(v_i,v_j) \in A$)), then control can pass from node v_i to node v_j during program execution. We define *Flow* in a syntax-directed manner as follows:

$Flow(Program) = Flow(Stmt\_list)$

$Flow(S) = (\{S\}, \phi, S, S)$, if S is a simple statement

$Flow(S_1; S_2) =$

$\quad let\ Flow(S_1) = (V_1, A_1, En_1, Ex_1),$

$\quad\quad Flow(S_2) = (V_2, A_2, En_2, Ex_2),$

$\quad\quad V' = V_1 \cup V_2,$

$\quad\quad A' = A_1 \cup A_2 \cup \{(Ex_1, En_2)\}$

$\quad in\ (V', A', En_1, Ex_2)$

$Flow($**if** P **then** S_1 **else** S_2 **end_if**$) =$

$\quad let\ Flow(S_1) = (V_1, A_1, En_1, Ex_1),$

$\quad\quad Flow(S_2) = (V_2, A_2, En_2, Ex_2),$

$\quad\quad Ex$ be a new dummy node,

$\quad\quad V' = V_1 \cup V_2 \cup \{P, Ex\},$

$\quad\quad A' = A_1 \cup A_2 \cup \{(P, En_1), (P, En_2), (Ex_1, Ex), (Ex_2, Ex)\}$

$\quad in\ (V', A', P, Ex)$

$Flow($**while** P **do** S **end_while**$) =$

$\quad let\ Flow(S) = (V, A, En, Ex),$

$\quad\quad V' = V \cup \{P\},$

[3] In program optimization applications vertices of a flow-graph correspond to *basic-blocks* in the program. But for our purposes, it is more convenient to associate vertices with simple-statements and predicates in the program.

$$A' = A \cup \{(P,En), (Ex,P)\}$$
$$in\ \ (V', A', P, P)$$

Example: Consider the program in Figure 3.1 (we will refer to this program several times later in the chapter). Symbols f_i and g_i in the assignment statements are used here to denote some unspecified side-effect-free functions with which we are not presently concerned; only the names of variables used in the computation are relevant. Labels S1, S2, etc. are included only for reference; they are not part of the program. Figure 3.2 shows the flow-graph for this program (ignore the node annotations U, D and R for the moment). Node $2'$ and $5'$ are the dummy exit nodes for **if** statements 2 and 5 respectively. □

3.2.2 Use and Def Sets

Each vertex in the flow-graph has a *use* and a *def* set associated with it. The Use set of a vertex consists of all variables that are referenced during the computation associated with the vertex, and the Def set consists of the variable computed at the vertex, if any. *use* and *def* are also defined in a syntax-directed manner for *Simple_stmt* and *Predicate_exp* syntactic categories:

$$use(Var := Exp) = use(Exp)$$
$$def(Var := Exp) = \{Var\}$$

$$use(\mathbf{read}(Var)) = \phi$$
$$def(\mathbf{read}(Var)) = \{Var\}$$

$$use(\mathbf{write}(Var)) = \{Var\}$$
$$def(\mathbf{write}(Var)) = \phi$$

$$use(Predicate\_exp) = use(Exp)$$
$$def(Predicate\_exp) = \phi$$

```
        begin
S1:         read(X);
S2:         if (X < 0)
            then
S3:             Y := f1(X);
S4:             Z := g1(X);
            else
S5:             if (X = 0)
                then
S6:                 Y := f2(X);
S7:                 Z := g2(X);
                else
S8:                 Y := f3(X);
S9:                 Z := g3(X);
                end_if;
            end_if;
S10:        write(Y);
S11:        write(Z);
        end.
```

Figure 3.1 Example Program 1

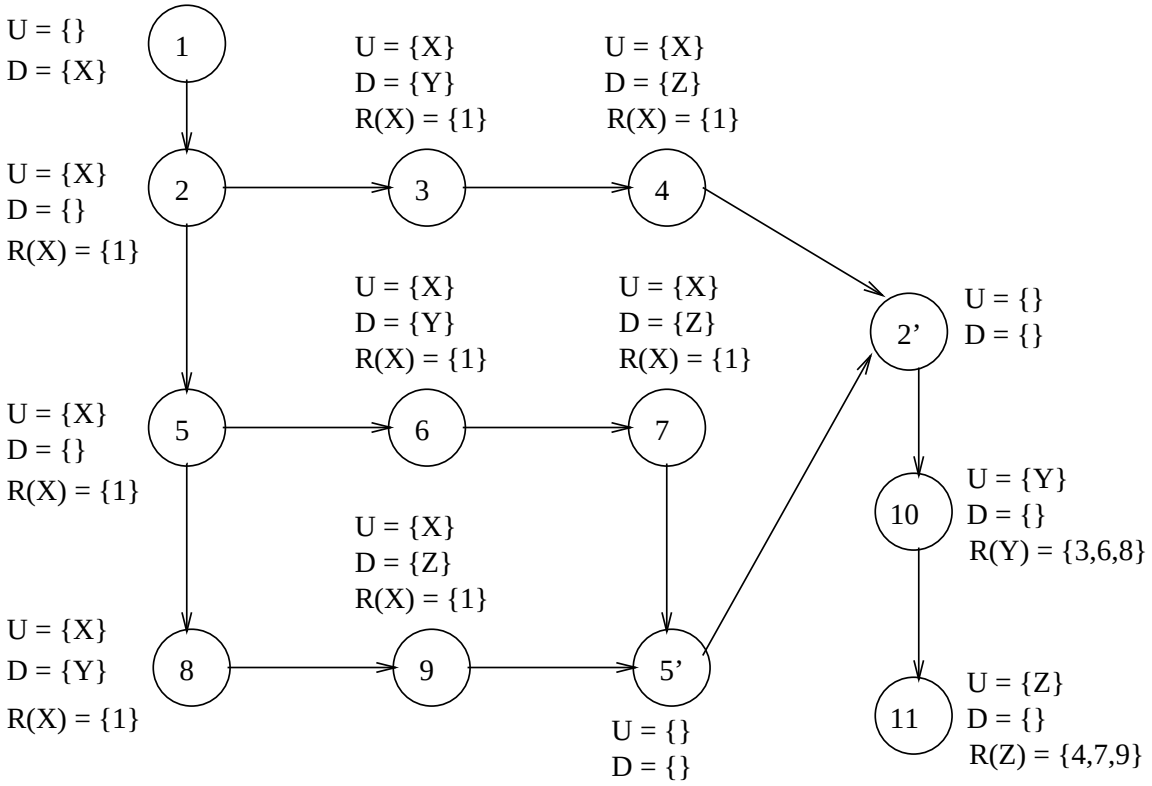

Figure 3.2 Flow Graph with use(U), def(D) and $StaticReachingDefns$(R) sets for the program in Figure 3.1

$use(Exp_1\ Binary\_op\ Exp_2) = use(Exp_1) \cup use(Exp_2)$

$use(Unary\_op\ Exp) = use(Exp)$

$use(Var) = \{Var\}$

$use(Const) = \phi$

use and *def* sets of dummy vertices in the flow-graph (introduced as exit nodes for **if-then-else** statements) are null sets.

Example: Consider the program in Figure 3.1. Figure 3.2 shows the *use* and *def* sets, denoted by U and D respectively, associated with all nodes in the flow-graph (ignore node annotations labeled R for the moment). Statement 6, for example, defines variable Y and uses variable X in computing the value assigned to Y. So we have U = {X} and D = {Y} for node 6. □

3.2.3 Reaching Definitions

Given a flow-graph, $\mathcal{F}$, a node n in $\mathcal{F}$, and a variable, *var*, we define *StaticReachingDefns*(*var*, n, $\mathcal{F}$), the set of all reaching definitions of variable *var* at node n in flow-graph $\mathcal{F}$, to be the set of all those nodes in $\mathcal{F}$ at which variable *var* is assigned a value and control can flow from that node to node n without encountering any redefinitions of *var*. More precisely:

StaticReachingDefns(*var*, n, $\mathcal{F}$) =

  let $\mathcal{F} = (V, A, En, Ex)$

  in $\bigcup_{(x,n)\in A}$ (*if* $var \in def(x)$

    then $\{x\}$

    else *StaticReachingDefns*(*var*, x, $(V, A-\{(x,n)\}, En, Ex)$))

Example: Figure 3.2 shows the *StaticReachingDefns* sets, denoted by R, for all nodes with nonempty *use* sets. For example, consider node 10: It has one variable Y in its *use* set. There are three definitions of Y, at nodes 3, 6 and 8, and all three definitions

can reach node 10, via paths 3 4 2' 10, 6 7 5' 2' 10, and 8 9 5' 2' 10 respectively, without encountering any redefinitions of Y along these paths. So we have R(Y) = {3,6,8} at node 10. □

3.2.4 Data Dependence

A Data Dependence Graph, *DataDep*, of a program, P, is a pair (V, D), where V is the same set of vertices as in flow-graph of P, and D is the set of edges that reflect data-dependencies between vertices in V. If there is an edge from vertex v_i to vertex v_j, it means that the computation performed at vertex v_i directly depends on the value computed at vertex v_j.[4] Or, more precisely:

$$
\begin{aligned}
&DataDep(P) = \\
&\quad let\ Flow(P) = (V,\ A,\ En,\ Ex), \\
&\qquad D = \bigcup_{\substack{n \in V \\ var \in use(n) \\ x \in StaticReachingDefns(var,\ n,\ Flow(P))}} \{(n,x)\} \\
&\quad in\ (V,\ D)
\end{aligned}
$$

Example: Consider the program in Figure 3.1 whose flow-graph is shown in Figure 3.2. Figure 3.3 shows the corresponding Data Dependence Graph. Consider node 10, for example: As shown in Figure 3.2, there is only one variable Y in its *use* set, and *StaticReachingDefns* set for Y at node 10 is {3, 6, 8}. So, there are three data dependence edges from node 10 to nodes 3, 6, and 8, respectively, in Figure 3.3. □

3.2.5 Control Dependence

A Control Dependence Graph, *ControlDep*, of a program, P, is a three-tuple (V, C, In), where V is the same set of vertices as in flow-graph of P, C is the set of edges that reflect control dependencies between vertices in $V \cup \{In\}$, and In is a dummy initial node that is *not* in V(unlike a flow-graph where En and Ex both belong to

[4] At other places in the literature, particularly that related to vectorizing compilers, e.g., [KKL+81, FOW87], direction of edges in the Data Dependence Graphs is reversed, but for the purposes of program slicing our definition is more suitable, as will become apparent later.

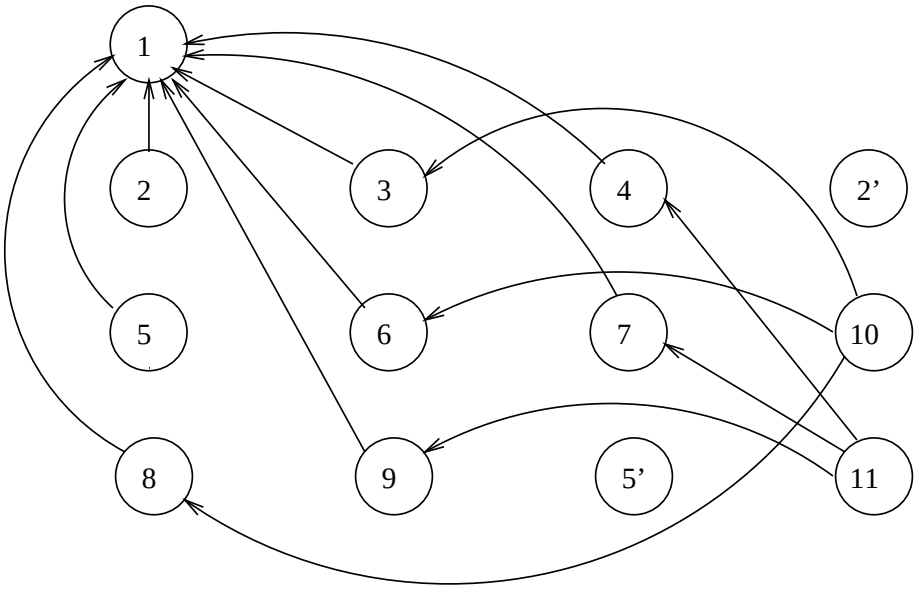

Figure 3.3 Data Dependence Graph for the flow-graph in Figure 3.2

V). If there is an edge from v_i to v_j in *ControlDep*, it means the execution of node v_i directly depends on the boolean value of the predicate at node v_j. *ControlDep* is defined in a syntax-directed manner as follows:

$ControlDep(Program) = ControlDep(Stmt\_list)$

$ControlDep(S) = (\{S\}, \{(S,In)\}, In)$, if S is a simple statement

$ControlDep(S_1; S_2) =$

$\quad let\ ControlDep(S_1) = (V_1, C_1, In),$

$\quad\quad ControlDep(S_2) = (V_2, C_2, In)$

$\quad in\ (V_1 \cup V_2, C_1 \cup C_2, In)$

$ControlDep($**if** P **then** S_1 **else** S_2 **end_if**$) =$

$\quad let\ ControlDep(S_1) = (V_1, C_1, In),$

$\quad\quad ControlDep(S_2) = (V_2, C_2, In),$

$\quad\quad V' = V_1 \cup V_2 \cup \{P\},$

$\quad\quad C' = (\bigcup_{(x,y) \in C_1 \cup C_2} if\ y{=}In\ then\ \{(x,P)\}\ else\ \{(x,y)\}) \cup \{(P,In)\}$

$\quad in\ (V', C', In)$

$ControlDep($**while** P **do** S **end_while**$) =$

$\quad let\ ControlDep(S) = (V, C, In),$

$\quad\quad V' = V \cup \{P\},$

$\quad\quad C' = \bigcup_{(x,y) \in C} (if\ y{=}In\ then\ \{(x,P)\}\ else\ \{(x,y)\}) \cup \{(P,In)\}$

$\quad in\ (V', C', In)$

Example: Figure 3.4 shows the Control Dependence Graph for the program in Figure 3.1. For example, statements 6, 7, 8 and 9 are immediately nested under the predicate at statement 5, so there are control dependence edges from nodes 6, 7, 8 and 9 to node 5. Statement 5 itself is immediately nested under predicate at statement 2, so there is an edge from node 5 to node 2 in the Control Dependence Graph. □

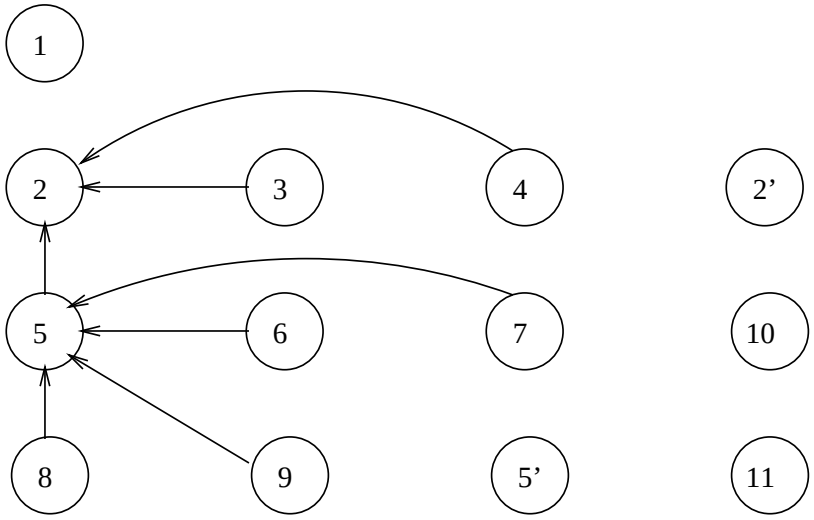

Figure 3.4 Control Dependence Graph of the Program in Figure 3.1

Note that every vertex v in V has at most one outgoing control dependence edge.

Definition: $ControlPred(v)$ denotes the predicate statement upon which node v is control dependent. More precisely, for a program P if $ControlDep(P) = (V, C, In)$, then

$$ControlPred(v) = \bigcup_{(v,\, x) \in C} \{x\}$$

Example: From the Control Dependence Graph of Figure 3.4, we get $ControlPred(10) = \phi$ whereas $ControlPred(9) = \{5\}$. □

Notice that Control Dependence Graph for our language corresponds exactly to the nesting structure of statements in the program.

3.2.6 Program Dependence Graph

A Program Dependence Graph, *ProgramDep*, of a program, P, is obtained by merging the Data and Control Dependence Graphs of P.[5] Or,

$ProgramDep(P) =$

$\quad let\ DataDep(P) = (V, D),$

$\quad\quad ControlDep(P) = (V, C, In)$

$\quad in\ \ (V, D \cup C)$

Example: Figure 3.5 shows the Program Dependence Graph of the program in Figure 3.1. It is the union of the Data Dependence Graph shown in Figure 3.3 and the Control Dependence Graph shown in Figure 3.4. Dummy nodes $2'$ and $5'$ have been omitted from the Program Dependence Graph as they do not have any data or control dependence edges associated with them; their only purpose was in the syntax directed construction of the program flow-graph. □

[5] In other applications like vectorizing compilers, a Data Dependence Graph may include other types of dependence edges besides data and control dependence, e.g., anti-dependence, output-dependence etc., but for the purposes of program slicing, the former two suffice.

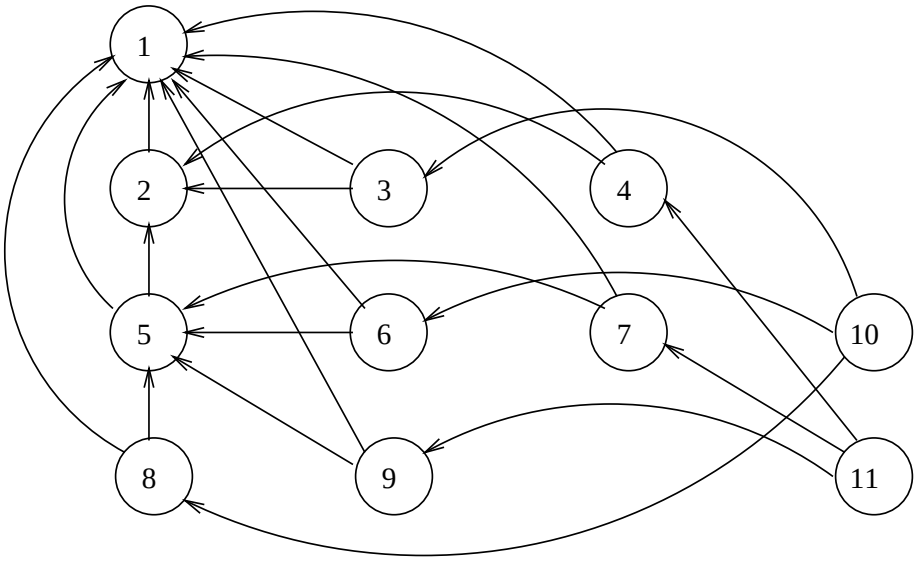

Figure 3.5 Program Dependence Graph of the Program in Figure 3.1

3.2.7 Reachable Nodes

Given a graph $\mathcal{G} = (V, A)$ and a vertex v in V, where V is the set of vertices and A is the set of edges in $\mathcal{G}$, $ReachableNodes(v, \mathcal{G})$ is the set of all those vertices that can be reached from v by following one or more edges in $\mathcal{G}$. Equivalently,

$ReachableNodes(v, \mathcal{G}) =$

$\quad let\ \mathcal{G} = (V, A)$

$\quad in\ \{v\} \cup \bigcup_{(v,x)\in A} ReachableNodes(x, (V, A-\{(v,x)\}))$

Example: Consider the node 10 in the Program Dependence Graph of Figure 3.5. Traversing the graph starting at node 10 and finding all nodes that can be reached from there, we get *ReachableNodes* set for node 10 to be $\{1, 2, 3, 5, 6, 8, 10\}$. □

3.3 Static Slicing

Given a program P, a simple statement or a predicate expression in P (or, equivalently, a node n in the flow-graph of P), and a variable var, the static slice of P with respect to variable var at node n is a subset[6] P' of P such that, for any input, whenever execution reaches node n in P', variable var will have the same value as it has when execution reaches node n in P. Of course, this means that P' should be such that, for any execution, node n is reached exactly the same number of times in P as well as P'.

Let $\mathcal{F} = (V, A, En, Ex)$ be the flow-graph of P. The static slice P' of P can be constructed by finding a subset of V consisting only of those nodes whose execution could possibly *affect* the value of variable var at node n. We call this subset $StaticSlice(P, var, n)$. If var is the only variable used at node n (e.g. if the node n corresponds to **write**(var) statement), then such a subset is easily obtained by traversing the Program Dependence Graph starting at node n and collecting all

[6] A program may also be viewed as an *ordered* set of statements where each statement is uniquely identified by its location in the program, e.g., using line numbers. In this sense, the set operations union, intersection, subset, etc., may also be applied over programs.

nodes reachable from there. If there are other variables besides *var* used at node n, then we need to select only those outgoing edges from node n that lead to nodes defining *var*, and then traverse the Program Dependence Graph from there on. If *var* is not used at node n, then we first need to find all reaching definitions of *var* at node n using the flow-graph of the program, and start traversing the Program Dependence Graph from those nodes. $StaticSlice(P,\ var,\ n)$ can be precisely defined as follows:

$$
\begin{aligned}
&StaticSlice(P,\ var,\ n) = \\
&\quad let\ \mathcal{F} = Flow(P), \\
&\qquad \mathcal{D} = ProgramDep(P) \\
&\quad in\ \bigcup_{x \in StaticReachingDefns(var,\ n,\ \mathcal{F})} ReachableNodes(x,\ \mathcal{D})
\end{aligned}
$$

Example: Consider the program in Figure 3.1. Suppose we wish to find the Static Slice with respect to variable Y at statement 10. We first find the set of reaching definitions of Y at node 10 using the flow-graph of the program shown in Figure 3.2. This set consists of the three nodes {3, 6, 8}. Now we find the set of all reachable nodes from these three nodes in the Program Dependence Graph of the program shown in Figure 3.5. This set, which consists of nodes 1, 2, 3, 5, 6 and 8, gives us the desired slice. Figure 3.6 shows the slice; nodes belonging to the slice are shown in bold. □

3.4 Dynamic Slicing

In the previous example, the static slice for the program in Figure 3.1 with respect to variable Y at statement 10 contains all three assignment statements, namely, 3, 6 and 8, that assign a value to Y. But we know that for this program for any testcase only one of these statements may be executed. For example, for the testcase when X is -1, only statement 3 is executed. In this case, if the value of Y is found to be wrong at the **write** statement 10, either an error in function f_1 at statement 3 or an error in the **if** predicate at statement 2 is responsible for the error. The purpose of

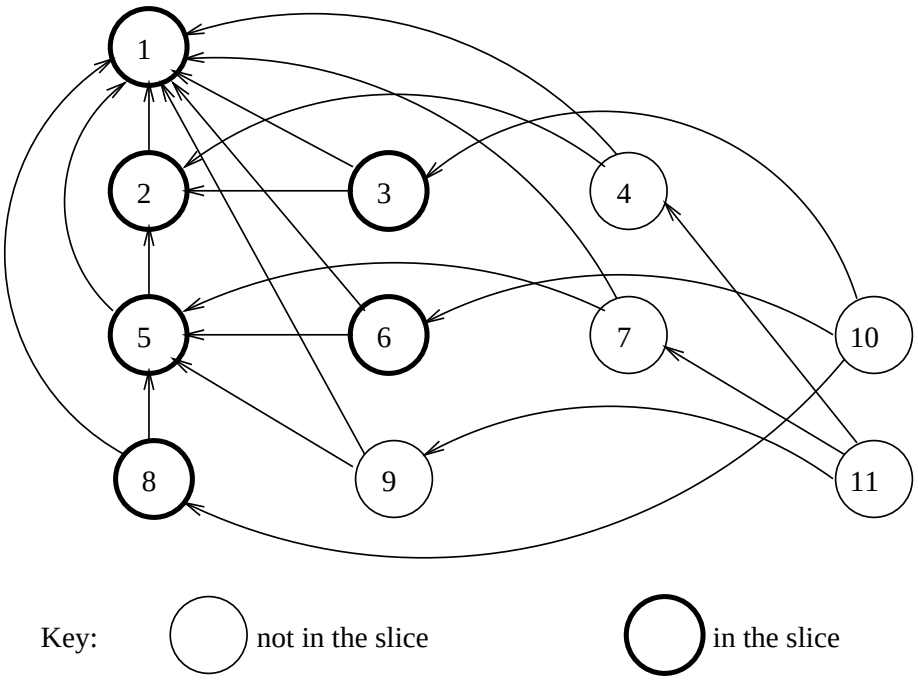

Figure 3.6 Static Slice for Variable Y at Statement 10 for the Program in Figure 3.1

dynamic slicing is to find the subset of program statements whose execution "really" affected the value of the given variable, as observed at the given execution point, for the *given* testcase. So, the dynamic slice for the testcase with X = -1, for variable Y at statement 10, contains statements 1, 2, 3, and 10, as opposed to the static slice which contains statements 1, 2, 3, 5, 6, 8, and 10. Clearly, while debugging the program, if the above error is observed, the dynamic slice would help localize the bug much more quickly than the static slice. This is because the dynamic slice will include only statements that did in fact influence the value of the variable in question for the current testcase and not all statements that "could have" affected its value for *any* testcase

In the next few sections, we describe some approaches to compute successively more refined dynamic slices. But first, for the purposes of defining our algorithms precisely, we need to formalize the notion of execution history.

3.4.1 Execution History

Let $\mathcal{F}$ be the flow-graph of program P. Let *test* be a testcase consisting of a specific set of input-values read by the program. We denote the execution history of the program P for *test* by a sequence *hist* = $<v_1, v_2, \ldots, v_n>$ of vertices in $\mathcal{F}$ appended in the order in which they are visited during the program execution. The execution history at any instance denotes the partial program execution till that instance.

Example: Consider the program in Figure 3.1. For the testcase X = -1, we get the execution history <1, 2, 3, 4, 10, 11>. Also, consider the program with a loop in Figure 3.7 (we will refer to this program again in the next section). Symbols f_1 and f_2 in statements 6 and 7 respectively are some unspecified functions not relevant for the current discussion. For the testcase N = 2, we get the execution history <1, 2, 3, 4, 5^1, 6^1, 7^1, 8^1, 5^2, 6^2, 7^2, 8^2, 5^3, 9>. Note that we use superscripts 1, 2, etc. to distinguish between multiple occurrences of the same statement in the execution history. □

```
        begin
S1:         read(N);
S2:         Z := 0;
S3:         Y := 0;
S4:         I := 1;
S5:         while (I <= N)
            do
S6:             Z := f1(Z, Y);
S7:             Y := f2(Y);
S8:             I := I + 1;
            end_while;
S9:         write(Z);
        end.
```

Figure 3.7 Example Program 2

Definition: $Last(hist)$ denotes the last node in *hist*, and $Prev(hist)$ denotes the subsequence with all but the last node in *hist*. That is,

$Last(<v_1, \ldots, v_{n-1}, v_n>) = v_n$

$Prev(<v_1, \ldots, v_{n-1}, v_n>) = <v_1, \ldots, v_{n-1}>$

We use the notation $<Prev(hist) \mid Last(hist)>$ to denote the two parts of *hist*. Also, $<>$ denotes the empty sequence.

Definition: $LastOccur(v, hist)$ denotes the last occurrence of the node v in *hist*. Or,

$LastOccur(v, <>) = \phi$

$LastOccur(v, <prevhist \mid lastnode>) =$

if *lastnode* an occurrence of v

then $\{lastnode\}$

else $LastOccur(v, prevhist)$

Example: For the execution history of program in Figure 3.1 for testcase X = -1, we have $LastOccur(9, <1, 2, 3, 4, 10, 11>) = \phi$. For the program in Figure 3.7 and testcase N = 2, we have $LastOccur(6, <1, 2, 3, 4, 5^1, 6^1, 7^1, 8^1, 5^2, 6^2, 7^2, 8^2, 5^3, 9>) = \{6^2\}$. □

Definition: $DynamicReachingDefn(var, hist)$ denotes the last occurrence of the node that assigns a value to *var* in the sequence *hist*. Or,

$DynamicReachingDefn(var, <>) = \phi$

$DynamicReachingDefn(var, <prevhist \mid lastnode>) =$

if $var \in def(lastnode)$

then $\{lastnode\}$

else $DynamicReachingDefn(var, prevhist)$

Example: For the execution history of program in Figure 3.1 for testcase X = -1, we have $DynamicReachingDefn(Y, <1, 2, 3, 4, 10, 11>) = \{3\}$. For the program in

Figure 3.7 and testcase N = 2, we have *DynamicReachingDefn*(Z, <1, 2, 3, 4, 5^1, 6^1, 7^1, 8^1, 5^2, 6^2, 7^2, 8^2, 5^3, 9>) = $\{6^2\}$. □

Note that both *LastOccur* and *DynamicReachingDefn* result in either the empty set, implying no occurrence of the desired node, or in a singleton consisting of the unique node desired.

Given an execution history *hist* of a program P on a testcase *test*, and a variable *var*, the dynamic slice of P with respect to *hist* and *var* is the set of all statements in *hist* whose execution had some *effect* on the value of *var* as observed at the end of the execution (we shall give a more precise definition of a dynamic slice in Section 3.4.4.1). Note that unlike static slicing where a slice is defined with respect to a given location in the program, we define dynamic slicing with respect to the end of an execution history. If a dynamic slice with respect to some intermediate point in the execution is desired, then we simply need to consider the partial execution history up to that point.

3.4.2 Dynamic Slicing: Approach 1

For the program in Figure 3.1, we saw above that the static slice with respect to variable Y at statement 10 contains all three assignment statements—3, 6, and 8. But clearly, for any given testcase, only one of these statements is executed. If we marked the nodes in the Program Dependence Graph that get executed for the current testcase, and traverse only the marked nodes in the graph, the slice obtained will not contain nodes that were not executed for the current testcase. In other words, we first take a "projection" of the Program Dependence Graph with respect to only those nodes that are reached during the program execution for the current testcase, and then use the algorithm of Section 3.3 on the projected Dependence Graph to find the desired slice:

$$DynamicSlice1(P, hist, var) = ReachableNodes(DynamicReachingDefn(var, hist), Project(ProgramDep(P), Nodes(hist)))$$

$Project(\mathcal{D}, V') =$

$\quad let\ \mathcal{D} = (V, A)$

$\quad\quad A' = \bigcup_{(x, y) \in A}\ (if\ (x \in V \cap V')\ and\ (y \in V \cap V')\ then\ \{(x, y)\}\ else\ \phi)$

$\quad in\ \ (V \cap V', A')$

$Nodes(<>) = \phi$

$Nodes(<prevhist \mid node>) = \{node\} \cup Nodes(prevhist)$

Example: For the program in Figure 3.1, for testcase X = −1, we have the execution history <1, 2, 3, 4, 10, 11>. Also we have *DynamicReachingDefn*(Y, <1, 2, 3, 4, 10, 11>) = 3. Traversing only the nodes that occur in the execution history, starting at node 3 in the Program Dependence Graph in Figure 3.5, we get the dynamic slice for Y at the end of the execution to be {1, 2, 3}. Figure 3.8 depicts this: All nodes in the graph are drawn dotted in the beginning. As statements are executed, corresponding nodes in the graph are made solid. Then the graph is traversed only for solid nodes, beginning at node 3, the last definition of Y in the execution history. All nodes reached during the traversal are made bold. The set of all bold nodes, {1, 2, 3} in this case, gives the desired slice. □

Unfortunately, *DynamicSlice1* does not always find accurate dynamic slices. It may sometimes include extra statements in the slice that did not affect the value of the variable in question for the given execution history. To see why, consider the program in Figure 3.7 and the testcase N = 1, which yields the execution history <1, 2, 3, 4, 5^1, 6, 7, 8, 5^2, 9>. Figure 3.9 shows the the result of applying the algorithm *DynamicSlice1* on the Program Dependence Graph of this program with respect to Z at the end of the execution. Looking at the execution history <1, 2, 3, 4, 5^1, 6, 7, 8, 5^2, 9> carefully, we find that statement 7 assigns a value to Y which is never used later, for none of the statements that appear after 7 in the execution history, namely, 8, 5, and 9, uses variable Y. So statement 7 should not be in the dynamic slice for the current testcase. *DynamicSlice1* includes it in the slice because statement 7 is executed under the current testcase, and statement 9 depends on statement 6 which

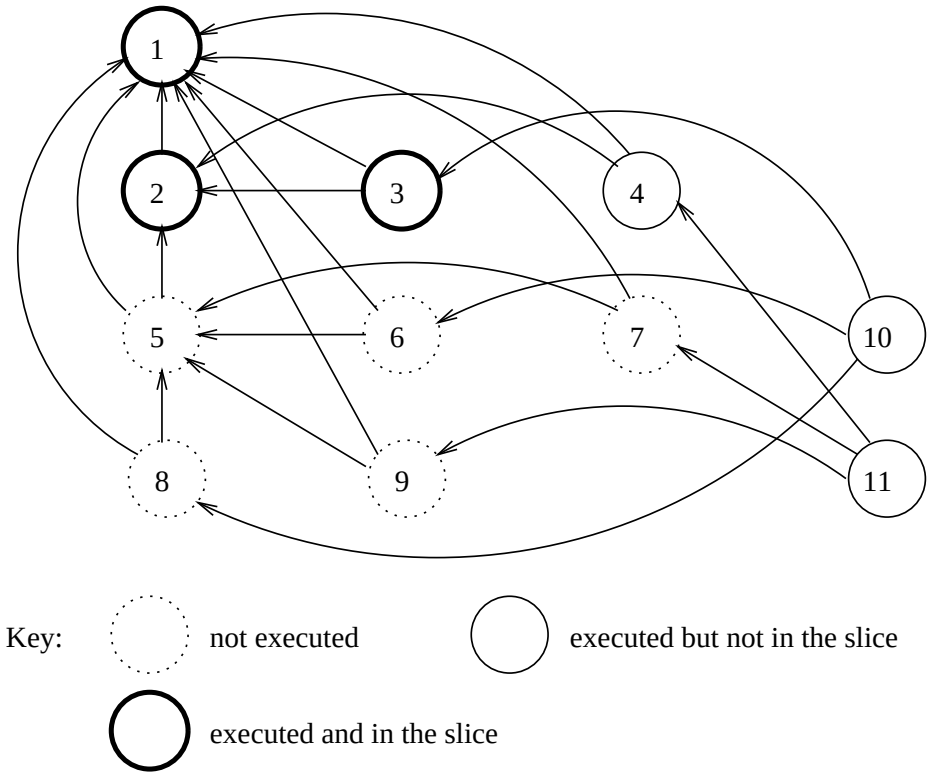

Figure 3.8 *DynamicSlice1* for the program in Figure 3.1, for testcase X = −1, for variable Y, at the end of execution

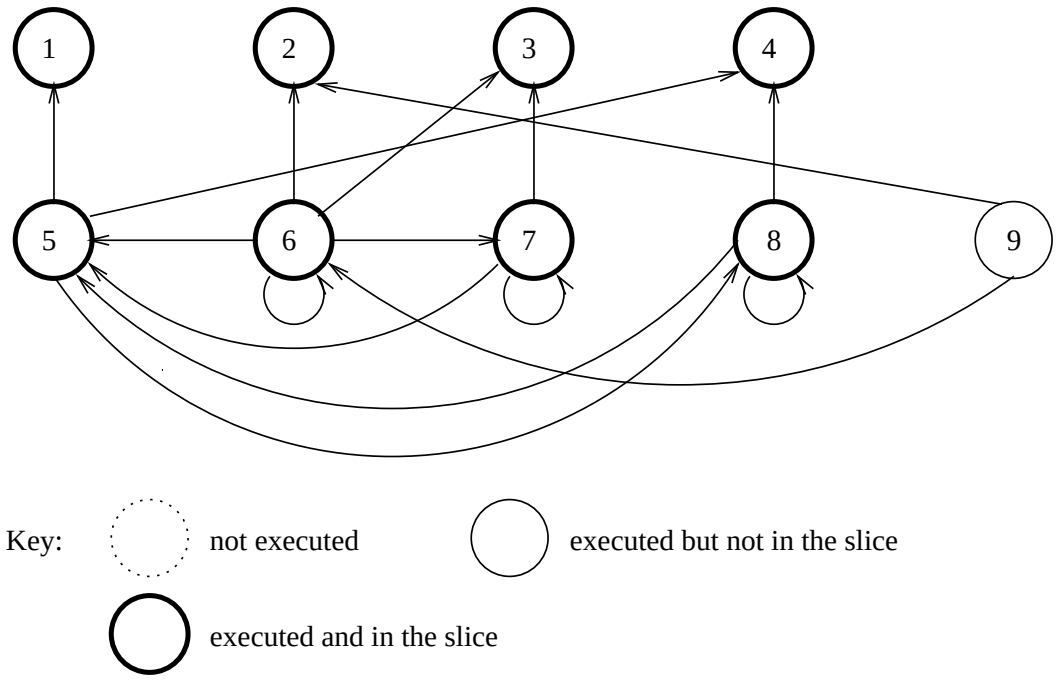

Figure 3.9 *DynamicSlice1* for the program in Figure 3.7, testcase N = 1, for variable Z, at the end of execution

has a data dependency edge to statement 7 in the Program Dependence Graph. In the next section we present a refinement to the above approach that avoids this problem.

3.4.3 Dynamic Slicing: Approach 2

The problem with the *DynamicSlice1* approach discussed above lies in the fact that a statement may have multiple reaching definitions of the same variable in the program flow-graph, and hence it may have multiple outgoing data dependence edges for the same variable in the Program Dependence Graph. Selection of such a node in the dynamic slice, according to *DynamicSlice1*, implies that all nodes to which it has outgoing data-dependence edges also be selected if the nodes have been executed, even though the corresponding data-definitions may not have reached the current node. In the example above, Statement 6 has multiple reaching definitions of the same variables: two definitions of variable Y from statements 3 and 7, and two of

variable Z from statements 2 and 6 itself. So it has two outgoing data dependency edges for each of variables Y and Z, to statements 3 and 7, and 2 and 6 respectively (besides a control dependence edge to node 5). For the testcase N = 1, each of these four statements is executed, so inclusion of statement 6 in the slice leads to the inclusion of other three statements 3, 7, and 2 as well, even though two of the data dependencies of statement 6—on statement 7 for variable Y and on itself for variable Z—are never activated for this testcase (N = 1), because the loop is iterated only once.

In a flow-graph a statement may have multiple reaching definitions of a variable because there could be multiple execution paths leading up to that statement, and each of these paths may have different statements assigning a value to the same variable. But for any single path, there can be at most one reaching definition of any variable at any statement. And as we are interested in examining dependencies for a single execution path—the execution history under the given testcase—inclusion of a statement in the dynamic slice should lead to inclusion of only those statements that actually defined values used by it under the current testcase. This suggests that we should mark the edges of the Program Dependence Graph as the corresponding dependencies occur during the program execution, and traverse the graph only along the marked edges. Or, more precisely:

$$DynamicSlice2(P,\ hist,\ var) =$$

$$\quad let\ ProgramDep(P) = (V,\ A)$$

$$\quad in\ \ ReachableNodes(DynamicReachingDefn(var,\ hist),\ (V,\ Edges(hist)))$$

$$Edges(<>) = \phi$$

$$Edges(<prevhist \mid next>) =$$

$$\quad let\ D = \bigcup_{\substack{var \in use(next) \\ x \in DynamicReachingDefn(var,\ prevhist)}} \{(next,\ x)\}$$

$$\quad\quad C = \bigcup_{\substack{x \in ControlPred(next) \\ y \in LastOccur(x,\ prevhist)}} \{(next,\ y)\}$$

$$\quad in\ D \cup C \cup Edges(prevhist)$$

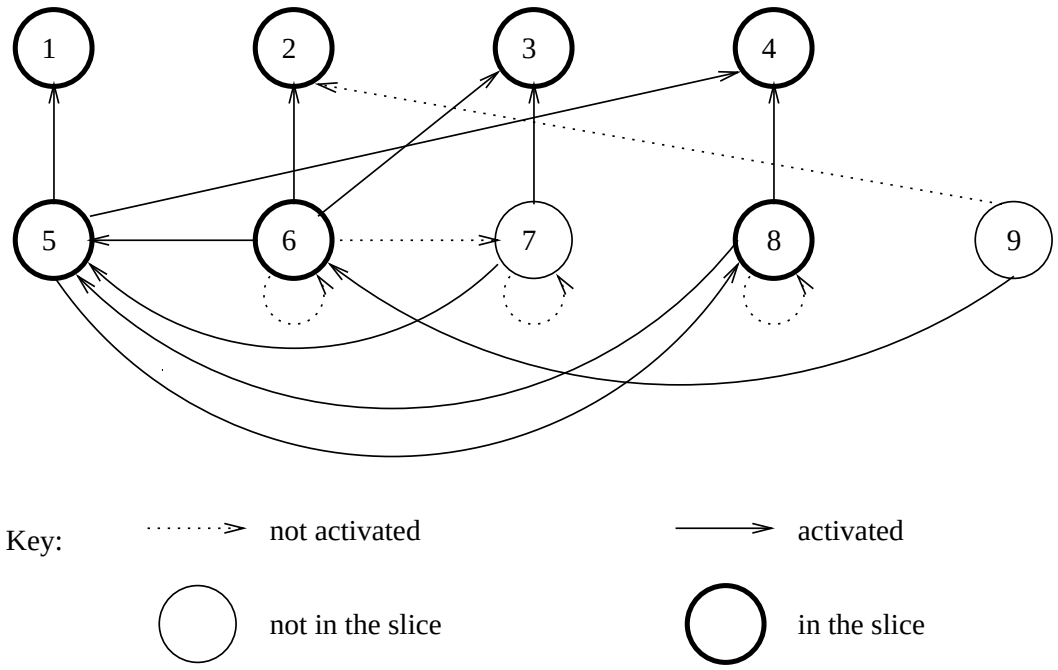

Figure 3.10 *DynamicSlice2* for the program in Figure 3.7, testcase N = 1, for variable Z, at the end of execution

Example: Consider again executing the program of Figure 3.7 for testcase N = 1. Applying *DynamicSlice2* on its execution history <1, 2, 3, 4, 5^1, 6, 7, 8, 5^2, 9> for variable Z yields the dynamic slice {1, 2, 3, 4, 5, 6, 8} which does not include statement 7. This is depicted in Figure 3.10: All edges are drawn as dotted lines in the beginning. As statements are executed, edges corresponding to the new dependencies that occur are drawn as solid lines. Then the graph is traversed only along solid edges and the nodes reached are made bold. The set of all bold nodes at the end gives the desired slice. Note that statement 7 that was included by *DynamicSlice1* in the slice in not included by *DynamicSlice2*. □

If programs had no loops then the above approach would always yield accurate dynamic slices. But in the presence of loops, it may sometimes include more statements than necessary in the slice. Consider the program in Figure 3.11 and the testcase when N = 2, and the two values of X read are −4 and 3. Then, for first time through

the loop statement 6, the **then** part of the **if** statement, is executed and the second time through the loop statement 7, the **else** part is executed. Now suppose the execution has reached just past statement 9 the second time through the loop and the second value of Z printed is found to be wrong. The execution history thus far is <1, 2, 3^1, 4^1, 5^1, 6, 8^1, 9^1, 10^1, 3^2, 4^2, 5^2, 7, 8^2, 9^2>. If we used *DynamicSlice2* to find the slice for variable Z for this execution history, we would have both statements 6 and 7 included in the slice, even though the value of Z, in this case, is only dependent on statement 7. Figure 3.12 shows a segment of the Program Dependence Graph (only statements 4, 6, 7, 8, and 9) along with the effect of executing *DynamicSlice2*. Data dependence edge from 8 to 6 is marked during the first iteration, and that from 8 to 7 is marked during the second iteration. As both these edges are marked, inclusion of statement 8 leads to inclusion of both statements 6 and 7, even though the value of Z observed at the end of the second iteration is only affected by statement 7.

As we mentioned earlier, *DynamicSlice2* will always find accurate dynamic slices if the program has no loops.[7] This is so because a statement can never appear more than once in any execution history if the program has no loops. Also, for any given occurrence of a statement in the execution history, that occurrence has at most one reaching definition of each variable used by that occurrence. Hence, for each statement occurrence and for each variable used by that occurrence, the corresponding node in the Program Dependence Graph will have at most one outgoing data dependence edge marked. As, in absence of loops, a statement can occur only once in the execution history, other outgoing data dependence edges for the statement, if any, are never marked. So whenever a node is selected in the slice, it leads to selection of only those nodes that affected the current occurrence of the selected node. And as we always start the selection of nodes for inclusion in the dynamic slice by selecting the node that directly affected the value of the given variable at the end of the execution history, we always obtain accurate dynamic slices.

[7]This is not to say that it will always find overlarge slices if the program has loops. Figure 3.10 shows an example where *DynamicSlice2* obtains an accurate dynamic slice for a program with a loop.

```
        begin
S1:         read(N);
S2:         I := 1;
S3:         while (I <= N)
            do
S4:           read(X);
S5:           if (X < 0)
              then
S6:               Y := f1(X);
              else
S7:               Y := f2(X);
              end_if;
S8:           Z := f3(Y);
S9:           WRITE(Z);
S10:          I := I + 1;
            end_while;
        end.
```

Figure 3.11 Example Program 3

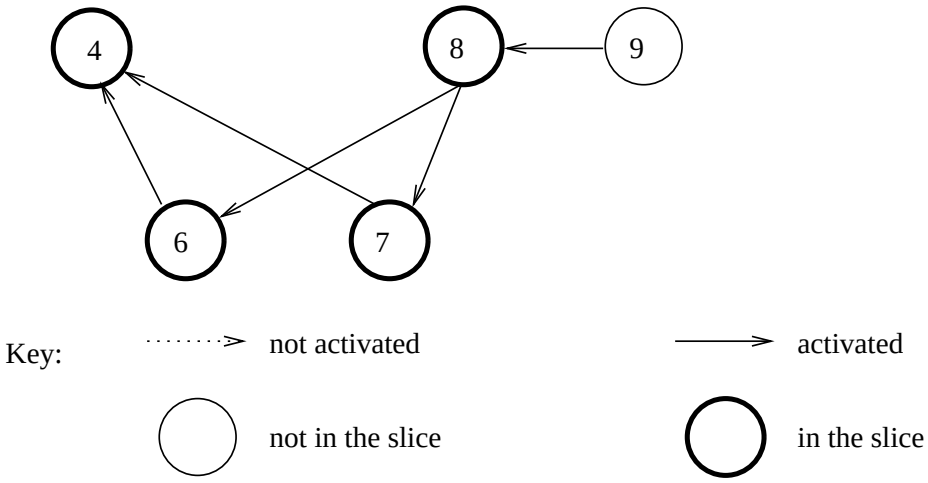

Figure 3.12 Subset of *DynamicSlice2* for the Program in Figure 3.11, testcase (N = 2, X = −4, 3), for Variable Z

It may seem that the difficulty with *DynamicSlice2* discussed above will disappear if, before marking the data-dependence edges for a new occurrence of a statement in the execution history, we first *unmarked* any outgoing dependence edges that are already marked for this statement. This scheme will work for the above example, but unfortunately, it may lead to wrong dynamic slices in other situations. Consider, for example, the program in Figure 3.13. Consider the case when the loop is iterated twice, the first time through statements 7 and 11, and the second time through statement 8 but skipping statement 11. If we obtain the dynamic slice for A at the end of execution, we will have statement 8 in the slice instead of statement 7. This is because when statement 9 is reached second time through the loop, the dependence edge from 9 to 7 (for variable Y) is unmarked and that from 9 to 8 is marked. Then, while finding the slice for A at statement 13, we will include statement 11, which last defined the value of A. As statement 11 used the value of Z defined at statement 9, 9 is also included in the slice. But inclusion of 9 leads to inclusion of 8 instead of 7, because the dependence edge to the latter was unmarked during the second iteration. The value of Z at statement 11, however, depends on value of Y defined by statement 7 during the first iteration, so 7 should be in the slice, not 8.

Thus the above scheme of unmarking previously marked edges with every new occurrence of a statement in the execution history does not work. This scheme is worse than both *DynamicSlice1* and *DynamicSlice2*, as the latter two find supersets of minimal dynamic slices while this may omit statements that belong to the slice.

3.4.4 Dynamic Slicing: Approach 3

DynamicSlice2, as we saw above, may lead to overlarge dynamic slices because a statement may have multiple occurrences in an execution history, and different occurrences of a statement may have different reaching definitions of the variables used at that statement. The Program Dependence Graph does not distinguish between these different occurrences, so inclusion of a statement in the dynamic slice by virtue of one occurrence may some times lead to inclusion of statements on which a different

```
        begin
S1:         read(N);
S2:         A := 0;
S3:         I := 1;
S4:         while (I <= N)
            do
S5:           read(X);
S6:           if (X < 0)
              then
S7:                 Y := f1(X);
              else
S8:                 Y := f2(X);
              end_if;
S9:           Z := f3(Y);
S10           if (Z > 0)
              then
S11:                A := f4(A, Z);
              else
              end_if;
S12:          I := I + 1;
            end_while;
S13:        write(A);
        end.
```

Figure 3.13 Example Program 4

occurrence of that statement is dependent. In other words, different occurrences of the same statement may have different dependencies, and it is possible that one occurrence contributes to the slice and another occurrence does not. Inclusion of one occurrence in the slice should lead to inclusion of only those statements on which this occurrence of the statement is dependent, not those on which some other occurrences may be dependent. This suggests that we should have different nodes for different occurrences of the same statement in the execution history, and each occurrence of a statement should have dependence edges to only those statements (their specific occurrences) on which this particular statement occurrence is dependent. Then every node in the dependence graph will have at most one outgoing edge for each variable used at the statement. We call this new dependence graph the Dynamic Dependence Graph. A program has different dynamic dependence graphs for different execution histories. In the next section we precisely define how a Dynamic Dependence Graph is built.

3.4.4.1 Dynamic Dependence Graph

The Dynamic Dependence Graph, *DynamicDep*, of an execution history *hist* is a two-tuple (V, A), where V is the *multi-set* of flow-graph vertices (i.e., multiple entries of the same element are treated as distinct), and A is the set of edges denoting dynamic data dependencies and control dependencies between vertices. We use the symbol $\uplus$ to denote a disjunctive union of elements that constructs multi-sets (i.e., sets allowing multiple occurrences of the same element). *DynamicDep* is defined as follows:

$$DynamicDep(<>) = (\phi, \phi)$$

$$DynamicDep(<prevhist \mid next>) =$$

$$\quad let\ DynamicDep(prevhist) = (V, A),$$

$$\quad\quad D = \bigcup_{\substack{var \in use(next) \\ x \in DynamicReachingDefn(var,\ prevhist)}} \{(next, x)\},$$

$$\quad\quad C = \bigcup_{\substack{x \in ControlPred(next) \\ y \in LastOccur(x,\ prevhist)}} \{(next, y)\}$$

in $(V \uplus \{next\},\ A \cup D \cup C)$

Example: Consider the program in Figure 3.11, and testcase (N = 3, X = −4, 3, −2), which yields the execution history <1, 2, 3^1, 4^1, 5^1, 6^1, 8^1, 9^1, 10^1, 3^2, 4^2, 5^2, 7^1, 8^2, 9^2, 10^2, 3^3, 4^3, 5^3, 6^2, 8^3, 9^3, 10^3, 3^4>. Figure 3.14 shows the Dynamic Dependence Graph for this execution history. The middle three rows of nodes in the figure correspond to the three iterations of the loop. Notice in particular occurrences of node 8 on these rows. During the first and third iterations, it depends on node 6 which corresponds to dependence of statement 8 for the value of Y assigned by node 6, whereas during the second iteration, it depends on node 7 which corresponds to the dependence of statement 8 for the value of Y assigned by node 7. □

Once we have constructed the Dynamic Dependence Graph for the given execution history, we can easily obtain the dynamic slice for a variable, *var*, by first finding the node corresponding to the last definition of *var* in the execution history, and then finding all nodes in the graph reachable from that node. *DynamicSlice3* can be defined precisely as follows:

DynamicSlice3(*hist*, *var*) =

ReachableNodes(*DynamicReachingDefn*(*var*, *hist*), *DynamicDep*(*hist*))

Example: Consider again the program in Figure 3.11, and testcase (N = 3, X = −4, 3, −2). Figure 3.14 shows the effect of executing *DynamicSlice3* on the Dynamic Dependence Graph for variable Z at the end of the execution. Nodes in bold belong to the slice. Note that statement 6 belongs to the slice whereas statement 7 does not. *DynamicSlice2*, on the other hand, would have included statement 7 as well. □

As the above algorithm accurately captures our notion of dynamic slicing, discussed informally above, we use this algorithm as the precise definition of dynamic slicing:

Definition: The Dynamic Slice of a variable, *var*, with respect to the end of an execution history, *hist*, is defined by:

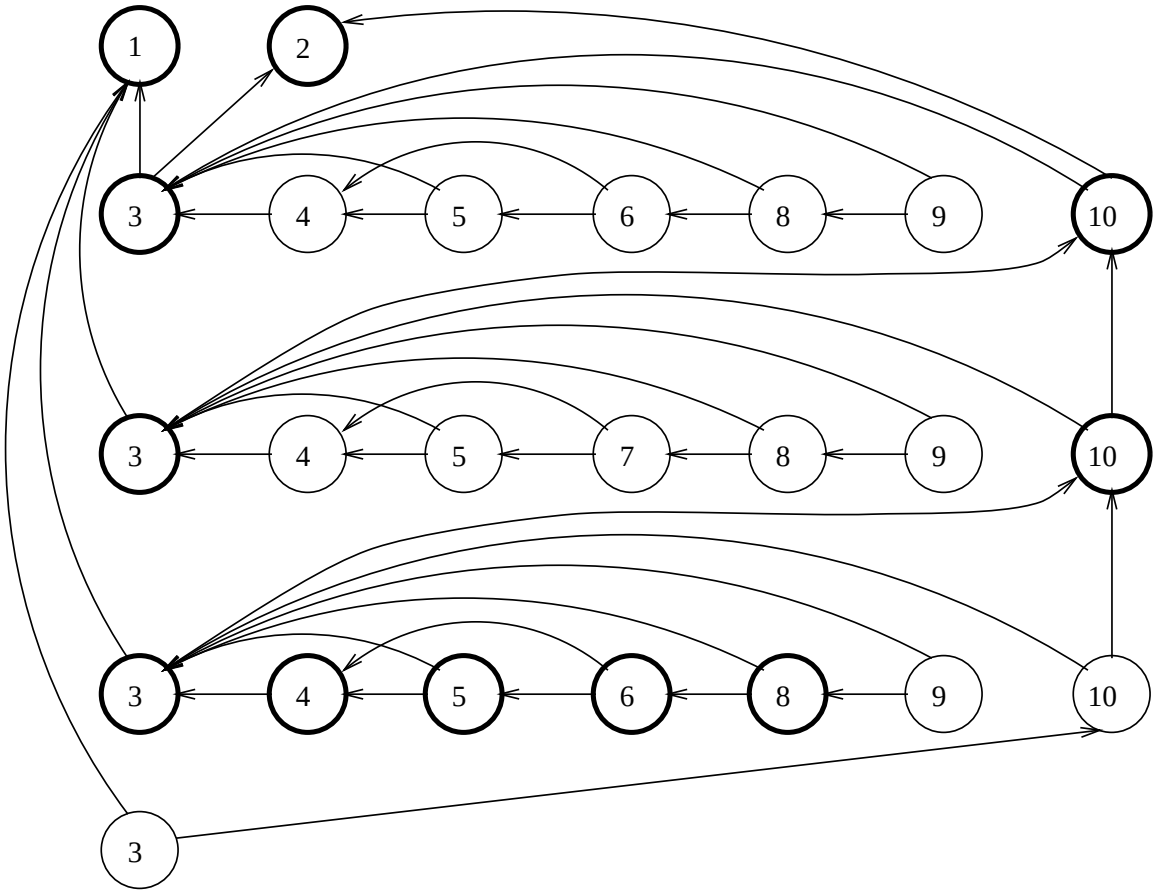

Figure 3.14 Dynamic Dependence Graph for the Program in Figure 3.11 for the testcase (N = 3, X = −4, 3, −2). Nodes in bold give the Dynamic Slice for this testcase with respect to variable Z at the end of execution

$DynamicSlice(hist,\ var) =$

$$ReachableNodes(DynamicReachingDefn(var,\ hist),\ DynamicDep(hist))$$

Contrast this definition of a dynamic slice with that proposed by Korel and Laski [KL88a]. Their definition may yield unnecessarily large dynamic slices. They require 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 other occurrences. The dynamic slice so obtained is executable and produces the same value(s) of the variable in question at the given location as the original program. For our purposes, the usefulness of a dynamic slice lies not in the fact that one can execute it, but in the fact that it isolates only those statements that affected a particular value observed at a particular location. For example, in the program of Figure 3.11 each loop iteration computes a value of Z, and each such computation is totally independent of computation performed during any other iteration. If the value of variable Z at the end of a particular iteration is found to be incorrect, we would like only those statements to be included in the slice that affected the value of Z as observed at the end of that iteration, not during all previous iterations. For example, for the testcase when N = 3, and X = $(-4, 3, -2)$, both statements 6 and 7 will be included in the dynamic slice with respect to Z under their definition, even though statement 7 does not affect the final value of Z in any way. It is interesting to note that our Approach 2 (which may yield an overlarge dynamic slice) would obtain the same dynamic slice as obtained under their definition.

The Dynamic Dependence Graph of an execution history contains a node for every occurrence of a statement (simple statement or predicate expression) in the execution history. This means that we need to save the entire execution history of a testcase to be able to construct the Dynamic Dependence Graph and find dynamic slices with respect to any variables at the end of the execution. In the next section, we discuss an approach to obtain accurate dynamic slices that does not require the entire execution history to be saved.

3.4.5 Dynamic Slicing: Approach 4

The size (total number of nodes and edges) of a Dynamic Dependence Graph as defined in Section 3.4.4.1 is, in general, *unbounded*. This is because the number of nodes in the graph is equal to the number of statements in the execution history, which, in general, may depend on values of run-time inputs. For example, consider the program in Figure 3.7. The number of statements in the execution history, and hence the size of the Dynamic Dependence Graph, of this program for any testcase depends on how many times the **while** loop at statement 5 is iterated, which in turn depends on the value read by variable N at statement 1. On the other hand, we know that every program can have only a finite number of possible dynamic slices because it contains only a finite number of statements, and a slice is a subset of statements. Further, only a subset of these possible dynamic slices arise in any given execution history. This suggests that we ought to be able to restrict the number of nodes in a Dynamic Dependence Graph so its size is not a function of the length of the corresponding execution history. We address below one such mechanism to achieve this.

3.4.5.1 Reduced Dynamic Dependence Graph

In a Reduced Dynamic Dependence Graph, instead of creating a new node for every occurrence of a statement in the execution history we create a new node only if another node with the same transitive dependencies does not already exist. To do this we maintain two tables called *DefnNode* and *PredNode*. *DefnNode* maps a variable name to a node in the Reduced Dynamic Dependence Graph that last assigned a value to that variable. *PredNode* maps a control predicate statement to a node in the Reduced Dynamic Dependence Graph that corresponds to the last occurrence of the predicate in the execution history thus far. We need these tables because we do not save the execution history. Also, we associate a set, *ReachableStmts*, with each node in the graph. This set consists of all statements one or more of whose occurrences can be reached from the given node. We maintain reachable *statements*

and not reachable *nodes* for each node because it is the statements that eventually define the dynamic slice, not their individual occurrences. Every time a statement, S_i, gets executed, we determine the set of nodes, D, that last assigned values to the variables used by S_i, and the last occurrence, C, of the control predicate node of the statement. The *ReachableStmts* set for this occurrence of S_i is obtained by taking the union of *ReachableStmts* of all nodes in $D \cup C$. If a node, n, associated with S_i already exists with the same *ReachableStmts* set, we associate the new occurrence of S_i with node n. Otherwise we create a new node with outgoing edges to nodes in $D \cup C$. The *DefnNode* table entry for the variable assigned at S_i, if any, is also updated to point to this node. Similarly, if the current statement is a control predicate, the corresponding entry in *PredNode* is updated to point to this node.

Precisely, a Reduced Dynamic Dependence Graph is a five-tuple (V, A, *ReachableStmts*, *DefnNode*, *PredNode*), where V and A are the sets of nodes and edges respectively, and *ReachableStmts*, *DefnNode* and *PredNode* are maps defined above.

$$
\begin{aligned}
&\mathit{ReducedDynamicDep1}(<>) = (\phi,\ \phi,\ \phi,\ \phi,\ \phi)\\
&\mathit{ReducedDynamicDep1}(<\mathit{prevhist} \mid \mathit{next}>) =\\
&\quad \mathit{let}\ \mathit{ReducedDynamicDep1}(\mathit{prevhist}) = (V,\ A,\ \mathit{ReachableStmts},\ \mathit{DefnNode},\ \mathit{PredNode}),\\
&\qquad D = \bigcup_{var \in use(next)} \mathit{DefnNode}(var),\\
&\qquad C = \bigcup_{x \in ControlPred(next)} \mathit{PredNode}(x),\\
&\qquad R = \{next\} \cup \bigcup_{x \in D \cup C} \mathit{ReachableStmts}(x),\\
&\qquad N = \mathit{SimilarNode1}(\mathit{next},\ R,\ V,\ \mathit{ReachableStmts}),\\
&\quad \mathit{in}\ \mathit{if}\ N{=}\phi\\
&\qquad \mathit{then}\ \mathit{AddNode1}(\mathit{next},\ D \cup C,\ R,\ (V,\ A,\ \mathit{ReachableStmts},\ \mathit{DefnNode},\ \mathit{PredNode}))\\
&\qquad \mathit{else}\ \mathit{let}\ \mathit{DefnNode}'(var) = \mathit{if}\ var \in def(next)\ \mathit{then}\ N\ \mathit{else}\ \mathit{DefnNode}(var),\\
&\qquad\qquad \mathit{PredNode}'(\mathit{Stmt}) = \mathit{if}\ (\mathit{next}\ \text{an occurrence of}\ \mathit{Stmt})\ \mathit{then}\ N\\
&\qquad\qquad\qquad\qquad \mathit{else}\ \mathit{PredNode}(\mathit{Stmt}))\\
&\qquad\quad \mathit{in}\ (V,\ A,\ \mathit{ReachableStmts},\ \mathit{DefnNode}',\ \mathit{PredNode}')
\end{aligned}
$$

$SimilarNode1(S,\ R,\ \phi,\ ReachableStmts) = \phi$

$SimilarNode1(S,\ R,\ V,\ ReachableStmts) =$

$\quad let\ V = \{v\} \cup V'$

$\quad in\ if\ (v$ an occurrence of $S)\ and\ (ReachableStmts(v) = R)$

$\quad\quad then\ \{v\}$

$\quad\quad else\ SimilarNode1(S,\ R,\ V',\ ReachableStmts)$

$AddNode1(v,\ D,\ R,\ \mathcal{G}) =$

$\quad let\ \mathcal{G} = (V,\ A,\ ReachableStmts,\ DefnNode,\ PredNode),$

$\quad\quad A' = \bigcup_{x \in D} \{(v,\ x)\},$

$\quad\quad ReachableStmts'(n) = (if\ n{=}v\ then\ R\ else\ ReachableStmts(n)),$

$\quad\quad DefnNode'(var) = (if\ var{\in}def(v)\ then\ \{v\}\ else\ DefnNode(var)),$

$\quad\quad PredNode'(S) = (if\ (v$ an occurrence of $S)\ then\ \{v\}\ else\ PredNode(S))$

$\quad in\ (V \uplus \{v\},\ A \cup A',\ ReachableStmts',\ DefnNode',\ PredNode')$

Example: Consider again the program in Figure 3.11, and testcase (N = 3, X = -4, 3, -2), which yields the execution history <1, 2, 3^1, 4^1, 5^1, 6^1, 8^1, 9^1, 10^1, 3^2, 4^2, 5^2, 7^1, 8^2, 9^2, 10^2, 3^3, 4^3, 5^3, 6^2, 8^3, 9^3, 10^3, 3^4>. Figure 3.15 shows the Reduced Dynamic Dependence Graph for this execution history. Every node in the graph is annotated with the set of all reachable statements from that node. Note that there is only one occurrence of node 10 in this graph, as opposed to three occurrences in the Dynamic Dependence Graph for the same program and the same testcase shown in Figure 3.14. This is because all three occurrences of node 10 in Figure 3.14 have the same set R. Hence only one node 10 is created in the Reduced Dynamic Dependence Graph. □

Once we have the Reduced Dynamic Dependence Graph for the given execution history, to obtain the dynamic slice for any variable *var* we first find the entry in the table *DefnNode* for *var*. The *ReachableStmts* set associated with that entry gives the desired dynamic slice. So we don't even have to traverse the Reduced

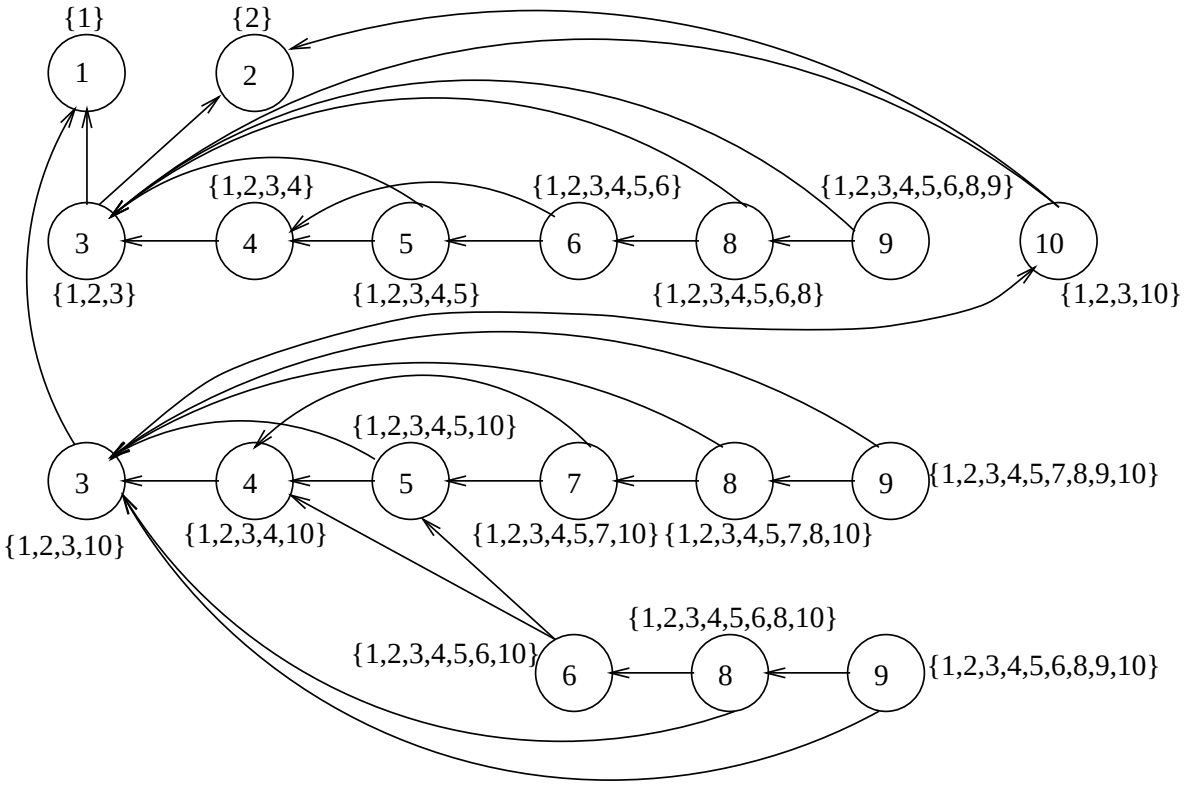

Figure 3.15 Reduced Dynamic Dependence Graph for the Program in Figure 3.11 for the testcase (N = 3, X = −4, 3, −2). Each node is annotated with *ReachableStmts*, the set of all statements reachable from that node

Dynamic Dependence Graph to find the slice. *DynamicSlice4* can be defined precisely as follows:

DynamicSlice4(*hist*, *var*) =

let *ReducedDynamicDep1*(*hist*) = (*V*, *A*, *ReachableStmts*, *DefnNode*, *PredNode*)

in *ReachableStmts*(*DefnNode*(*var*))

Example: Consider again the program in Figure 3.11, and testcase (N = 3, X = −4, 3, −2). Figure 3.15 shows the Reduced Dynamic Dependence Graph of the execution history for this testcase. The dynamic slice for variable Z at the end of execution is given by the *ReachableStmts* set, {1, 2, 3, 4, 5, 6, 8, 10}, associated with node 8 in the last row, as that was the last node to define value of Z. □

Note that under this approach we do not really need to construct the graph edges; the *ReachableStmts* sets associated with the nodes directly give the corresponding dynamic slices so we never need to traverse the edges. Also, as we always require the last occurrence of the node that defined a variable, we only need to keep one node for each statement. This means instead of looking for a statement occurrence node with the similar *ReachableStmts* set we can simply overwrite the *ReachableStmts* set of the unique node for the corresponding statement. This approach will work well if we always needed dynamic slices with respect to the end of the current execution history. But this simplification will also make obtaining dynamic slices with respect to prefixes of the current execution history much more difficult.

Constructing the Reduced Dynamic Dependence Graph requires that at each step in the execution history we determine the set of reachable statements associated with the new statement occurrence. This means a union of *ReachableStmts* sets of all immediate descendent nodes of the new node is performed at each step. In the next section we present a variation of the graph reduction mechanism discussed above where instead of performing the expensive set union operation at each step, we need to perform it only once for each node—at the time of its creation.

3.4.6 Efficient Reduction of Dynamic Dependence Graph

In this section we describe a new scheme to reduce the Dynamic Dependence Graph that is more efficient than the one described above. Under this scheme for every new occurrence of a statement in the execution history we only need to find the set of nodes on which the new occurrence is *immediately* dependent, and check if another node with same set of direct dependencies already exists. If such a node is found, then the new node is not created, but the *DefnNode* and *PredNode* tables are updated appropriately. If there were no circular dependencies in the dependence graph then this scheme of only considering direct dependencies would work fine. But in the presence of loops, the program may have circular dependencies, in which case the graph reduction described above will not occur; for every iteration of the loop we would have to create new node occurrences. We can avoid this problem, if whenever we need to create a new node, say for statement S_i, we first determine if any of its immediate dependents, say node v, already has a dependency on a previous occurrence of S_i and if the other immediate dependents of S_i are also reachable from v. This is easily done by checking if the *ReachableStmts* set to be associated with the new occurrence is a subset of the *ReachableStmts* set associated with v. If so we can merge the new occurrence of S_i with v. After this merge, during subsequent iterations of the loop the search for a node for S_i with the same immediate dependents will always succeed. So under this scheme, the expensive set union operations have to be performed only during initial iterations of a loop when new inter-statement dependencies are being activated, but not during subsequent iterations if the same old dependencies are being repeated. In the previous scheme, on the other hand, every iteration required computation of set unions. The new reduction scheme is precisely defined as follows.

$ReducedDynamicDep2(<>) = (\phi,\ \phi,\ \phi,\ \phi,\ \phi)$

$ReducedDynamicDep2(<prevhist \mid next>) =$

$\quad let\ ReducedDynamicDep2(prevhist) = (V,\ A,\ ReachableStmts,\ DefnNode,\ PredNode),$

$D = \bigcup_{var \in use(next)} DefnNode(var)$,

$C = \bigcup_{x \in ControlPred(next)} PredNode(x)$,

$N = SimilarNode2(next, D \cup C, V)$,

$DefnNode'(var) = (if\ var \in def(next)\ then\ N\ else\ DefnNode(var))$,

$PredNode'(S) = (if\ (next$ an occurrence of $S)\ then\ N\ else\ PredNode(S))$

in if $N = \phi$

then $AddNode2(next, D \cup C, (V, A, ReachableStmts, DefnNode, PredNode))$

else $(V, A, ReachableStmts, DefnNode', PredNode')$

$SimilarNode2(S, D, \phi) = \phi$

$SimilarNode2(S, D, V) =$

let $V = \{v\} \cup V'$

in if (v an occurrence of S) *and* $(SimilarDeps(v, D, A))$

then $\{v\}$

else $SimilarNode2(S, D, V')$

$SimilarDeps(v, \phi, A) = true$

$SimilarDeps(v, D, A) =$

let $D = \{x\} \cup D'$

in if $(x = v)$ *or* $((v, x) \in A)$

then $SimilarDeps(v, D', A)$

else false

$AddNode2(v, D, \mathcal{G}) =$

let $\mathcal{G} = (V, A, ReachableStmts, DefnNode, PredNode)$,

$R = \{v\} \cup \bigcup_{x \in D} ReachableStmts(x)$,

$A' = \bigcup_{x \in D} \{(v, x)\}$,

$ReachableStmts'(x) = (if\ x = v\ then\ R\ else\ ReachableStmts(x))$,

$DefnNode'(var) = (if\ var \in def(v)\ then\ \{v\}\ else\ DefnNode(var))$,

$PredNode'(S) = (if\ (v \text{ an occurrence of } S)\ then\ \{v\}\ else\ PredNode(S))$

$in\ if\ CyclePossible(D,\ R,\ ReachableStmts) = \{n\}$

$then\ MergeNode(v,\ n,\ D,\ \mathcal{G})$

$else\ (V \uplus \{v\},\ A \cup A',\ ReachableStmts',\ DefnNode',\ PredNode')$

$CyclePossible(\phi,\ R,\ ReachableStmts) = \phi$

$CyclePossible(D,\ R,\ ReachableStmts) =$

$let\ D = \{x\} \cup D'$

$in\ if\ (R \subseteq ReachableStmts(x))$

$then\ \{x\}$

$else\ CyclePossible(D',\ R,\ ReachableStmts)$

$MergeNode(v_1,\ v_2,\ D,\ \mathcal{G}) =$

$let\ \mathcal{G} = (V,\ A,\ ReachableStmts,\ DefnNode,\ PredNode),$

$A' = \bigcup_{x \in D} \{(v_2,\ x)\},$

$DefnNode'(var) = (if\ (var \in def(v_1)\ then\ \{v_2\}\ else\ DefnNode(var)),$

$PredNode'(S) = (if\ (v_1 \text{ an occurrence of } S)\ then\ \{v_2\}\ else\ PredNode(S))$

$in\ (V,\ A \cup A',\ ReachableStmts,\ DefnNode',\ PredNode')$

Example: Consider again the program in Figure 3.11, and testcase (N = 3, X = -4, 3, -2), which yields the execution history <1, 2, 3^1, 4^1, 5^1, 6^1, 8^1, 9^1, 10^1, 3^2, 4^2, 5^2, 7^1, 8^2, 9^2, 10^2, 3^3, 4^3, 5^3, 6^2, 8^3, 9^3, 10^3, 3^4>. Figure 3.16 shows the Reduced Dynamic Dependence Graph obtained by applying *ReducedDynamicDep2* to this execution history. Note that the second occurrence of node 3 is merged with its immediate dependent node 10 because the *ReachableStmts* set, {1, 2, 3, 10}, of the former was a subset of that of the latter. The third occurrence of node 3 in the execution history has node 1 and node 10 as immediate dependents. As these immediate dependencies are also contained in the merged node (10,3), the third occurrence of node 3 is also associated with this node. The dynamic slice for variable Z at the end of execution

is given by *ReachableStmts* set, {1, 2, 3, 4, 5, 6, 8, 10}, associated with node 8 in the last row, as that was the last node to define value of Z. □

3.5 Summary

The conventional notion of a program slice—the set of all statements that *might* affect the value of a variable occurrence—is totally independent of the program input values. Program debugging, however, involves analyzing the program behavior under the specific inputs that revealed the bug. In this chapter, we addressed the dynamic counterpart of the static slicing problem—finding all statements that *really* affected the value of a variable occurrence for the given program inputs—and examined several approaches to computing dynamic slices.

Our prototype debugging tool, SPYDER, provides facilities for displaying both static and dynamic program slices. Figure 3.17 shows a program similar to that in Figure 1.1 except that it does not use any array or structure variables. Suppose this program is also executed for testcase #1 ($N = 2$ and the sides of the two triangles being (3, 3, 3) and (6, 5, 4), respectively). Figure 3.18 shows its static slice with respect to area at line 37. Figure 3.19 shows the corresponding dynamic slice when the execution is stopped there during the first loop iteration. Figure 3.20 shows the corresponding slice during the second iteration. Note that the static slice contains every statement except the assignments to sum on lines 13 and 37, because an assignment to sum does not affect the value of area; it is the other way round. In the case of the first dynamic slice, only the statements relevant to the computation of the area of an equilateral triangle belong to the slice. Although assignments on lines 18–20 are executed, they are not included in the slice because the values they compute do not contribute towards computation of the area of an equilateral triangle. The same assignments, however, are included in the second dynamic slice because the values they compute are responsible for incorrectly classifying the second triangle as a right triangle instead of a scalene triangle. Clearly, this second dynamic slice provides finer error localization information compared to the static slice.

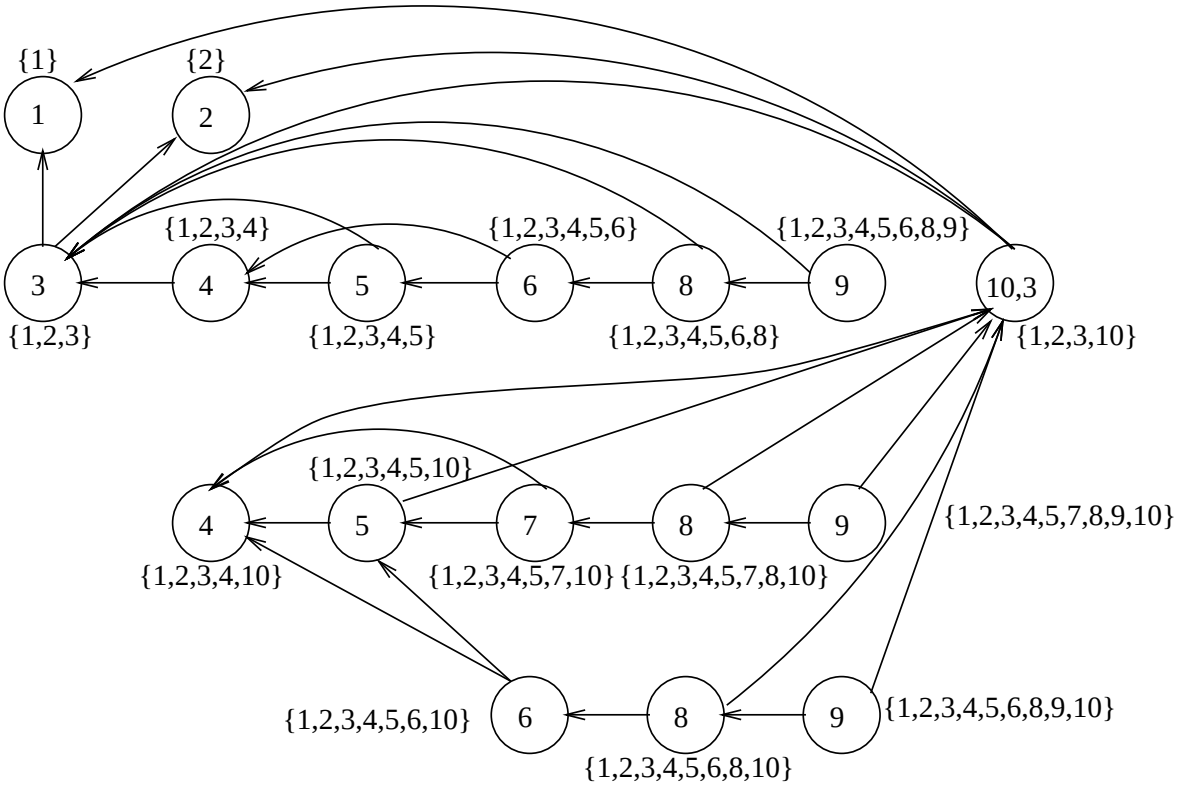

Figure 3.16 Reduced Dynamic Dependence Graph for the Program in Figure 3.11 for the testcase (N = 3, X = −4, 3, −2), obtained using *DynamicSlice5*. Each node is annotated with *ReachableStmts*, the set of all statements reachable from that node

SPYDER also provides a facility to obtain approximate dynamic slices using Approach 1 discussed in Section 3.4.2. Figure 3.21 shows the approximate dynamic slice with respect to area on line 37 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 shown in Figure 3.19. Figure 3.22 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 3.20 and the corresponding static slice shown in Figure 3.18.

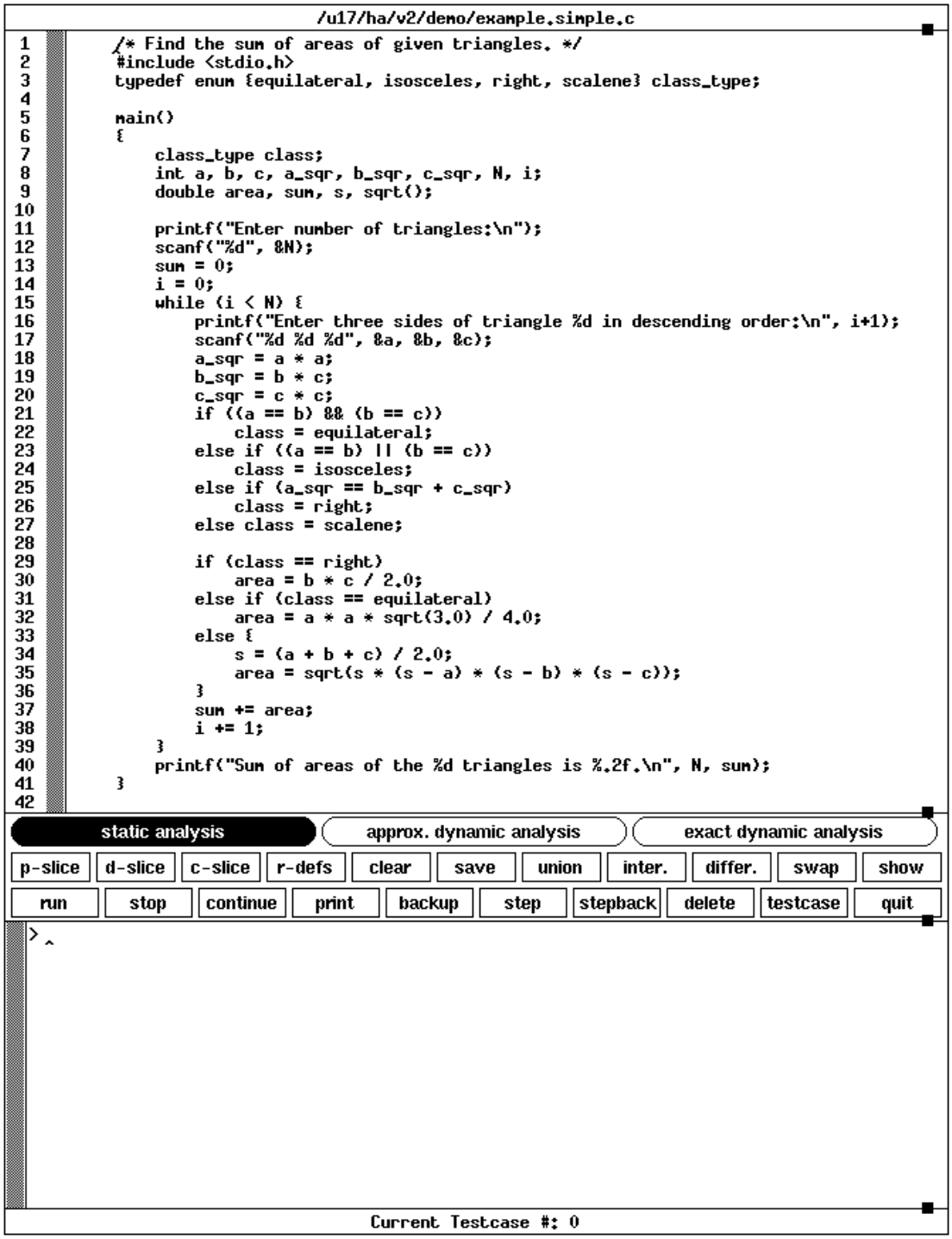

Figure 3.17 A variant of the program in Figure 1.1

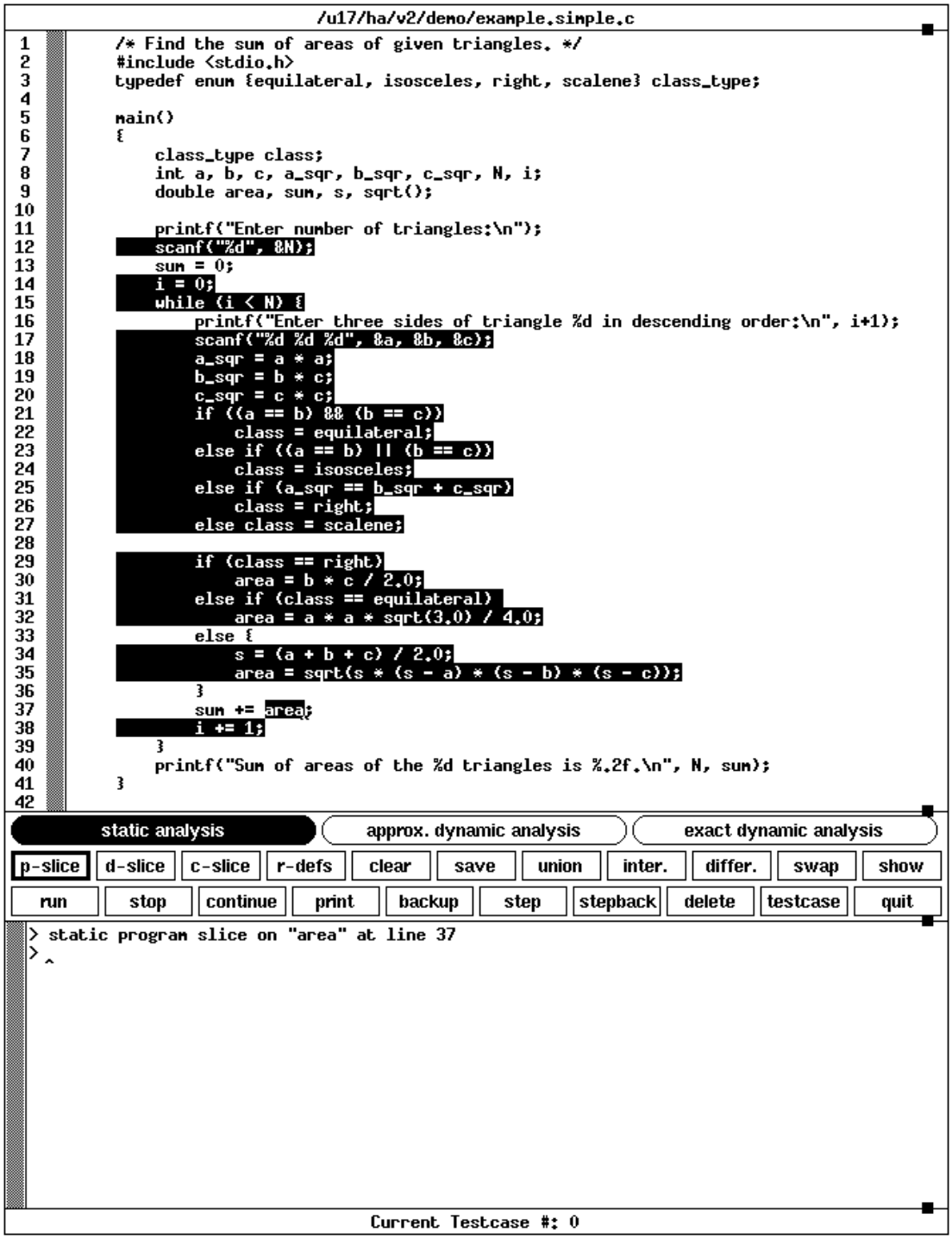

Figure 3.18 Static slice with respect to area on line 37.

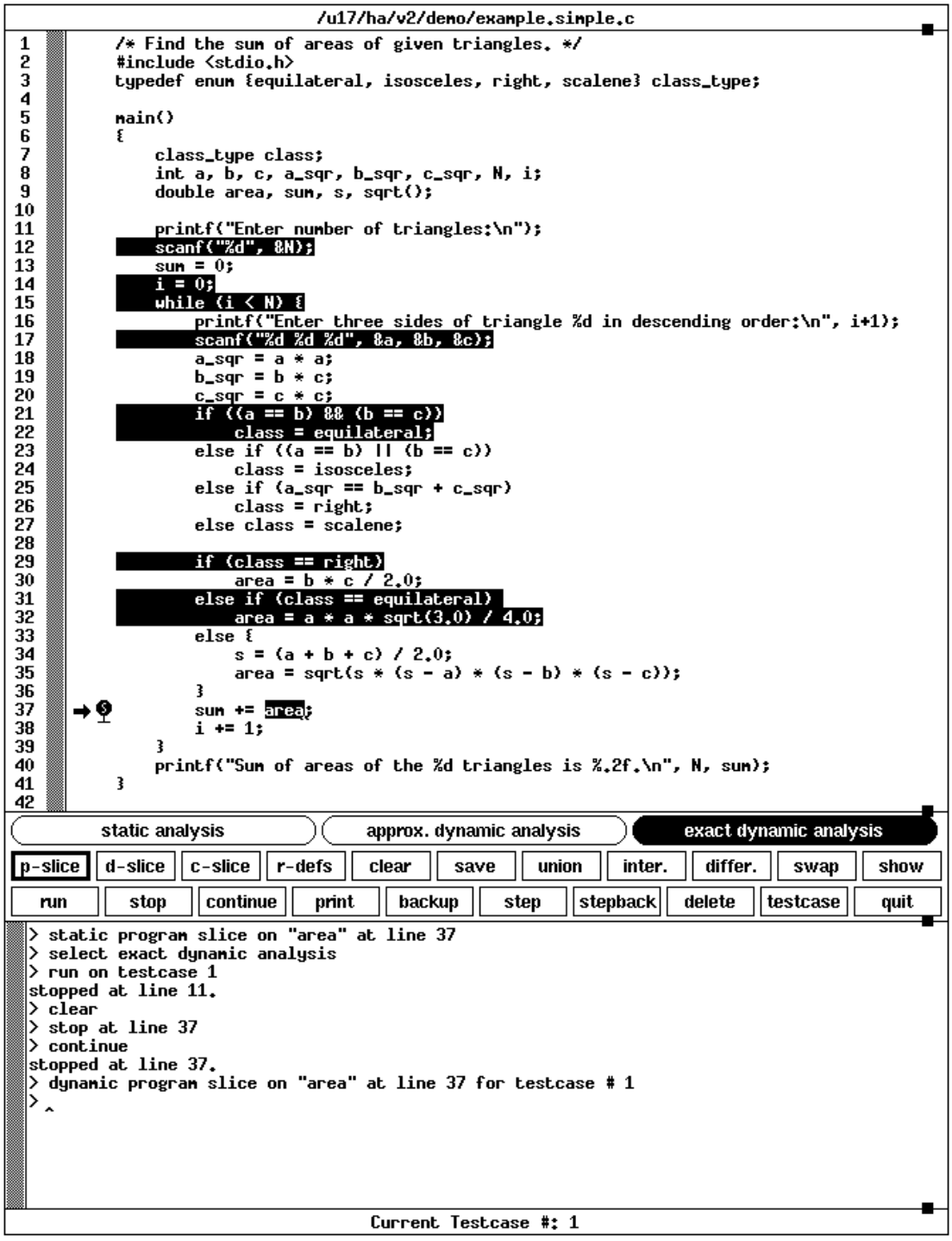

Figure 3.19 Dynamic slice with respect to area on line 37 during the first loop iteration.

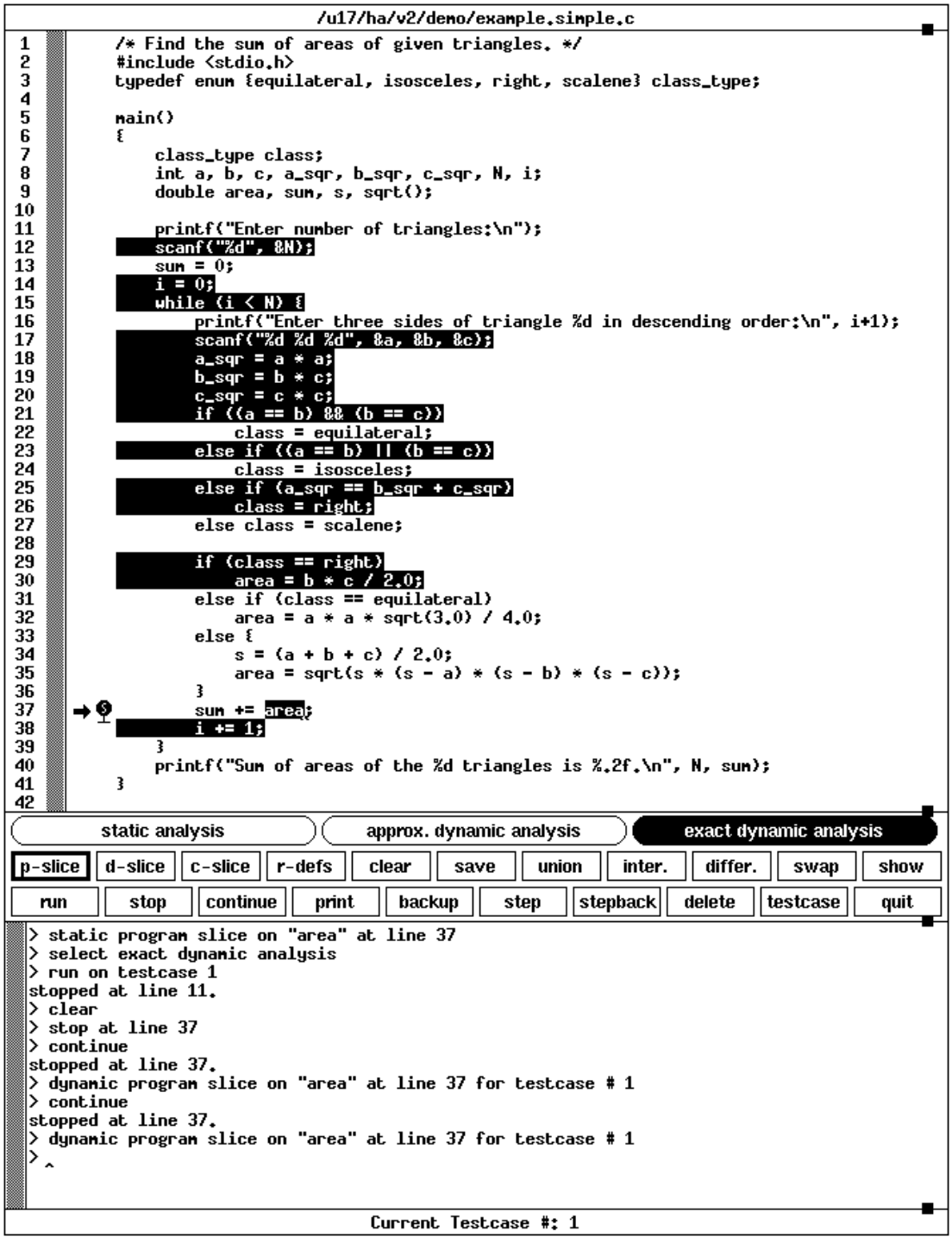

Figure 3.20 Dynamic slice with respect to area on line 37 during the second loop iteration.

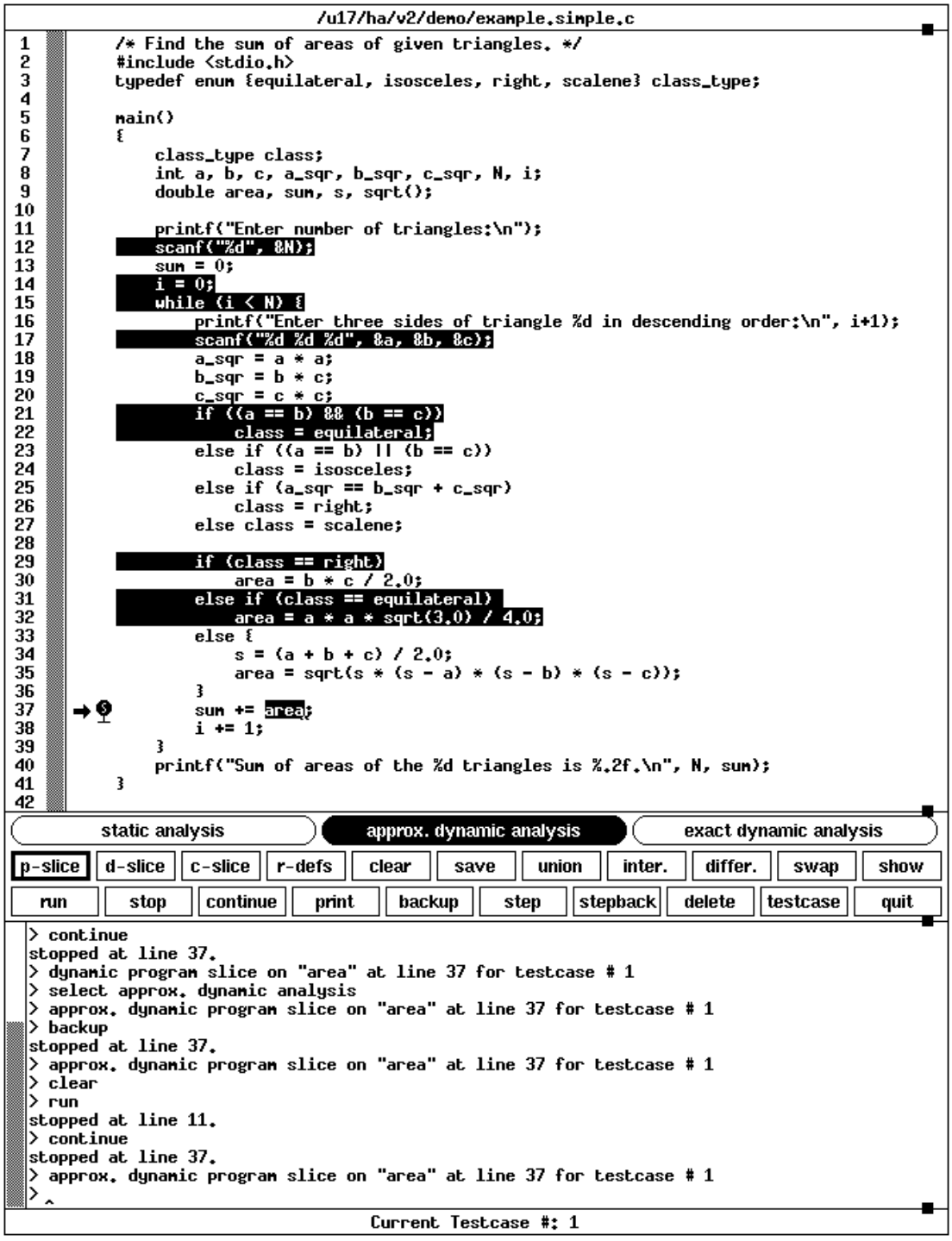

Figure 3.21 Approximate dynamic slice on area on line 37 during the first loop iteration.

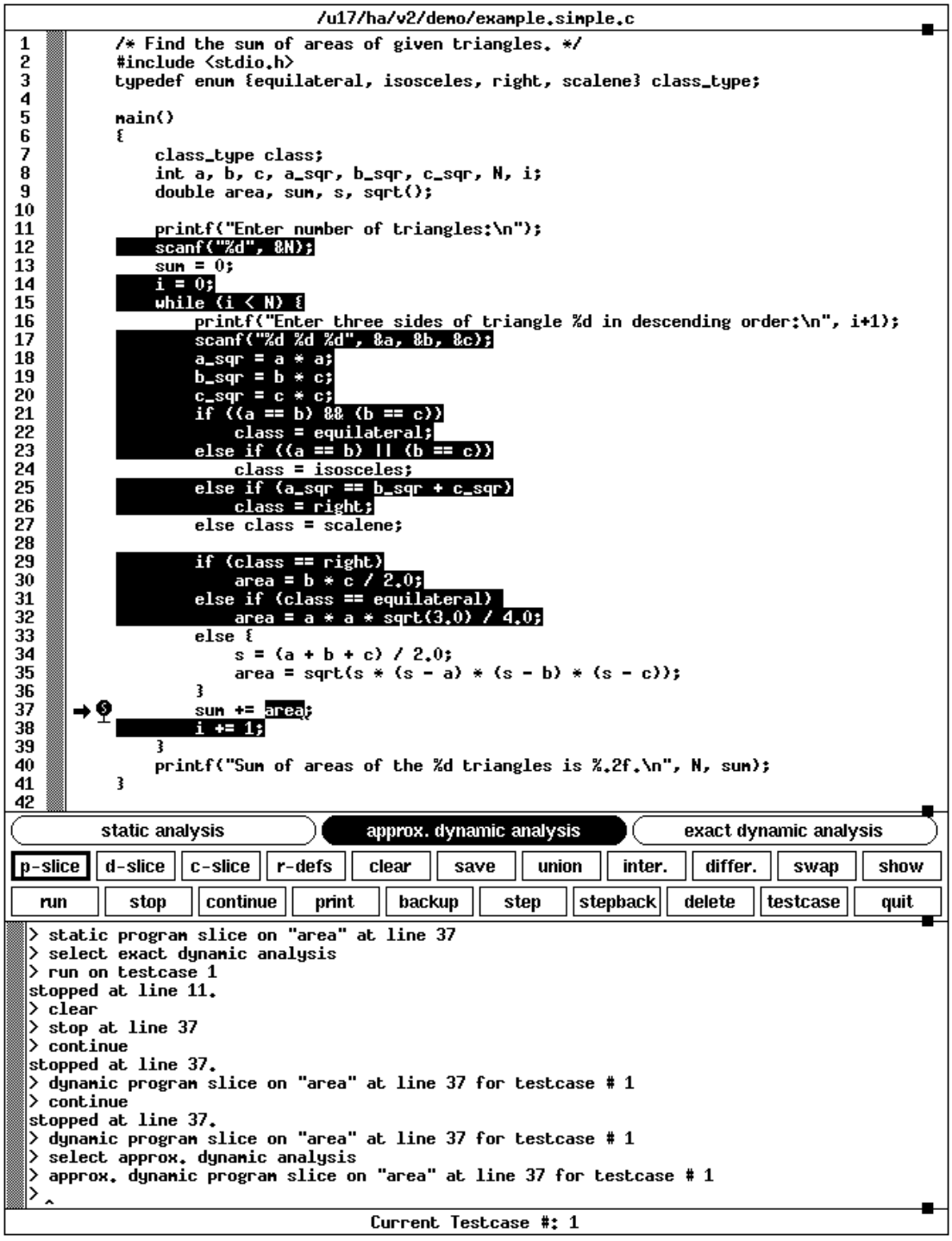

Figure 3.22 Approximate dynamic slice on area on line 37 during the second loop iteration.

4. COMPLETE DYNAMIC SLICING

In the previous chapter, we discussed several approaches to finding dynamic program slices of programs that involved only scalar variables. Slicing is even more useful in debugging programs that use complex data-structures involving pointers—when interstatement dependencies are hard to visualize by manual examination of the source code. In this chapter we discuss how the techniques described in the previous chapter can be extended to find slices of programs that involve pointers and composite variables.

Scalar variables are relatively easy to handle because the memory location that corresponds to a scalar variable is fixed and known at compile time; it does not vary during the course of program execution. Hence, if one statement modifies a scalar variable and another statement references a scalar variable, it is easy to determine, at compile time, if the latter statement references the same memory location modified by the former.

The chief difficulty in dealing with an indirect reference through a pointer or an array element reference[1] is that the memory location referenced by such an expression cannot, in general, be determined at compiled time. Further, when such a reference occurs inside a loop, the memory location referenced may vary from one loop iteration to another. The difficulty is compounded if the language used is not strongly-typed and permits integer arithmetic over pointer variables. Techniques proposed in [CWZ90, HPR89a, LH88] may be used to obtain conservative static approximations of what a pointer might point to at run time, but in the presence of unconstrained pointers, as in C, such analysis has only limited usefulness. In this case we are forced to make the most conservative assumption: an indirect assignment through a pointer

[1]Not regarding an array as a single unit.

can potentially define *any* variable. The outcome of this assumption is that static slices of programs involving pointers tend to be very large; in many instances they include the whole programs themselves. Fortunately, it is possible to perform precise dynamic dependence analysis even when the language is not strongly-typed.

In this chapter, we present an approach to obtain dynamic program slices when the language permits unconstrained pointers. Besides pointers, composite variables such as arrays, records, and unions are also handled uniformly under this approach. It also allows precise interprocedural dynamic slicing to be performed. We first present, in Section 4.1, a general framework to obtain static slices in the presence of pointers and composite variables, and then extend it to the dynamic case in Section 4.2. While the static slicing algorithm assumes that an indirect assignment may potentially modify any variable, the dynamic slicing algorithm detects exact dependencies. Section 4.3 discusses how our approach may be extended to the interprocedural case.

4.1 Static Slicing with Pointers and Composite Variables

In Section 3.2.3, we defined reaching definitions for scalar variables. A definition of a variable var at a statement S_1 reaches its use at statement S_2, if there is a path from S_1 to S_2 in the flow-graph of the program such that no other node along the path defines var. But what happens if S_1 defines an array element $A[i]$, and S_2 uses an array element $A[j]$; if S_1 defines a field of a record $s.f$, and S_2 uses the whole record s; or if S_1 defines a variable X, and S_2 uses a pointer dereference expression $*p$? To be able to handle such situations, we first introduce the notion of intersection of two l-valued expressions.

4.1.1 Intersection of L-valued Expressions

An expression is said to be an l-valued expression if a memory location can be associated with it. A simple check to find if an expression is an l-valued expression is to check if it can appear on the left hand side of an assignment statement. For example, expressions var, $A[i]$, $s.f$, $B[i].r.x$, $*p$, are all l-valued expressions. On the

other hand, none of the expressions 103, $x + y$, or $a > b$, is l-valued. The presence of pointers and composite variables such as arrays and records in a programming language requires that both *use* and *def* sets of statements be defined in terms of l-valued expressions.

A *use* expression e_1 is said to *intersect* with a *def* expression e_2, if the memory location associated with e_1 may overlap with that associated with e_2. We identify three types of intersections between l-valued expressions: *complete intersection, maybe intersection*, and *partial intersection*.

4.1.1.1 Complete Intersection

A *use* expression e_1 completely intersects a *def* expression e_2 if the memory location associated with e_1 is totally contained in that associated with e_2. For example, consider the following code fragment:

```
S1:     X := ...
          ⋮
S2:       := ...X ...
```

Here, use of variable X at S2 completely intersects its definition at S1. Also, in the following code fragment,

```
S1:     s := ...
          ⋮
S2:       := ...s.f ...
```

use of field $s.f$ at S2 completely intersects the definition of record s at S1.

4.1.1.2 Maybe Intersection

Consider the following situation:

```
S1:     A[i] := ...
```

```
                    ⋮
S2:                 := ... A[j] ...
```

Whether or not the use of $A[j]$ at S2 intersects with the definition of $A[i]$ at S1 depends on the actual values of variables i and j at statements S1 and S2, respectively. If their values are the same, the two expressions intersect, otherwise they do not. We refer to such intersections as *maybe* intersections. Use of pointer dereferencing also causes *maybe* intersections. In the following code fragment,

```
S1:         *p := ...
                ⋮
S2:             := ... X ...
```

use of variable X at S2 *maybe*-intersects with the definition at S1 because the pointer variable p may or may not be pointing at variable X.

4.1.1.3 Partial Intersection

Consider the following scenario:

```
S1:         s.f := ...
                ⋮
S2:             := ... s ...
```

The whole record s is used at S2, but only one of its fields is defined at S1. A similar situation occurs if an array is used at S2, and only one of its elements is defined at S1. We refer to such intersections as *partial* intersections. If a *use* expression e_1 partially intersects with a *def* expression e_2, we define $PreExp(e_1, e_2)$ to be the portion of the memory location associated with e_1 that lies before that associated with e_2. Similarly we define $PostExp(e_1, e_2)$ to be the portion of the memory location associated with e_1 that lies after that associated with e_2.

4.1.2 Static Reaching Definitions Revisited

Let *CompleteIntersect*, *MaybeIntersect*, and *PartialIntersect* be boolean functions that determine if two l-valued expressions have *complete, maybe*, or *partial* intersections, respectively. We can now extend our definition of *StaticReachingDefns*, defined in Section 3.2.3 for programs involving only scalar variables, to that involving pointers and composite variables.

$StaticReachingDefns(var,\ n,\ \mathcal{F}) =$

$\quad let\ \mathcal{F} = (V,\ A)$

$\quad in\ \bigcup_{(x,n)\in A}\ if\ def(x) = \phi$

$\qquad\qquad then\ StaticReachingDefns(var,\ x,\ (V,\ A-\{(x,n)\}))$

$\qquad\qquad else\ let\ def(x) = \{var'\},\ A' = A-\{(x,n)\}$

$\qquad\qquad\quad in\ if\ CompleteIntersect(var,\ var')$

$\qquad\qquad\qquad then\ \{x\}$

$\qquad\qquad\qquad else\ if\ MaybeIntersect(var,\ var')$

$\qquad\qquad\qquad\quad then\ \{x\} \cup StaticReachingDefns(var,\ x,\ (V,\ A'))$

$\qquad\qquad\qquad\quad else\ if\ PartialIntersect(var,\ var')$

$\qquad\qquad\qquad\qquad then\ let\ e_1 = PreExp(var,\ var'),$

$\qquad\qquad\qquad\qquad\qquad e_2 = PostExp(var,\ var')$

$\qquad\qquad\qquad\qquad\quad in\ \{x\} \cup StaticReachingDefns(e_1,\ x,\ (V,\ A'))$

$\qquad\qquad\qquad\qquad\qquad\quad \cup StaticReachingDefns(e_2,\ x,\ (V,\ A'))$

$\qquad\qquad\qquad\qquad else\ StaticReachingDefns(var,\ x,\ (V,\ A'))$

Note that both *maybe* and *partial* intersections may occur together. For example, consider the following situation:

```
S1:     A[i].f := ...
                  ⋮
S2:             := ...A[j] ...
```

Because we check for *maybe* intersection before *partial* intersection, the former takes precedence over the latter whenever they occur together.

The definitions of data-, control-, and program dependence, and that of a static slice remain the same as given in Sections 3.2.4, 3.2.5, 3.2.6, and 3.3, respectively. Only we now use the new definition of static reaching definitions described above to find the data dependence edges of the program dependence graph.

4.2 Dynamic Slicing with Pointers and Composite Variables

Dynamic slicing differs from static slicing in that the former has no *maybe* intersections. This implies that for each use of a scalar variable, there is at most one dynamic reaching definition; and for each use of a composite variable, there is at most one dynamic reaching definition of each of its scalar components. To define dynamic slices in the presence of composite variables and pointers, we generalize the notion of an l-valued expression to that of a memory cell.

4.2.1 Use and Def Sets Revisited

A *memory cell* is a tuple (adr, len) where adr represents its address in memory, and len represents its length in bytes.[2] The memory-cell corresponding to an l-valued expression e_1 is given by the tuple ($AddressOf(e_1)$, $SizeOf(e_1)$), where $AddressOf(exp)$ gives the current address associated with the l-valued expression exp at runtime, and $SizeOf(exp)$ gives the number of bytes required to store the value of exp.

We now define *use* and *def* sets of all simple statements and predicates in terms of memory cells instead of l-valued expressions. Though the length component of these memory-cells may be determined at compile time, the address components may

[2]Or the smallest addressable unit on the computer, e.g. a word. For languages where memory allocation for a variable is not necessarily contiguous, definition of a memory-cell may be changed to include the set of all its contiguous subcells.

have to be determined at runtime just before the corresponding simple statement or predicate is executed.

4.2.2 Dynamic Reaching Definitions Revisited

Now, instead of determining intersection of l-valued expressions, we check if two memory cells intersect. Using this formulation, we redefine *DynamicReachingDefns* function, earlier defined in Section 3.4.1, as follows:

$DynamicReachingDefns(cell, <>) = \phi$

$DynamicReachingDefns((adr, 0), hist) = \phi$

$DynamicReachingDefns(cell, hist) =$

$\quad let\ hist = <prevhist \mid next>$

$\quad in\ if\ def(next) = \phi$

$\quad\quad then\ DynamicReachingDefns(cell, prevhist)$

$\quad\quad else\ let\ def(next) = \{cell'\}$

$\quad\quad\quad in\ if\ CellIntersect(cell, cell')$

$\quad\quad\quad\quad then\ \{next\} \cup DynamicReachingDefns(PreCell(cell, cell'), prevhist)$

$\quad\quad\quad\quad\quad\quad \cup\ DynamicReachingDefns(PostCell(cell, cell'), prevhist)$

$\quad\quad\quad\quad else\ DynamicReachingDefns(cell, prevhist)$

$CellIntersect(useCell, defCell)$ returns true if there is any overlap between the two cells. *PreCell* and *PostCell* return the non-overlapping portions of the *useCell* that lie before and after the overlapping portion, respectively. It is possible that one or both of these portions may be empty ($len = 0$). The case when both pre- and post-cells are empty is analogous to *complete* intersection in static slicing; the case when one or both are non-empty is analogous to *partial* intersection; and, as we mentioned earlier, there are no *maybe* intersections in the dynamic case.

The advantage of this formulation is that all the usual problems associated with handling pointers in the static case are automatically taken care of in the dynamic case because all *use* and *def* sets are resolved in terms of memory cells; there is no

ambiguity in determining if two memory cells overlap. As in the case of static slicing, the definitions of dynamic dependence graph and dynamic slice remain the same as those given in Section 3.4.4. Only the definition of dynamic reaching definitions has changed.

4.3 Interprocedural Dynamic Slicing

The dynamic slicing approach described above can be easily extended to obtain slices of programs with procedures and functions. We first consider the case when parameters are passed by value, as in C. In this case, we simply need to treat a procedure invocation, $proc(actual_1, actual_2, \ldots, actual_n)$, to be a sequence of assignments $formal_i = actual_i, 1 \leq i \leq n$, where $formal_i$ is the ith formal parameter of *proc*. The *use* set of each of these assignments is computed in terms of memory-cells just before the procedure is invoked, and the *def* set is computed just after the control enters the procedure. Memory cells that correspond to *def* sets belong to the current activation record of *proc* on the stack.

Note that unlike interprocedural static slicing [HRB90], our approach for dynamic slicing does not require that we determine which global variables are referenced inside a procedure, or which variables may be aliases to each other, nor do we need to eliminate name conflicts among variables in different procedures.

Call-by-reference is even easier to handle: no initial assignments to formal parameters need to be made. The address of a formal parameter variable is automatically resolved to that of the corresponding actual parameter. Call-by-result parameter passing is handled by making assignments, $actual_i = formal_i$, just before control returns to the calling procedure. Call-by-value-result can be handled similarly by making appropriate assignments both at the beginning and the end of the procedure.

4.4 Summary

Static program slices tend to be over large and imprecise when the program being debugged involves pointers and composite variables such as arrays, records, and unions (see examples below). They lose their usefulness altogether if the language involved is not strongly-typed and permits use of unconstrained pointers. In this chapter we have shown that we can find accurate dynamic slices even in the presence of unconstrained pointers and composite variables. The approach outlined provides a uniform framework for handling pointers as well as various types of composite variables. It does not require that the language be strongly-typed or that any runtime checks (out-of-bound array element reference, illegal pointer dereference, etc.) be performed.

Figure 4.1 shows a simple program involving pointers. 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 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 = 1, j = 3, k = 3). Figure 4.1 also shows the static program slice with respect to a[i] on line 29. Figure 4.2 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 lines 7–16 also be included in the slice. The dynamic program 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, a[1], is in the slice; assignments to all other elements of the array do not belong to the slice. If we obtain the dynamic program slice for a[j]

on line 29, both indirect assignments on lines 26 and 27 are in the dynamic slice, as shown in Figure 4.3. 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[3].

Figure 4.4 shows a variant of the above program where a loop is used to initialize the array instead of using a separate assignment for each array element. If we execute this program for the same testcase (i = 1, j = 3, k = 3), we get the following output: a[1] = 2, a[3] = 4, a[10] = 0. Instead of printing the value of a[3] it prints that of a[10]. This implies that the value of k somehow got corrupted during the program execution. If we obtain the dynamic slice of k on line 27, we would expect only line 8 to be in the slice as that is the only place in the program where k is modified. Instead, we find that the loop on lines 10–17 is included in the dynamic slice, as shown in Figure 4.4. This suggests that the variable k was clobbered during the execution of the loop. Further examination reveals that the fault indeed lies either with the loop predicate: it iterates ten times when the array is declared to be only eight elements long, or with the array declaration: it is declared to be eight elements long instead of being ten elements long. Figure 4.5 shows the memory allocation made by our compiler for all variables along with their contents at the end of the program execution for the above testcase[3]. Note that the memory location that corresponds to variable k indeed overlaps with that for a[9]. It is situations like this where precise dynamic analysis in terms of memory cells is invaluable in revealing the fault.

The program in Figure 4.6 shows the same program as in Figure 1.1 except that the segment of code that determines the class of a triangle has been moved into a procedure. Figure 4.6 also shows the inter-procedural dynamic slice with respect to area on line 53 during the second loop iteration in the main program when the program is executed for the testcase: $N = 1$ and the sides of the two triangles are (3, 3, 3) and (6, 5, 4).

[3]Memory allocation will vary from compiler to compiler.

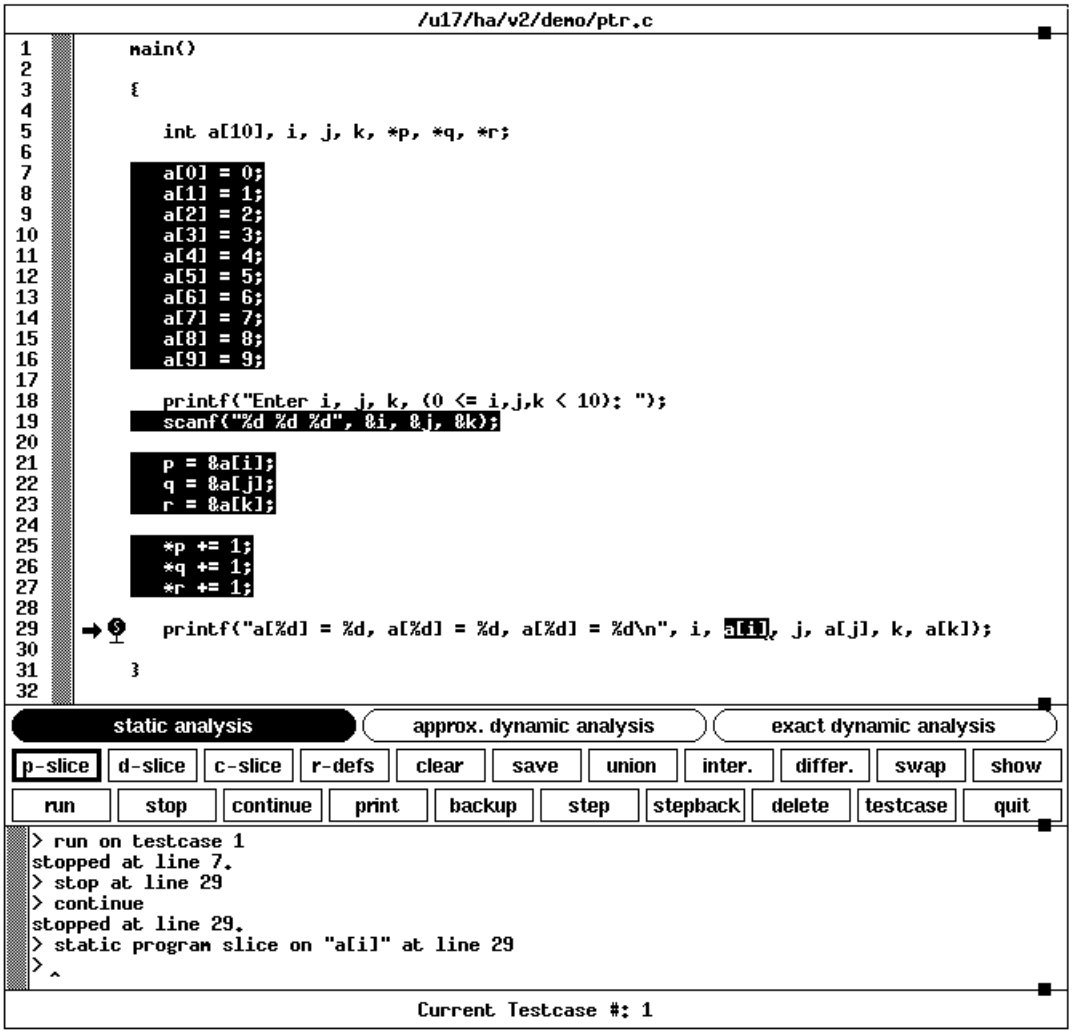

Figure 4.1 Static slice with respect to a[i] on line 29.

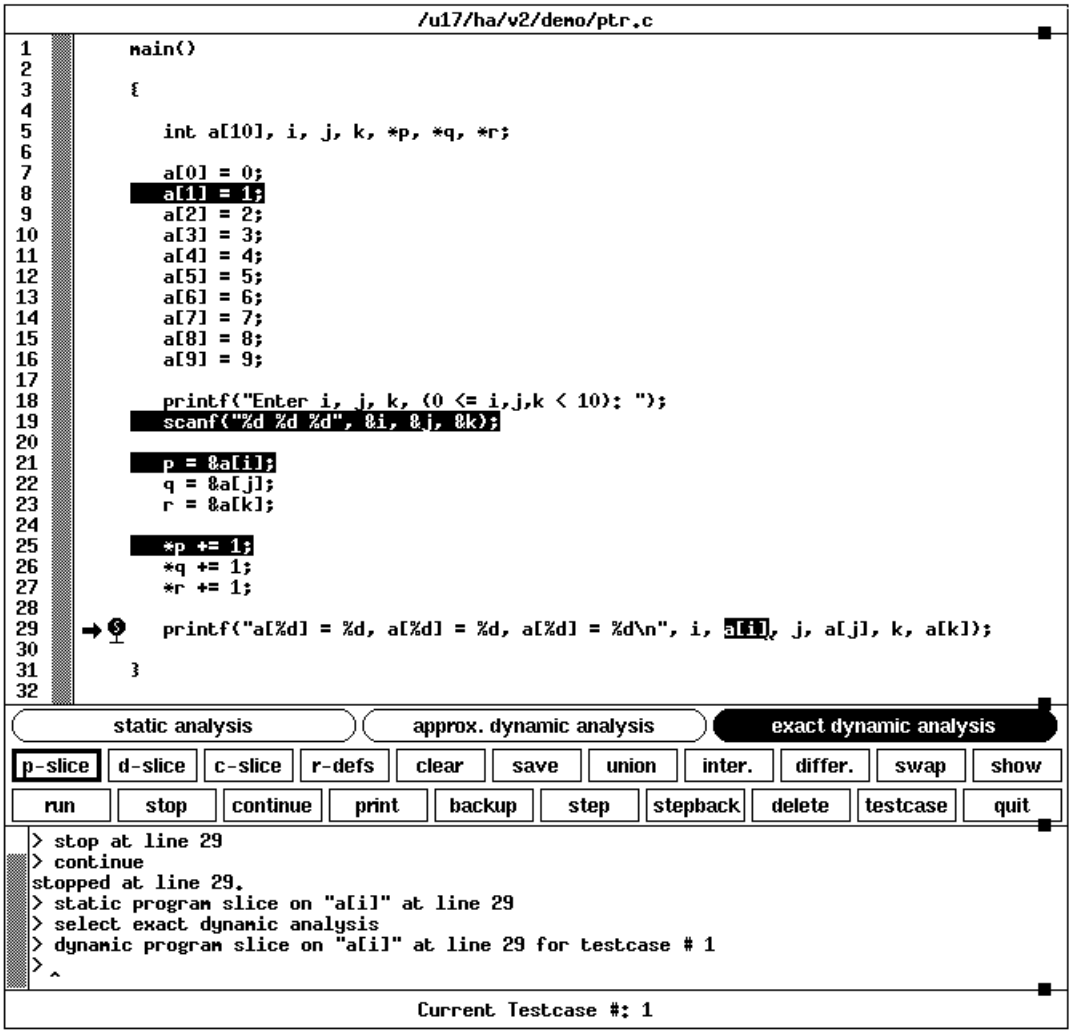

Figure 4.2 Dynamic slice with respect to a[i] on line 29.

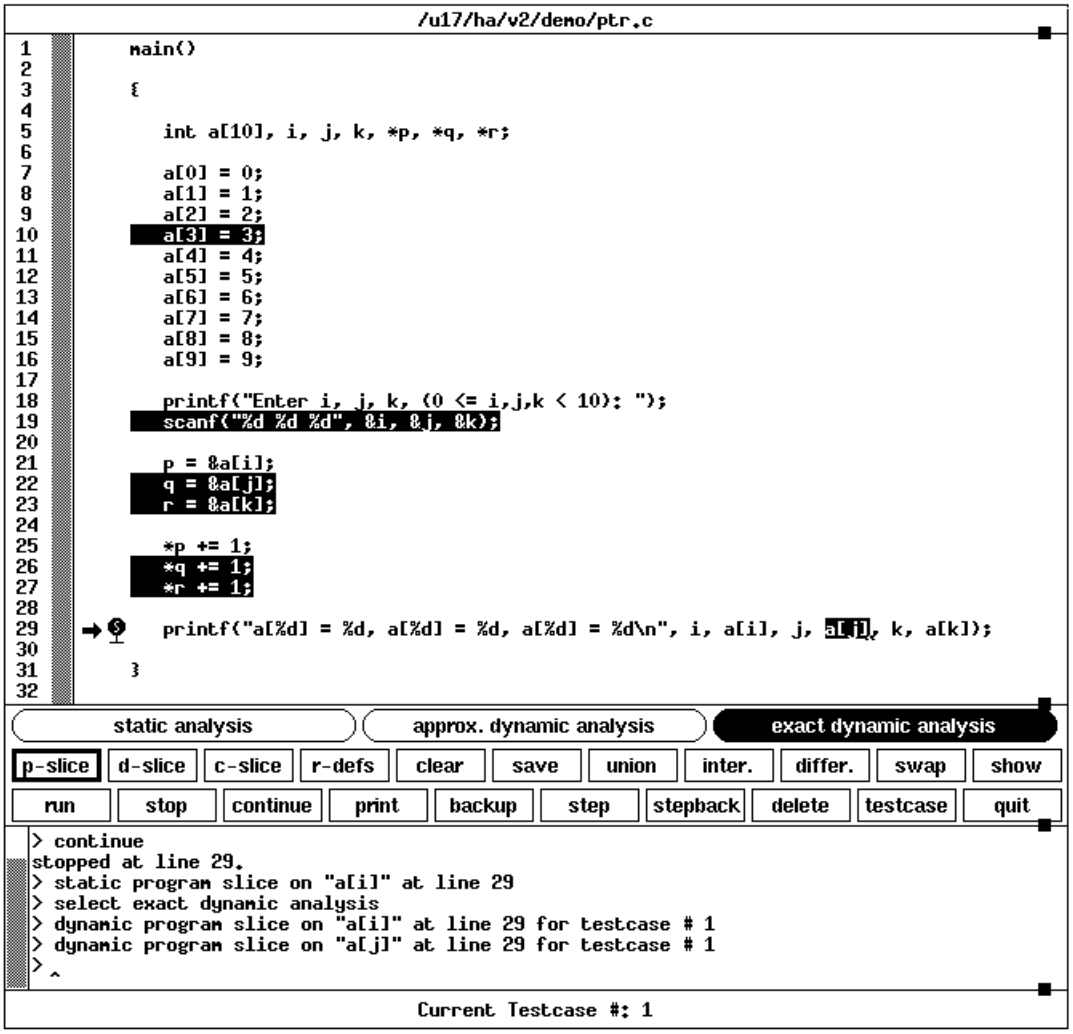

Figure 4.3 Dynamic slice with respect to a[j] on line 29.

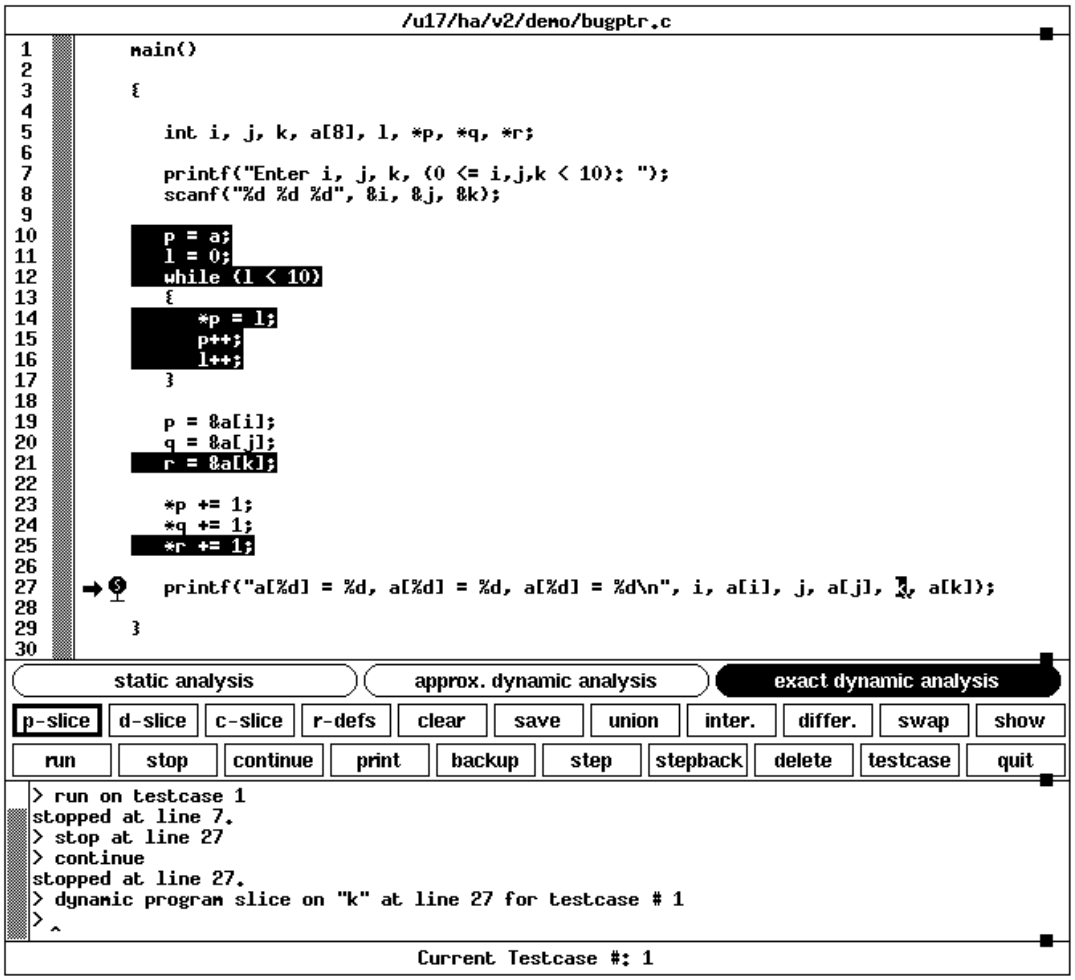

Figure 4.4 Dynamic slice with respect to k on line 27.

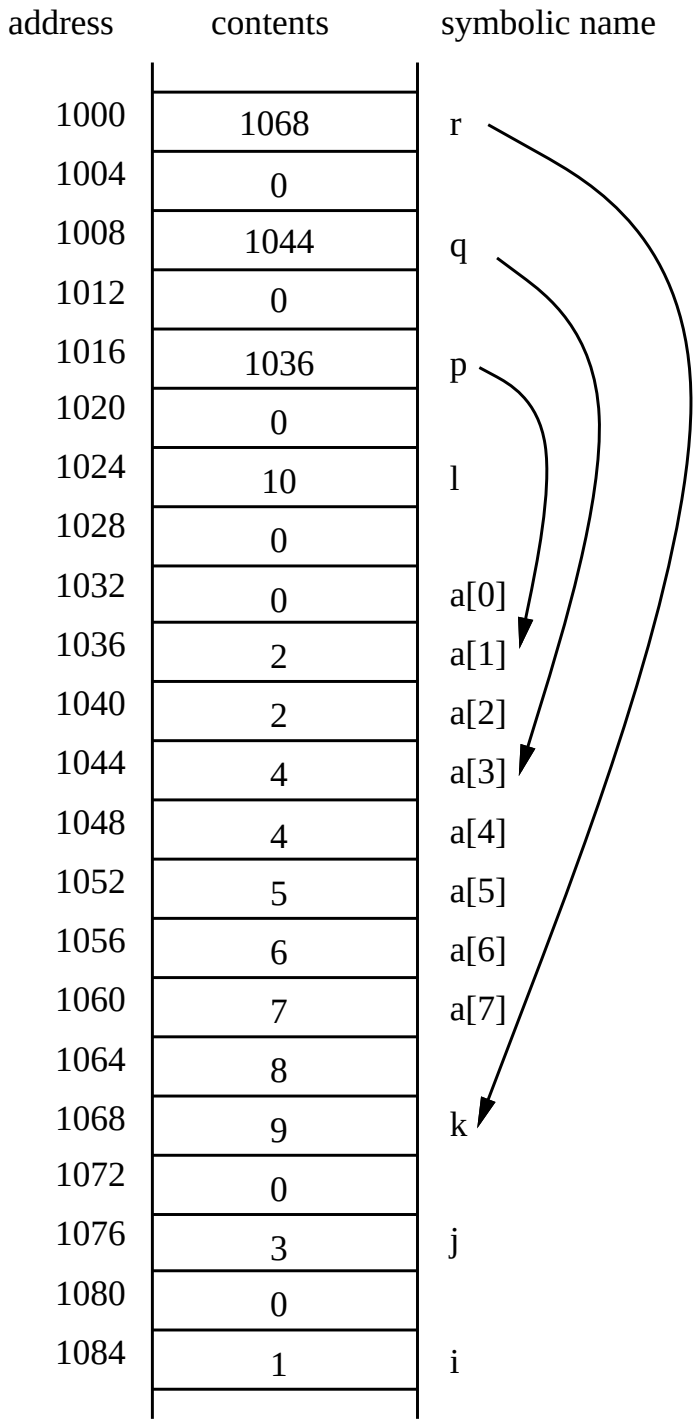

Figure 4.5 Storage layout of the program in Figure 4.4 at the end of the program execution for the testcase (i = 1, j = 3, k = 3).

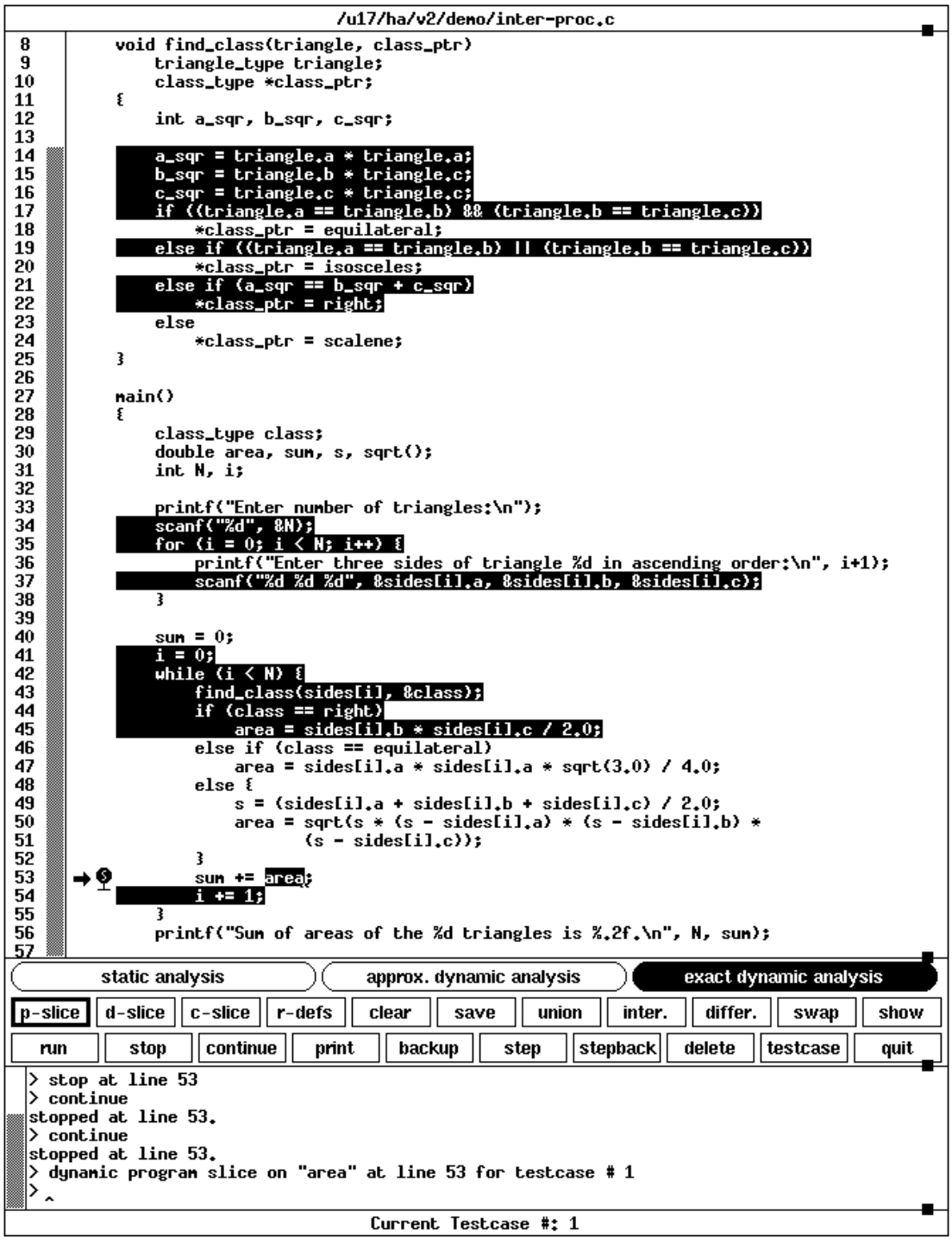

Figure 4.6 Interprocedural dynamic program slice with respect to area on line 53 during the second loop iteration.

5. LOCAL V/S GLOBAL SLICING

In Chapters 3 and 4 we informally defined a dynamic slice with respect to a variable, a program location, and a testcase to be the set of those statements that affect the value of the given variable at the given location when the program is executed for the given testcase, and presented a constructive definition of a dynamic slice. In this chapter we take another look at what it really means for a statement to "affect" the value of a variable at a given location or for a statement to have "influenced" the control reaching a certain location.

5.1 Local Analysis

A program fault manifests itself, directly or indirectly, in one of the following ways:

Case 1. Value of an expression, exp, at location, L, is observed to be incorrect.

Case 2. Control has incorrectly reached a location, L, it should never have reached.

Case 3. Control didn't reach the desired location, L.

Case 1

If the value of exp at location L is incorrect, there are two possibilities:

Case 1.1. Function computed by exp is incorrect, e.g., exp should have been $x + y$ instead of $x - y$. If we have determined this, we have localized the fault.

Case 1.2. Value of one (or more) of the variables, var, used in exp is incorrect, e.g., value of expression $x + y$ is incorrect because the value of x is incorrect. In this case, we must find the dynamic reaching definition, R, of var at L. Note that

there may be several *static* reaching definitions of exp at L, but there is only one corresponding *dynamic* reaching definition.[1]

Having found the unique reaching definition R of the erroneous variable var at L, there are two further possibilities:

Case 1.2.1. Value of the expression, $exp1$, computed at R is incorrect. In this case, we are back to Case 1 with respect to the value of $exp1$ computed at R.

Case 1.2.2. R is the wrong reaching definition of var. If this is the case, there are four further possibilities:

Case 1.2.2.1. Control shouldn't have reached R, in which case we are back to Case 2 with respect to the location of R.

Case 1.2.2.2. The correct definition of var is missing: either there is a missing assignment to var along the path between R and L, or one of the assignments along this path is incorrectly assigning a value to the wrong variable. If we have determined this, we have localized the fault.

Case 1.2.2.3. Control correctly reached R, but it didn't reach the correct definition, $R2$, of var because it took the wrong path between R and L. In this case, we are back to Case 3 with respect to location of $R2$.

Case 1.2.2.4. R shouldn't have been included in the program at all. In this case, as we have discovered an extraneous assignment, we have localized the fault.

Case 2

If control shouldn't have reached the location L, there are two possibilities: there is no predicate enclosing L, or L is immediately enclosed by a predicate, $pred$.

[1] If var is a composite variable, e.g., an array or a record, we find the reaching definition of the scalar component(s) of var with the wrong value.

Case 2.1. L is not enclosed by any predicate. If control shouldn't have reached L, it should have been enclosed by a predicate that would have prevented control from reaching L. This means there is a missing predicate enclosing L. In this case, we have localized the fault.

Case 2.2. L is immediately enclosed by a predicate, *pred*, e.g., an if-then-else or a while predicate. Again, there are two possibilities:

Case 2.2.1. Control shouldn't have reached *pred* either. Then we are back to Case 2 with respect to location of *pred*.

Case 2.2.2. Control should have reached *pred* but not L. In this case, there are two more possibilities:

Case 2.2.2.1. *pred* evaluated incorrectly, in which case we are back to Case 1 with respect to the value of *pred*.

Case 2.2.2.2. There is a missing predicate along the path between *pred* and L. If we have found this, we have localized the fault.

Case 3

If control didn't reach the desired location L, it must be immediately enclosed by a predicate, *pred* which must have prevented control from reaching L. There are three possibilities in this case:

Case 3.1. Control didn't reach *pred* either. In this case, we are back to Case 3 with respect to the location of *pred*.

Case 3.2. Control correctly reached *pred* but not L. In this case, *pred* evaluated incorrectly, so we are back to Case 1 with respect to the value of *pred*.

Case 3.3. *pred* should never have been included in the program. As we have found an extraneous predicate in this case, we have localized the fault.

5.2 Global Analysis

For large programs, the above step by step analysis may be much too tedious to perform. If the fault is far removed from the location where it is manifested, it may take a long time before we find the fault. Notice that in the above analysis many times we need to recursively follow one of the three Cases 1, 2, or 3. The basis situations, which imply the end of the search, are:

- An assignment statement is found to compute an incorrect function.
- A predicate expression is found to compute an incorrect function.
- An assignment statement is found to assign a value to a wrong variable.
- The desired assignment to a variable is missing.
- The desired predicate expression guarding a given statement is missing.
- An extraneous assignment is found to be present in the program.
- An extraneous predicate is found to be present in the program.

The inductive steps that require recursive application of Cases 1, 2, or 3 are:

- We find the dynamic reaching definition of a variable at a given location.
- We find the predicate immediately enclosing a given statement.

Sometimes fault localization may be expedited if, in the above analysis, we combined many successive recursive steps into one. With this in mind, let us revisit the three cases discussed above.

Case 1 Revisited

If the value of an expression, exp, at location L is incorrect, there are the following possibilities:

- There exists an incorrect assignment, A, such that the wrong value computed by A has propagated into the value of exp at L through a transitive closure of reaching definitions.
- An assignment, A, whose computation should have propagated into the value of exp at L is missing from the program.
- An assignment, A, whose computation has propagated into the value of exp at L should never have been executed.
- An assignment, A, whose computation should have propagated into the value of exp at L never got executed.
- An assignment, A, whose computation has propagated into the value of exp at L should never have been included in the program.
- A assignment, A, inside a loop, got executed an incorrect number of times (more or less than necessary).

Case 2 Revisited

If control incorrectly reached a location, L, then there are two possibilities:

- There is a missing predicate enclosing the given location (not necessarily enclosing it immediately; it may be enclosing it several nesting levels up).
- One of the several predicates that are enclosing the given location has evaluated incorrectly.

Case 3 Revisited

If control didn't reach a given location, L, there are again two possibilities:

- One of the several predicates enclosing L must have evaluated incorrectly.
- One of the predicates enclosing L should never have been included in the program.

5.2.1 Dynamic Data Slice

In Case 1 of the global analysis, we need to find all assignments whose computations have propagated into the current value of *exp*. This can be done by taking the transitive closure of dynamic reaching definitions of variables used in *exp* at the given location. We call the set of all assignments that belong to this closure as the "dynamic *data* slice" with respect to the given expression, location, and testcase. If we know that the current value of *exp* is incorrect, then by examining its dynamic data slice, we can find if a relevant assignment is missing, or if one of the assignments computes a wrong function. Similarly, by examining the dynamic data slice, we can also check if one of the assignments that shouldn't be present in the slice is present there, and vice versa. If it is the former case (missing or incorrect assignment), we have localized the fault. If it is the latter case (wrong assignment reached or the correct assignment not reached), we are one step closure to finding the fault: we should continue our search using Case 2 or 3 of global analysis with respect the new location. If, on the other hand, examining the data slice does not suggest any of these two cases, it indicates that the error must have been caused because an assignment in the data slice was executed an incorrect number of times. That is, the fault lies with the execution frequency of an assignment, not with the assignment itself. This means one of the loop predicates enclosing assignments in the data slice must be faulty. We may have to resort to local analysis described above to detect such situations.

5.2.2 Control Slice

In Cases 2 and 3 of the global analysis, we need to find all the predicates that enclose a given location. This can be done by taking the transitive closure of control predicates starting with the given location. We call the set of all predicates that belong to this closure as the "*control* slice" of the given location. If we know that control has incorrectly reached a given location, we can examine the control slice and check if a relevant predicate is missing, or if one of the predicates has evaluated incorrectly. In the former case, we have localized the fault. In the latter case, we

are again one step closer to finding the fault: the predicate that evaluated incorrectly either computes an incorrect function, or one of the arguments to the function has a wrong value. In the former case, we have found the fault; in the latter case we continue our search using Case 1 of global analysis with respect to the incorrect argument.

5.2.3 Dynamic Program Slice

Note that in the global analysis, we keep alternating between data and control slices until we have localized the fault. Often times a fault manifests itself indirectly several levels of indirection away from the fault itself. In such situations it may be possible to localize the fault more quickly if we determined the closure of all relevant data and control slices. That will give us the set of *all* statements, assignments as well as predicates, that have any effect on the variable and/or location in question. If we find that the value of an expression at some location is incorrect, we first find its dynamic data slice. Then for each assignment in the data slice we find its control slice. Next, for each predicate in the control slice we find its dynamic data slice, and so on, until we reach a situation when no new statements can be added to this set. We call the set of statements so obtained as the "dynamic *program* slice" with respect to the given *expression* at the given location. Similarly, if we find that control has incorrectly reached a given location, or if control didn't reach a given location, we first find the control slice of that location. Then for each predicate in the control slice, we find its dynamic data slice. Then for each assignment in the data slice we find its control slice, and so on, until the situation is reached when no new statements can be added. We call the resulting set as the dynamic program slice with respect to the given *location*. So a dynamic program slice is really the transitive closure of data and control slices with respect to each expression and location in the dynamic data/control slice with respect to the variable/location in question.

5.2.4 Static Slices

Just as we defined a dynamic data slice, we may define *static data slice* with respect to a variable and a location to be the transitive closure of static reaching definitions of the given variable at the given location.

Notice that while defining a control slice above we did not use the word 'dynamic' in front of it. This is because, unlike reaching definitions, a statement always has at most one predicate immediately enclosing it. No narrowing of enclosing predicates can occur at run time. Thus the control slice with respect to a given location remains the same in both static and dynamic cases. If control incorrectly reaches a statement during program execution, we must examine all predicates enclosing the statement; if, on the other hand, a desired statement is *not* reached during program execution, we must still examine the same set of predicates.

The transitive closure of all relevant static data slices and control slices gives us the *static program slice* with respect to the variable and/or location in question. A static program slice with respect to a variable at a given location includes all statements that *could* affect the value of the variable observed at the given location when the program is executed for *any* testcase. Unlike a dynamic slice, a static program slice also has another property: it is an executable program itself. It evaluates the variable in question identically to the original program for *all* testcases [Wei84].

5.3 Summary

A program slice may be viewed as a closure of the corresponding data and control slices. A data slice is the closure of relevant data dependencies alone while a control slice gives the closure of control dependencies alone. Depending on the size and complexity of the program, sometimes it may be better to examine data or control slices instead of examining their closure (program slice). On the other hand, if the person debugging the program is completely familiar with the program code, and the fault is several indirections removed from its manifestation, examining the program

slice may expedite fault localization. By the same token, if the person is not too familiar with the program code or if the fault is only one or two indirections away from its manifestation, the best course may be to follow direct reaching definitions or examine the enclosing predicate.

Our prototype debugging tool, SPYDER, provides mechanisms to allow both local as well as global analysis to be performed. Besides program slices, it also provides facilities for displaying data and control slices as well as direct reaching definitions. And all these may be obtained for both static and dynamic cases. For example, Figure 5.1 shows the static reaching definitions of variable area at line 43. Figure 5.2 shows the unique dynamic reaching definition for the same variable occurrence during the second iteration of the enclosing **while** loop for testcase #1.

Figure 5.3 shows the dynamic data slice for sum at line 46 for testcase #1. Note that the assignment on line 35 that computes the area of a right triangle is included in the data slice even though neither of the two triangles in this testcase are right triangles. This suggests that we should pursue Case 2 with respect to line 35. Figure 5.4 shows the corresponding static data slice.

Figure 5.5 shows the control slice with respect to the assignment on line 35. This statement is incorrectly reached during the second loop iteration when the program is executed for the testcase #1 because the enclosing predicate on line 34 evaluates incorrectly: it evaluates to *true* instead of *false* because the value of variable class examined by the predicate is incorrect at that instance. This suggests that we pursue Case 1 with respect to value of class on line 34.

Figure 5.6 shows the dynamic program slice for area at line 43 when the execution is stopped there during the second loop iteration for testcase #1. Note that the erroneous assignment on line 24 is included in the slice. Also note that assignment on line 37 that computes the area of the first triangle is not included in the slice because, in this case, the computation of area during one loop iteration does not affect that during another iteration. If, however, we obtained the dynamic program slice with respect to sum on line 46, both assignments on lines 35 and 37 will be included in the

slice because computation of area during each iteration affects the final value of sum. Figure 5.7 shows the corresponding static program slice.

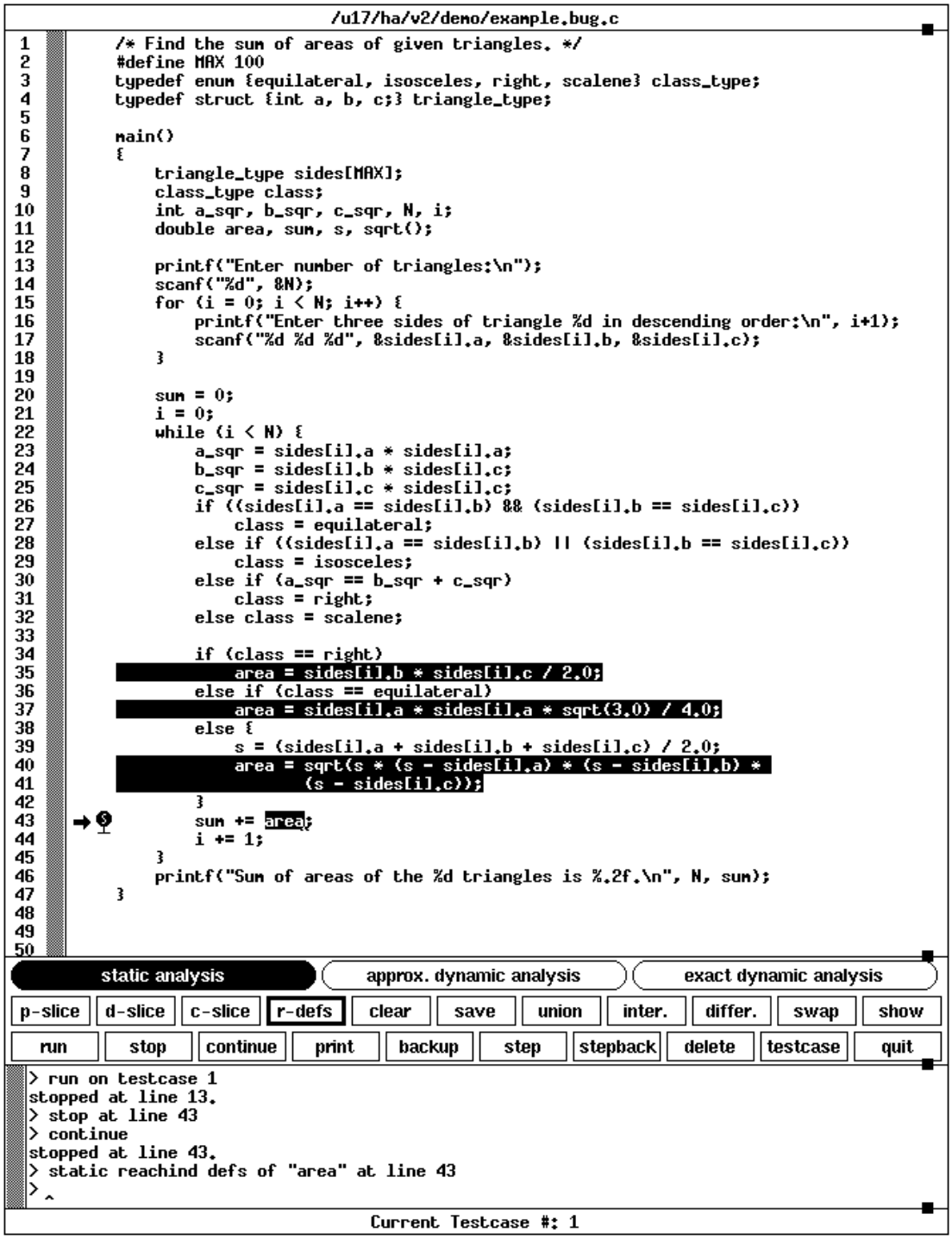

Figure 5.1 Static reaching definitions of area on line 43

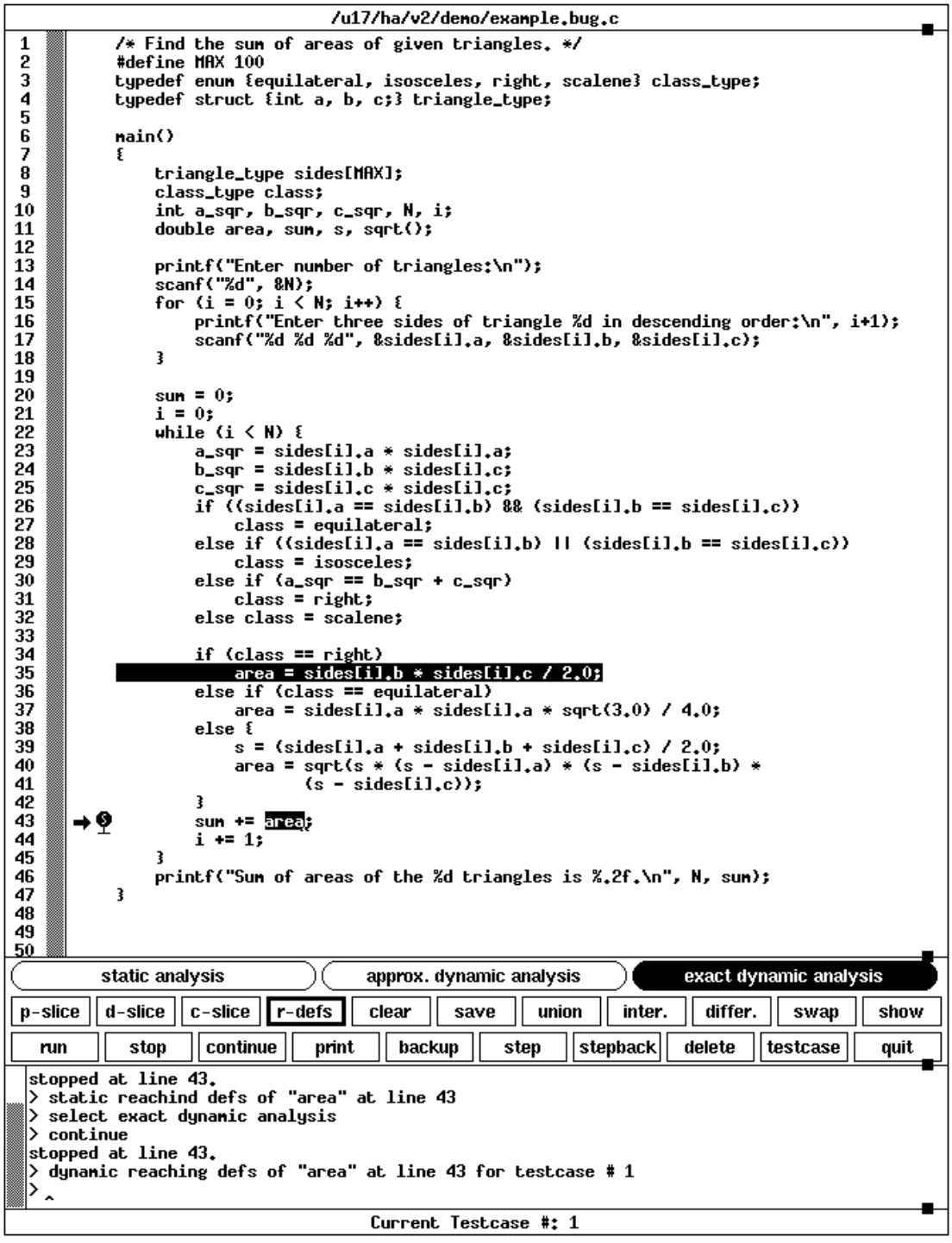

Figure 5.2 Dynamic reaching definition of area on line 43 during the second loop iteration

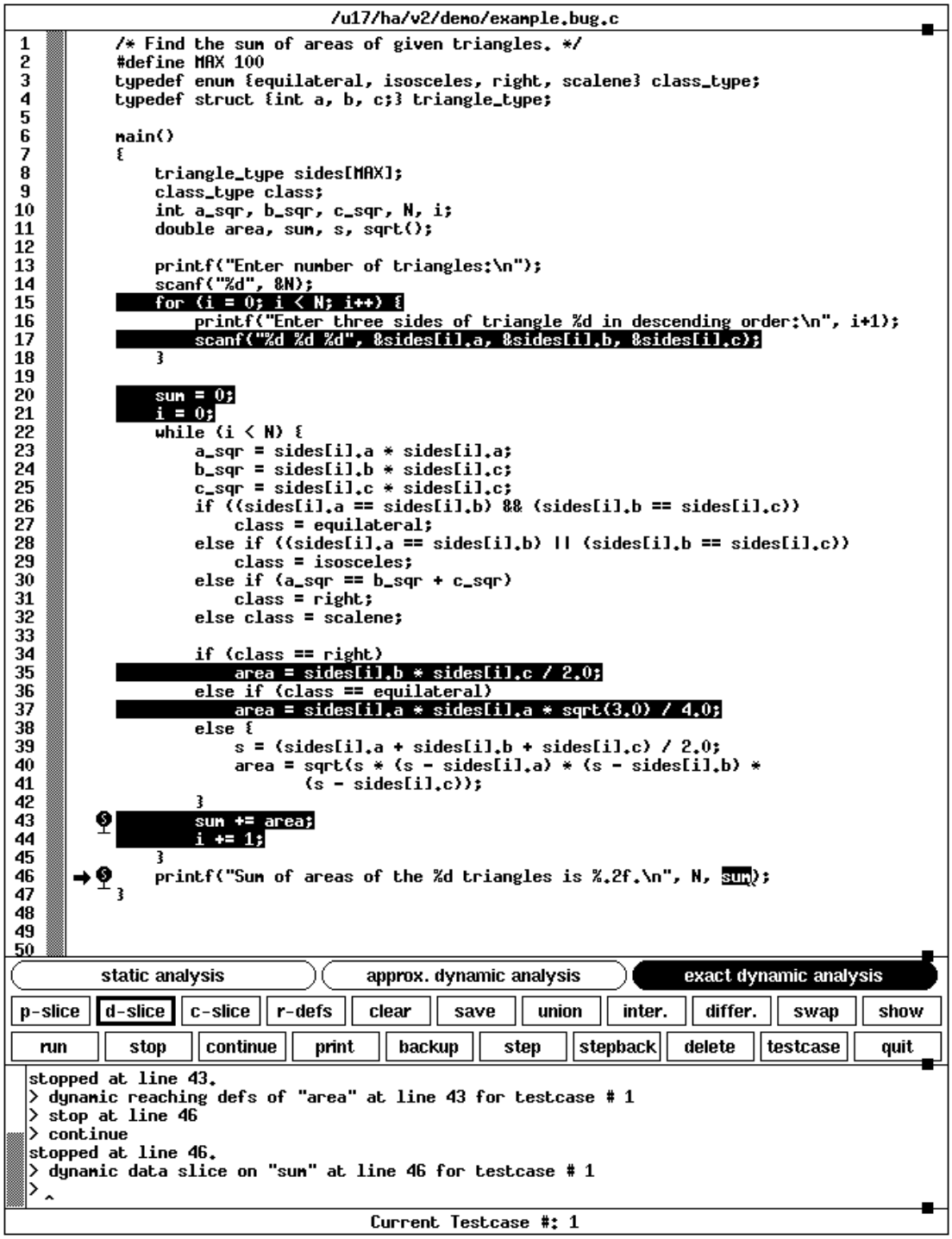

Figure 5.3 Dynamic data slice with respect to sum on line 46 for the testcase #1.

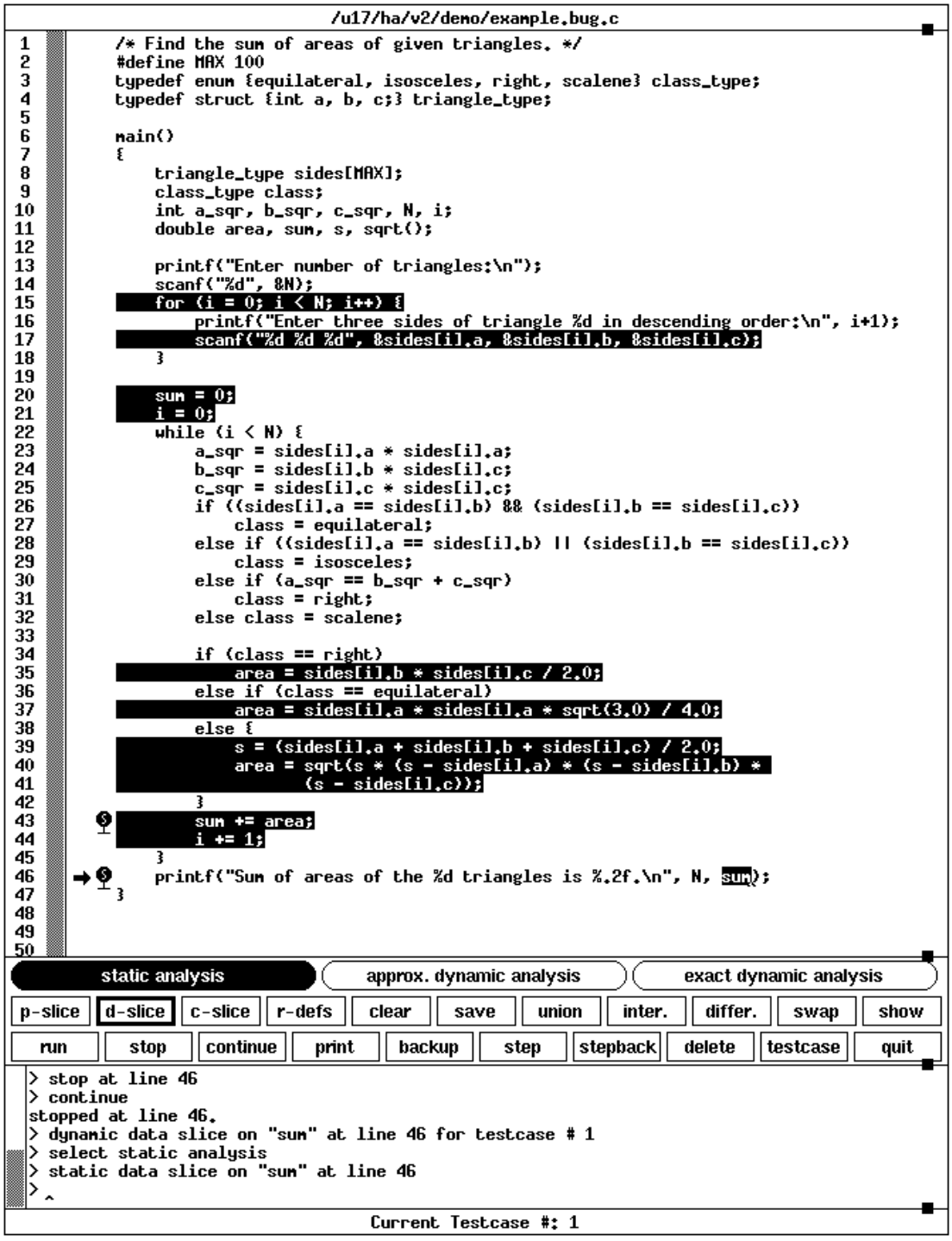

Figure 5.4 Static data slice with respect to sum on line 46.

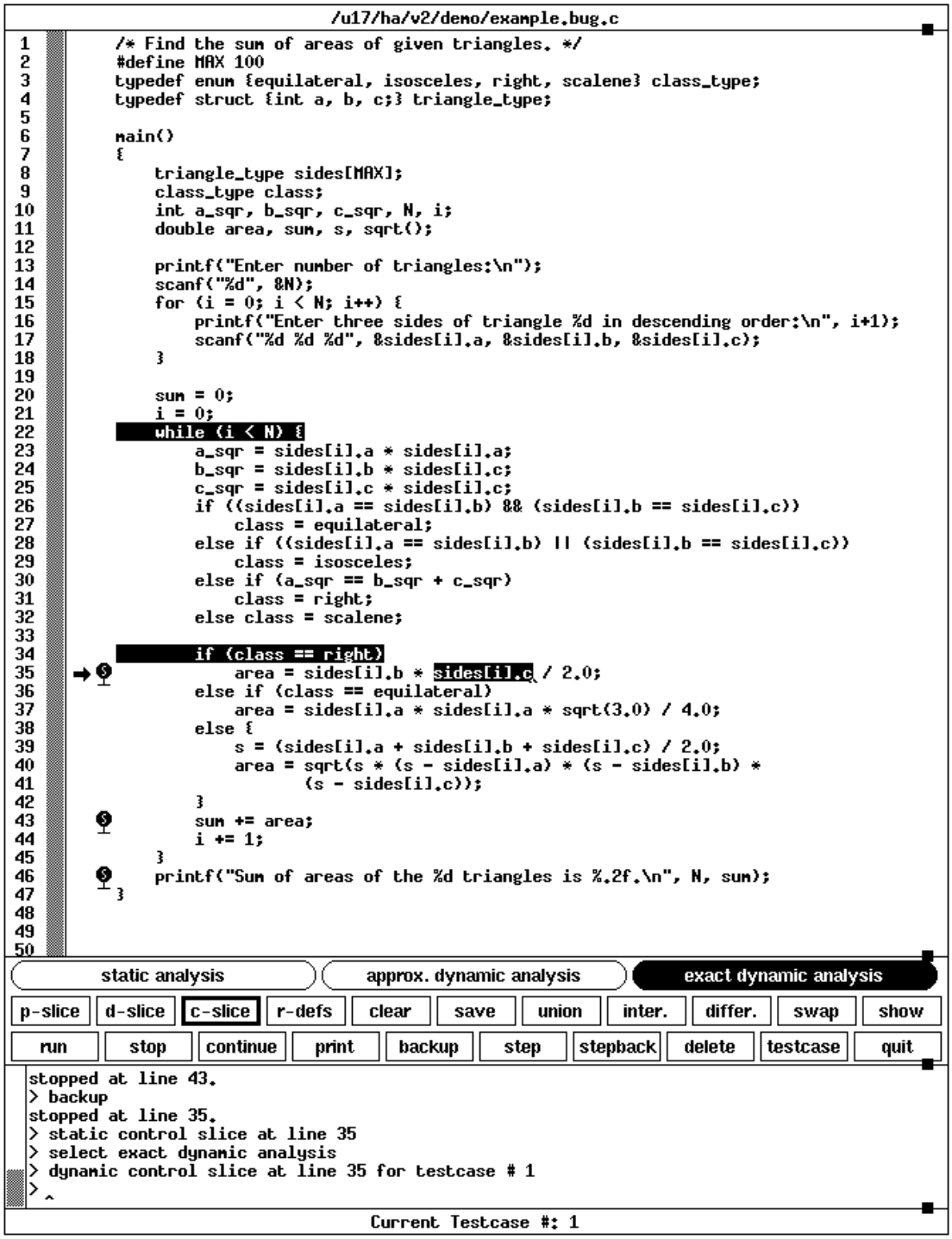

Figure 5.5 Control slice with respect to the statement on line 35.

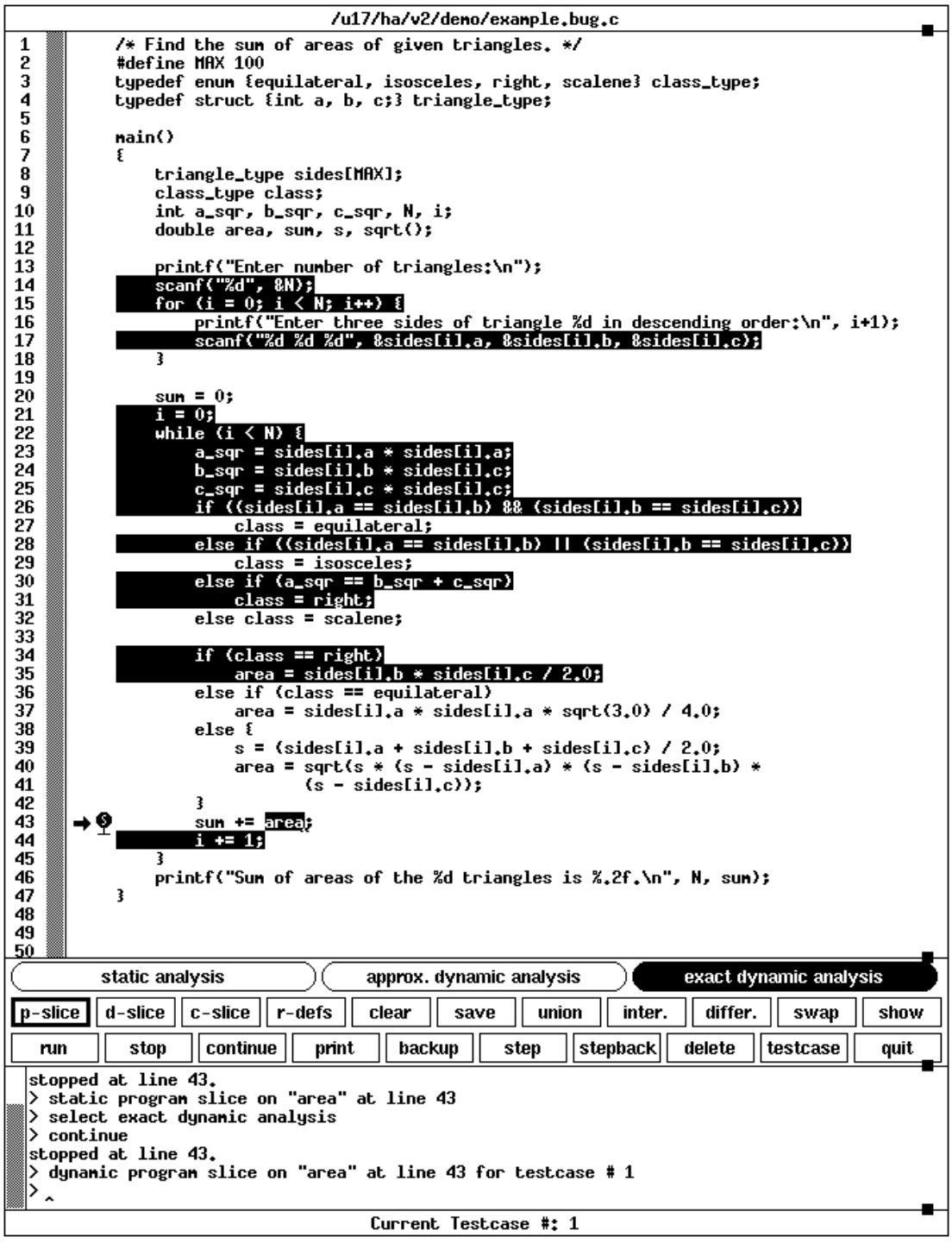

Figure 5.6 Dynamic program slice with respect to area on line 43 during the second loop iteration for the testcase #1.

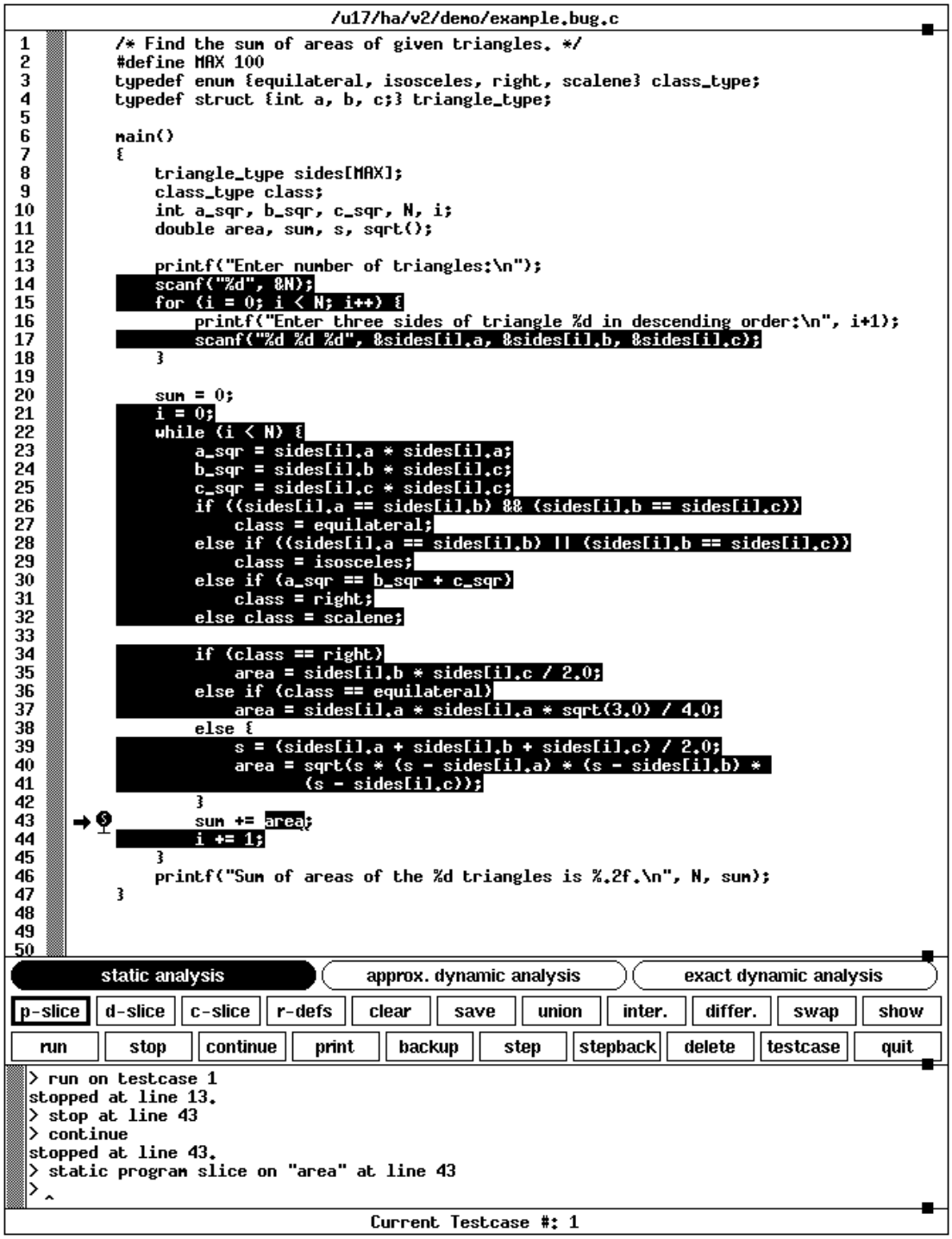

Figure 5.7 Static program slice with respect to variable area on line 43

6. FURTHER FAULT LOCALIZATION

In Chapters 3 and 4 we discussed various techniques to find dynamic program slices. A dynamic slice is helpful in program debugging because it highlights only those statements in the program that are relevant to the erroneous behavior of the program. Sometimes we may be able to identify the fault simply by examining the dynamic slice: we may notice that a statement in the slice computes an incorrect function, or we may find that a statement we expected to be included in the slice is not included there or vice versa. But oftentimes simply examining the slice is not enough; we need to carry out further investigation into the program behavior before we can localize the fault. In this chapter we discuss how we can further narrow down our search for the fault by combining multiple dynamic slices.

6.1 Combining Dynamic Program Slices

In Chapter 2, we mentioned that program dicing [LW87] attempts to reduce the size of the relevant program text to be examined by combining multiple static slices. We can naturally extend that approach to using dynamic slices instead of static slices to get even finer error localization information. But a dynamic slice has four arguments—program, variable, location, and testcase, and instead of varying only the variable argument we can also vary the other arguments—testcase, location, and even the program itself, to generate multiple dynamic slices and use the same approach to combine them. In this section we explore how we can use this observation to derive several heuristics that help further localize the fault.

6.1.1 Varying the Testcase Argument

A testcase, T, for a program, P, is said to be an *error-revealing* testcase if P produces an incorrect output when executed on T [DM91], and it is called a *non-error-revealing* testcase if it produces the correct output.

6.1.1.1 Difference of Dynamic Slices

Many times we have a situation where a program works correctly on one testcase but fails on another. In other words, we have an error-revealing testcase, T_e, and a non-error-revealing testcase, T_n. One way to proceed in this situation would be to ask the following question: What does this program do differently under T_e that it does not do under T_n? That is, we should examine the *difference* in the program behavior under T_e and T_n. One way to find this difference is to see how the two dynamic program slices with respect to T_e and T_n differ. If the dynamic slice of a program, P, with respect to a faulty variable, *var*, at a program location, *loc*, for an error-revealing testcase, T_e, is denoted by $DPS(P, var, loc, T_e)$, and that with respect to a non-error-revealing testcase, T_n, is denoted by $DPS(P, var, loc, T_n)$, then one heuristic would be to examine the set of statements given by:

$$DPS(P, var, loc, T_e) - DPS(P, var, loc, T_n)$$

For simplicity, we also write this as:

$$DPS(\cdot, \cdot, \cdot, T_e) - DPS(\cdot, \cdot, \cdot, T_n)$$

where omitting the first three arguments to *DPS* means their values are fixed; only the fourth argument varies.

For example, consider again the program in Figure 3.17. Figure 6.1 shows its dynamic program slice with respect to sum on line 40 for testcase #1 (recall that for **testcase #1** $N = 2$ and the lengths of sides of the two triangles are (3, 3, 3) and (6, 5, 4), respectively). The program incorrectly evaluates the value of sum for this testcase, so this is an error-revealing testcase. But when the same program is executed

for another testcase where $N = 2$, and the lengths of the sides of the two triangles are (4, 4, 4) and (5, 3, 3), respectively, the value of **sum** output on line 40 is correct. We will refer to this testcase as **testcase #2**. So testcase #2 is a non-error-revealing testcase. Figure 6.2 shows the dynamic program slice with respect to **sum** on line 40 for testcase #2. Figure 6.3 shows the difference between the two dynamic slices, or, $DPS(P$, **sum**, line 40, testcase #1) $-$ $DPS(P$, **sum**, line 40, testcase #2). Examining the difference, in this case, should lead to faster localization of the fault on line 19 compared to examining the complete dynamic program slice shown in Figure 6.1.

Collofello and Cousins [CC87b] have proposed a similar approach where they find basic-blocks (referred to as "decision-to-decision-paths" by them) in the program that control reaches when the program is executed for an error-revealing testcase but that are not reached when the program is executed for any non-error-revealing testcase. But as a basic-block may also contain statements totally unrelated to the fault, this approach yields much coarser error localization information compared to ours.

Sometimes after we subtract a dynamic slice with respect to a non-error-revealing testcase from that with respect to an error-revealing testcase, we may be left with an empty set. At other times, even if the set is non-empty, we may not find any fault with statements in the difference set. These cases are generally indicative of one of the following situations:

- A predicate in the program has a fault which causes some statements to be incorrectly included in the slice with respect to the error-revealing testcase, but it causes the same statements to be correctly included in the slice with respect to the non-error-revealing testcase. Thus the extraneous statements in the former slice do not get included in the difference set.

- A fault in a loop predicate is causing the loop body to execute an incorrect number of times in the case of the error-revealing testcase, but it causes the

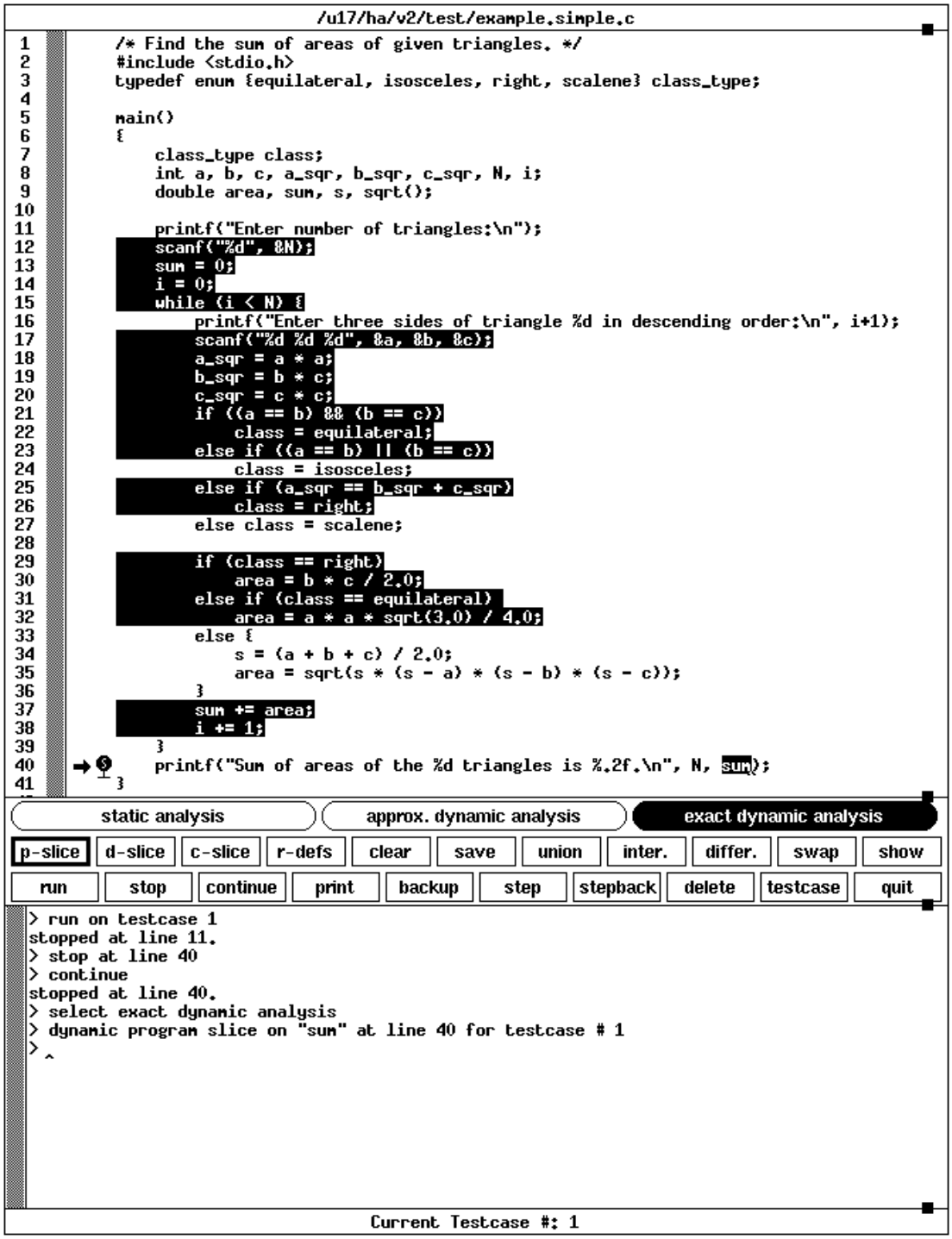

Figure 6.1 Dynamic program slice of sum on line 40 for testcase #1.

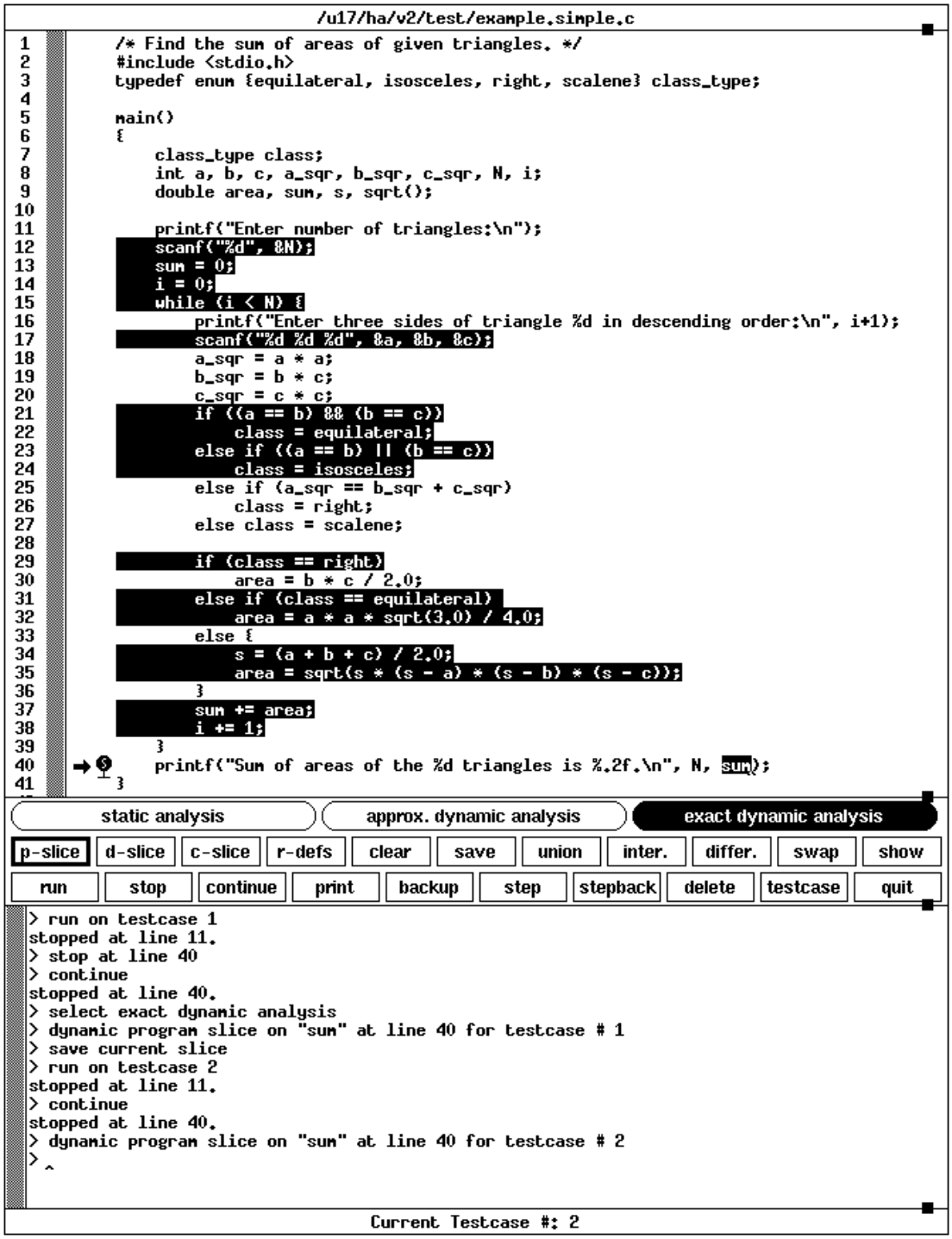

Figure 6.2 Dynamic program slice of sum on line 40 for testcase #2.

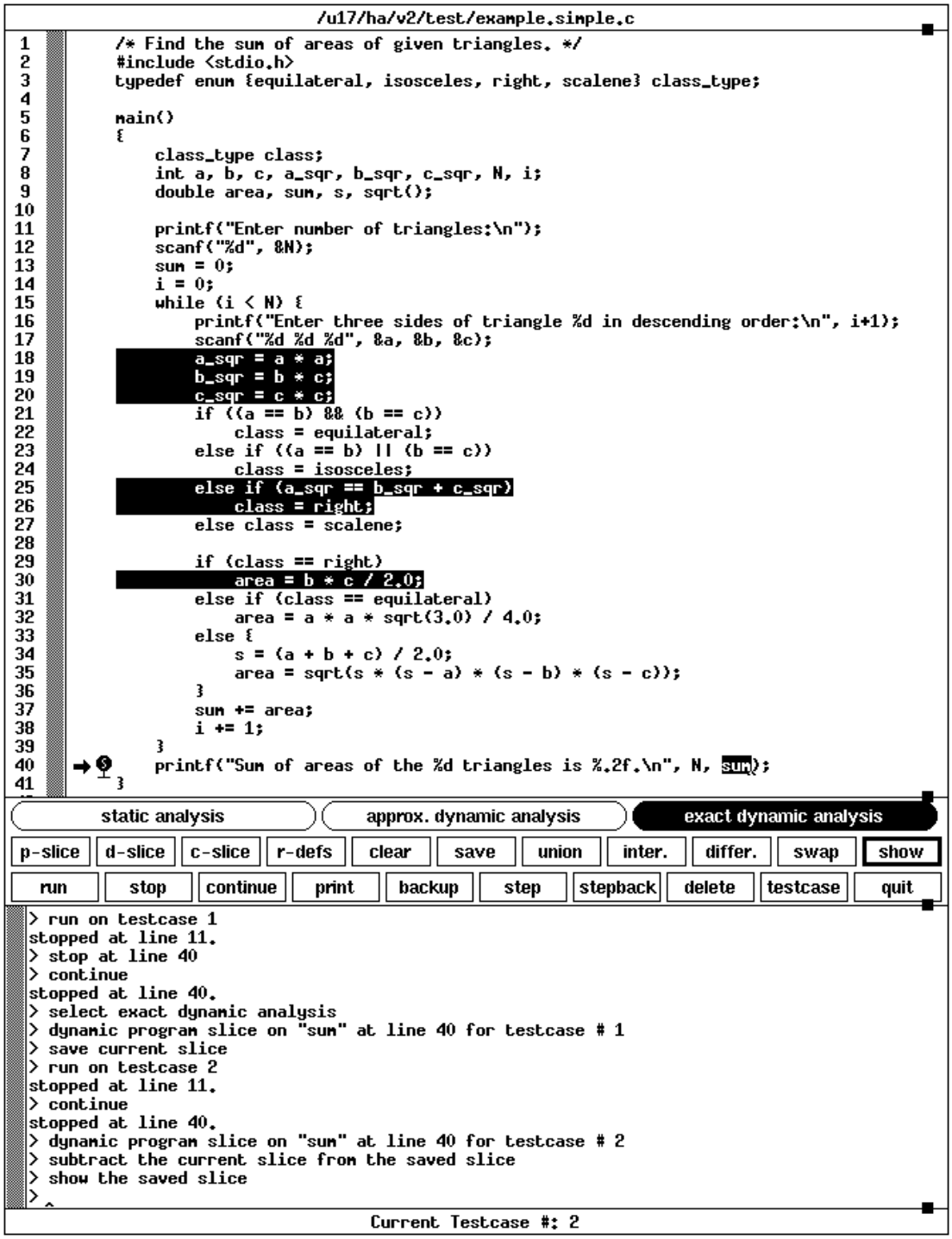

Figure 6.3 Dynamic program slice for testcase #1 minus that for testcase #2.

same loop body to execute the correct number of times in the case of the non-error-revealing testcase. As the faulty loop predicate belongs to both slices it is not included in their difference set.

- A faulty assignment statement, which affects the value of the variable in question, is coincidentally computing the correct value in the case of the non-error-revealing testcase, but it is unable to shield the fault in the case of the error-revealing testcase. As the faulty assignment belongs to both slices, it does not belong to their difference set.

All three cases above are instances of coincidental correctness—the first two involving coincidentally correct evaluation of a faulty predicate and the third involving that of a faulty assignment—when the program is executed on the non-error revealing testcase. In such situations we may be able to find other non-error-revealing testcases that do not cause the above problem. For some faults, however, it may be impossible to find a non-error-revealing testcase that does not cause coincidental correctness. Oftentimes the fault may be such that the program fails on every testcase we try. For such faults we may not be able to find a non-error-revealing testcase at all. In these situations we may not be able to apply the strategy outlined above, but we may still be able to localize the fault using some of the other strategies discussed below.

The success of the above strategy depends on judicious selection of the non-error-revealing testcase T_n. For this strategy to work, we should try to find a non-error-revealing testcase that is "similar" to T_e. In the example above, T_n was similar to T_e in that both involved an equilateral triangle. If, on the other hand, the non-error-revealing testcase has nothing in common with the error-revealing testcase, examining the difference between the corresponding slices may not be meaningful at all. examined. For example, consider another non-error-revealing testcase, **testcase #3**, where $N = 1$, and the sides of the triangle are (5, 3, 3). If we subtracted the dynamic slice of sum on line 40 for testcase #3 from that for testcase #1, we will get very

little reduction in the relevant code to be examined. Figure 6.4 shows the resulting set after performing this subtraction.

6.1.1.2 Union of Dynamic Slices

In the strategy outlined above, we subtracted the dynamic slice with respect to *one* non-error-revealing testcase from that with respect to an error-revealing testcase. Very often we have a situation where the program under test fails on a specific testcase but works correctly on many other testcases. That is, we have only one error-revealing testcase, T_e, but several non-error-revealing testcases, $T_{n_1}, T_{n_2}, \ldots, T_{n_q}$. In this case, we may be able to get a further reduction in the relevant code to be examined if we subtracted the dynamic slices with respect to all non-error-revealing testcases from the dynamic slice with respect to the single error-revealing testcase. That is, we should examine:

$$DPS(\cdot, \cdot, \cdot, T_e) - \bigcup_{i=1,2,\ldots,q} DPS(\cdot, \cdot, \cdot, T_{n_i})$$

For example, for the same program mentioned above, consider an error-revealing testcase, **testcase #4**, where $N = 3$ and the lengths of sides of the three triangles are (5, 5, 5), (6, 5, 4), and (3, 2, 2), respectively (we now have a total of four testcases for this program: two error-revealing testcases, viz. #1 and #4, and two non-error-revealing testcases, viz. #2 and #3). Figure 6.5 shows the set resulting from subtraction of the dynamic slice with respect to the non-error-revealing testcase #3 from that with respect to the error-revealing testcase #4. If, instead, we subtract dynamic slices with respect to both non-error-revealing testcases #2 and #3 we end up with a much smaller set of statements to examine, as shown in Figure 6.6.

6.1.1.3 Intersection of Dynamic Slices

If a non-error-revealing testcase is hard to find for an erroneous program, it should be relatively easy to find another error-revealing testcase for the program. If we have two error-revealing testcases, T_{e_1} and T_{e_2}, another debugging strategy would be to

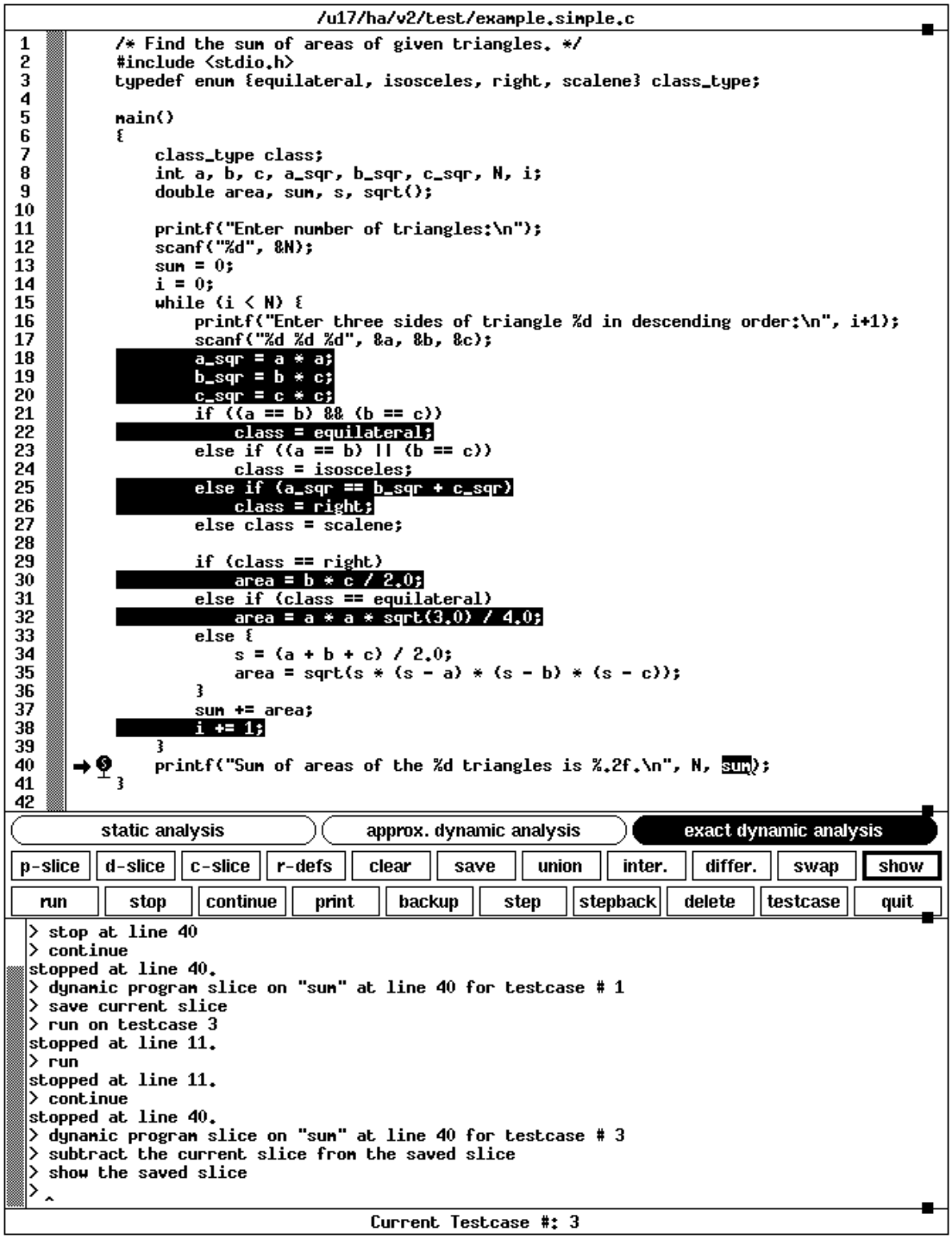

Figure 6.4 Dynamic program slice for testcase #1 minus that for testcase #3.

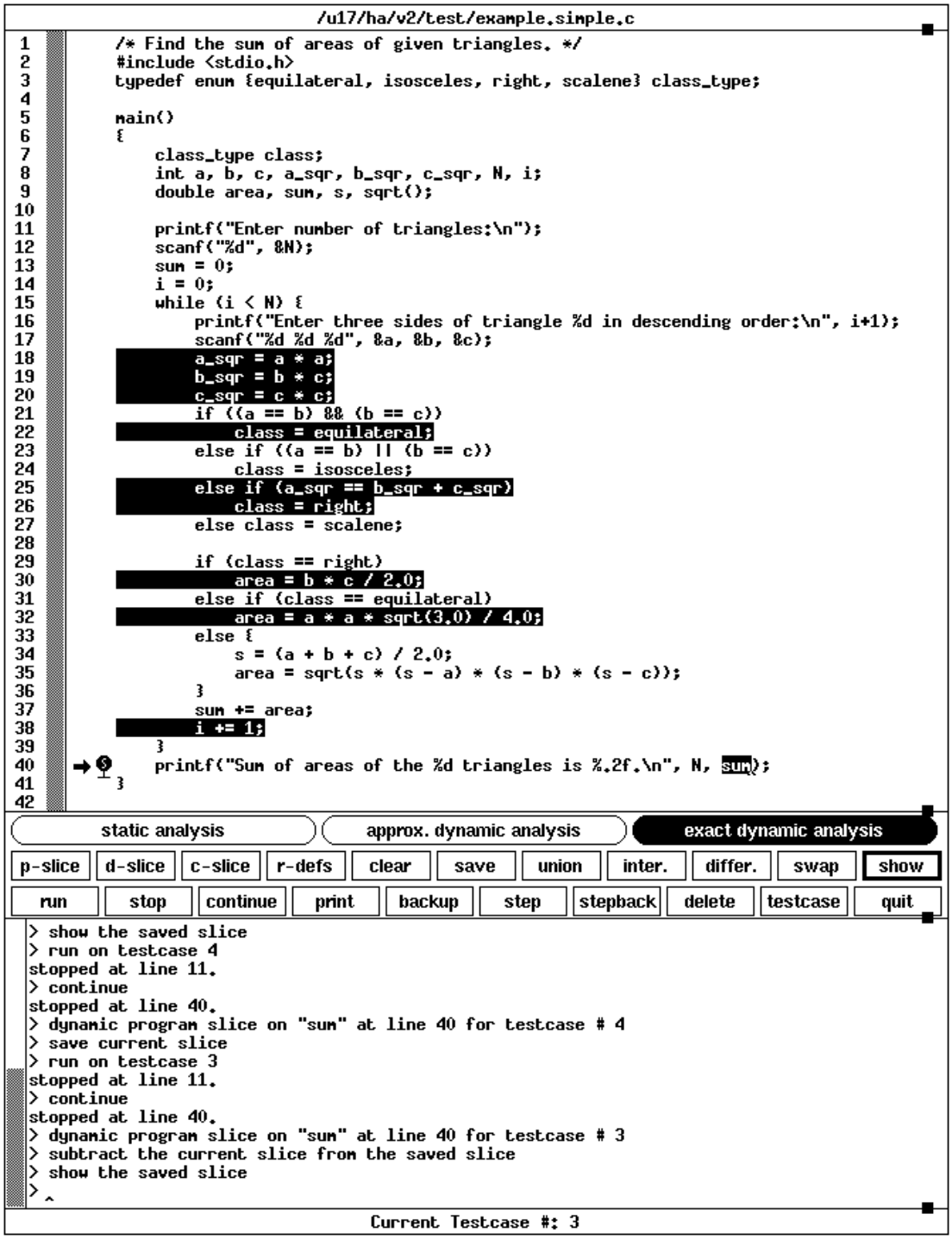

Figure 6.5 Dynamic program slice for testcase #4 minus that for testcase #3.

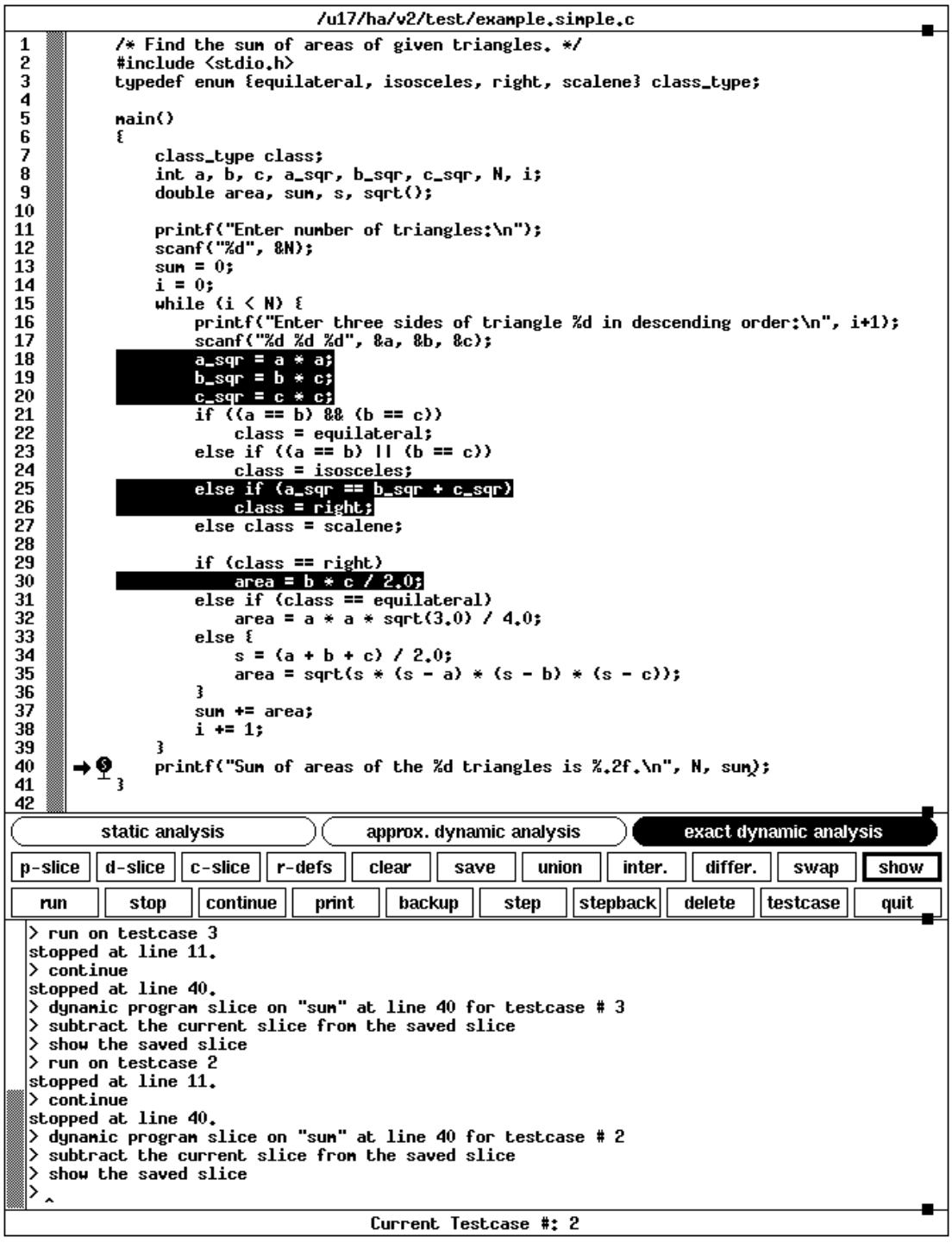

Figure 6.6 Dynamic program slice for testcase #4 minus those for testcases #2 and #3.

examine the intersection of the dynamic slices with respect to the two testcases. That is, we should examine:

$$DPS(\cdot, \cdot, \cdot, T_{e_1}) \cap DPS(\cdot, \cdot, \cdot, T_{e_2})$$

If we have several error-revealing testcases, $T_{e_1}, T_{e_2}, \ldots, T_{e_p}$, we should check the intersection of dynamic slices with respect to each of these testcases. That is, we should examine:

$$\bigcap_{i=1,2,\ldots,p} DPS(\cdot, \cdot, \cdot, T_{e_i})$$

We may try the above intersection strategy even if we are able to find non-error-revealing testcases for the program. For example, consider another error-revealing testcase, **testcase #5**, where $N = 2$ and the lengths of the sides of the two triangles are (6, 5, 4) and (5, 3, 3), respectively, for the same program mentioned above. Figure 6.7 shows the intersection of the dynamic slices with respect to sum on line 40 for testcases #1 and #5. The resulting set, in this case, should lead to relatively faster localization of the fault as it is smaller than each dynamic slice with respect to the two testcases.

But the intersection strategy may not always lead to reduction in the size of the relevant code to be examined, depending on the nature of the fault. If we have both a set of error revealing testcases and a set of non-error-revealing testcases, we may first obtain the intersection of dynamic slices with respect to all error-revealing testcases and then subtract the dynamic slice with respect to each non-error-revealing testcase from the intersection. That is, if we have p error-revealing testcases, $T_{e_1}, T_{e_2}, \ldots, T_{e_p}$, and q non-error-revealing testcases, $T_{n_1}, T_{n_2}, \ldots, T_{n_q}$, we may examine:

$$\bigcap_{i=1,2,\ldots,p} DPS(\cdot, \cdot, \cdot, T_{e_i}) \quad - \bigcup_{i=1,2,\ldots,q} DPS(\cdot, \cdot, \cdot, T_{n_i})$$

For example, Figure 6.8 shows the result of subtracting dynamic slices of sum on line 40 for non-error-revealing testcases #2 and #3 from the intersection shown in Figure 6.7.

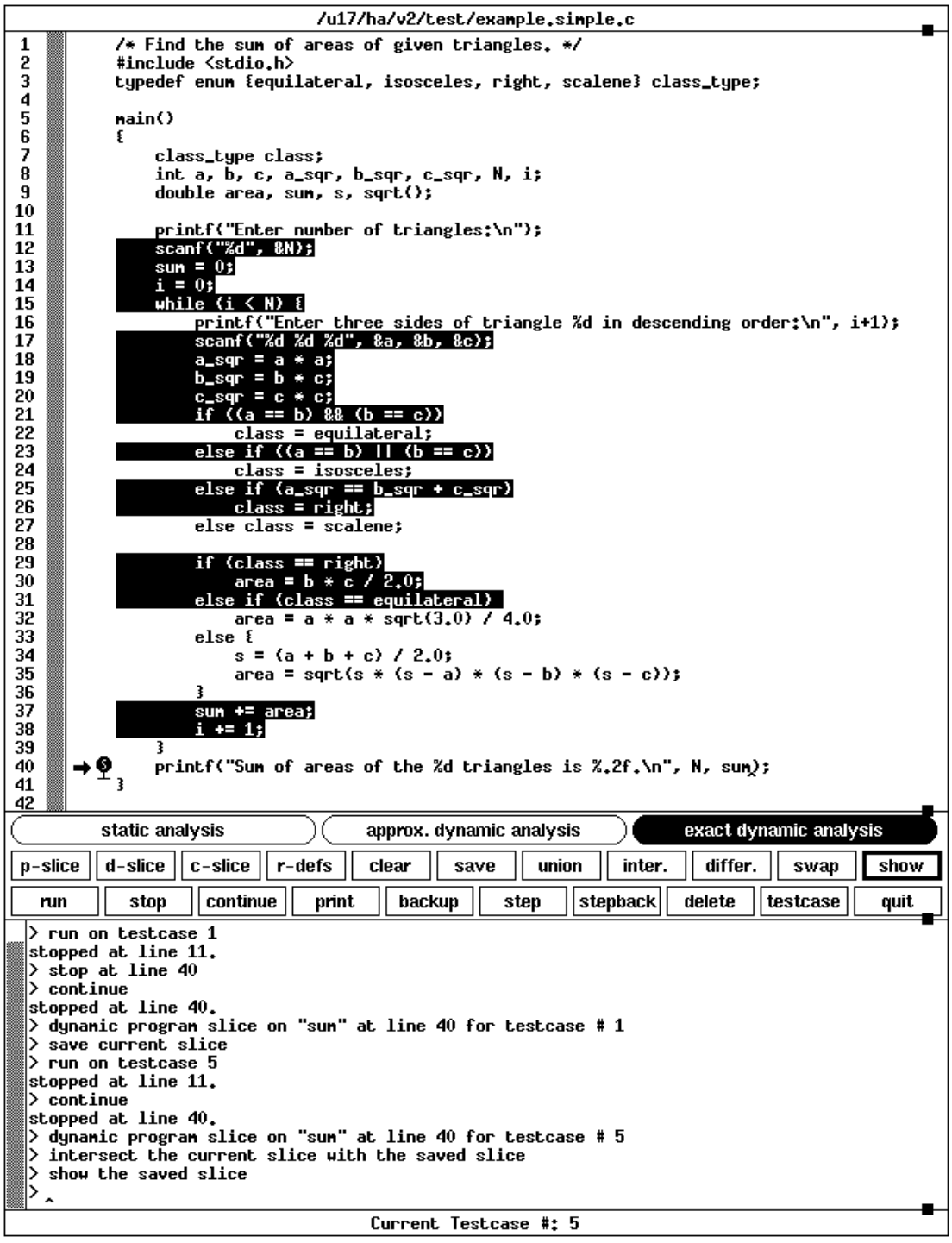

Figure 6.7 Intersection of dynamic program slices of sum on line 40 for testcases #1 and #5.

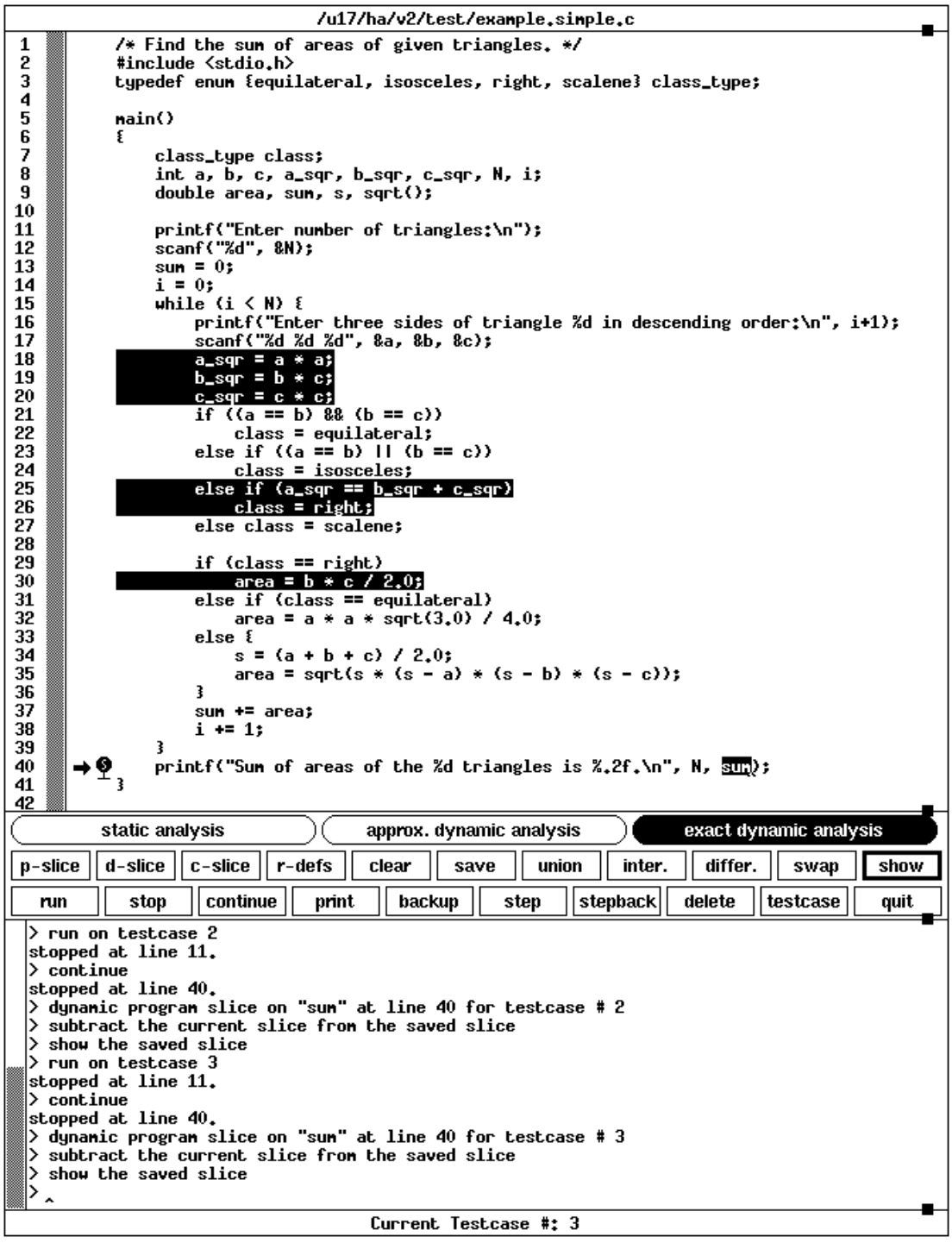

Figure 6.8 Result of subtracting dynamic program slices of sum on line 40 for testcases #2 and #3 from the intersection of the corresponding slices for testcases #1 and #5.

6.1.2 Varying the Variable Argument

So far, while combining two or more dynamic slices, we always fixed the first three arguments of a dynamic slice and varied only the fourth argument. Instead, we could also fix the last three arguments and vary the first. That is, for the same testcase and the same program location, we may find that the value of one variable is incorrect but that of another is correct. In this case, another debugging strategy would be to subtract the dynamic slice with respect to the latter from that with respect to the former and examine the difference. In other words, we should examine:

$$DPS(\cdot, var_e, \cdot, \cdot) - DPS(\cdot, var_n, \cdot, \cdot)$$

where var_e denotes a variable whose value is observed to be erroneous and var_n denotes a variable whose value in not erroneous. This strategy is the dynamic analogue of program dicing strategy proposed by Lyle and Weiser [LW87]

For example, consider the program in Figure 6.9. It reads a date (month, day, year) and finds the corresponding day-of-the-year and day-of-the-week. This program has a fault on line 68 where the right-hand-side of the assignment performs an addition instead of a subtraction. Consider the testcase when the date entered is (month=3, day=1, year=1991). The program incorrectly computes day-of-the-week for this date as a Monday instead of a Friday. But it correctly computes day-of-the-year as 60. Figure 6.9 shows the dynamic slice with respect to date.day_of_the_week on line 80 for this testcase and Figure 6.10 shows the dynamic program slice for date.day_of_the_year at the same location for the same testcase. Figure 6.11 shows the result of subtracting the latter from the former. Clearly, this difference should lead to faster localization of the fault on line 68 compared to the dynamic slice shown in Figure 6.9.

Just as we generalized the strategies discussed in Section 6.1.1 from considering only two testcases to multiple testcases, we can also generalize the above strategy from considering only two variables to multiple variables. That is, instead of subtracting the dynamic slice with respect to one correct variable from that of an incorrect one, we may subtract dynamic slices of several correct variables from the dynamic slice

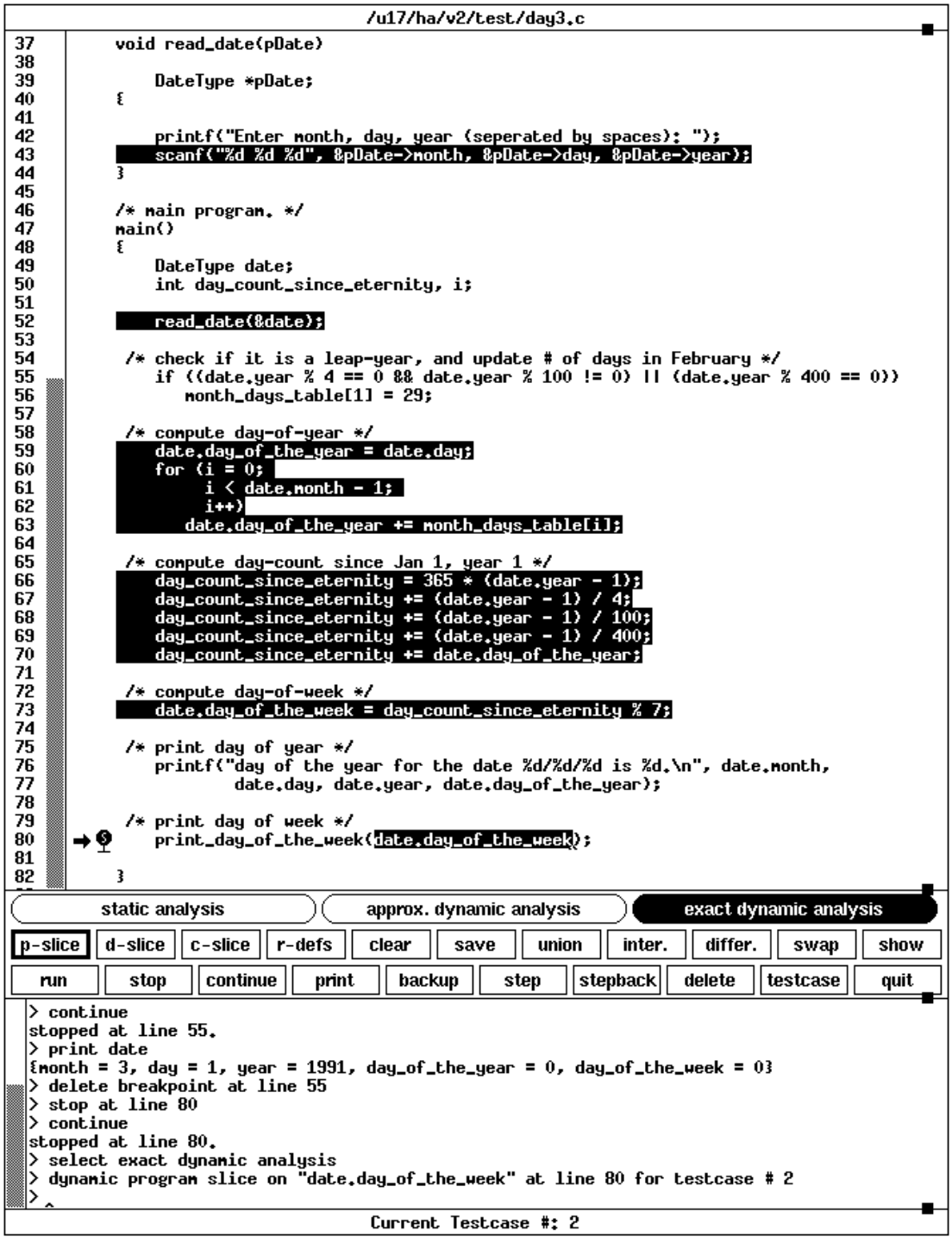

Figure 6.9 Dynamic program slice of date.day_of_the_week on line 80 for testcase (month=3, day=1, year=1991)

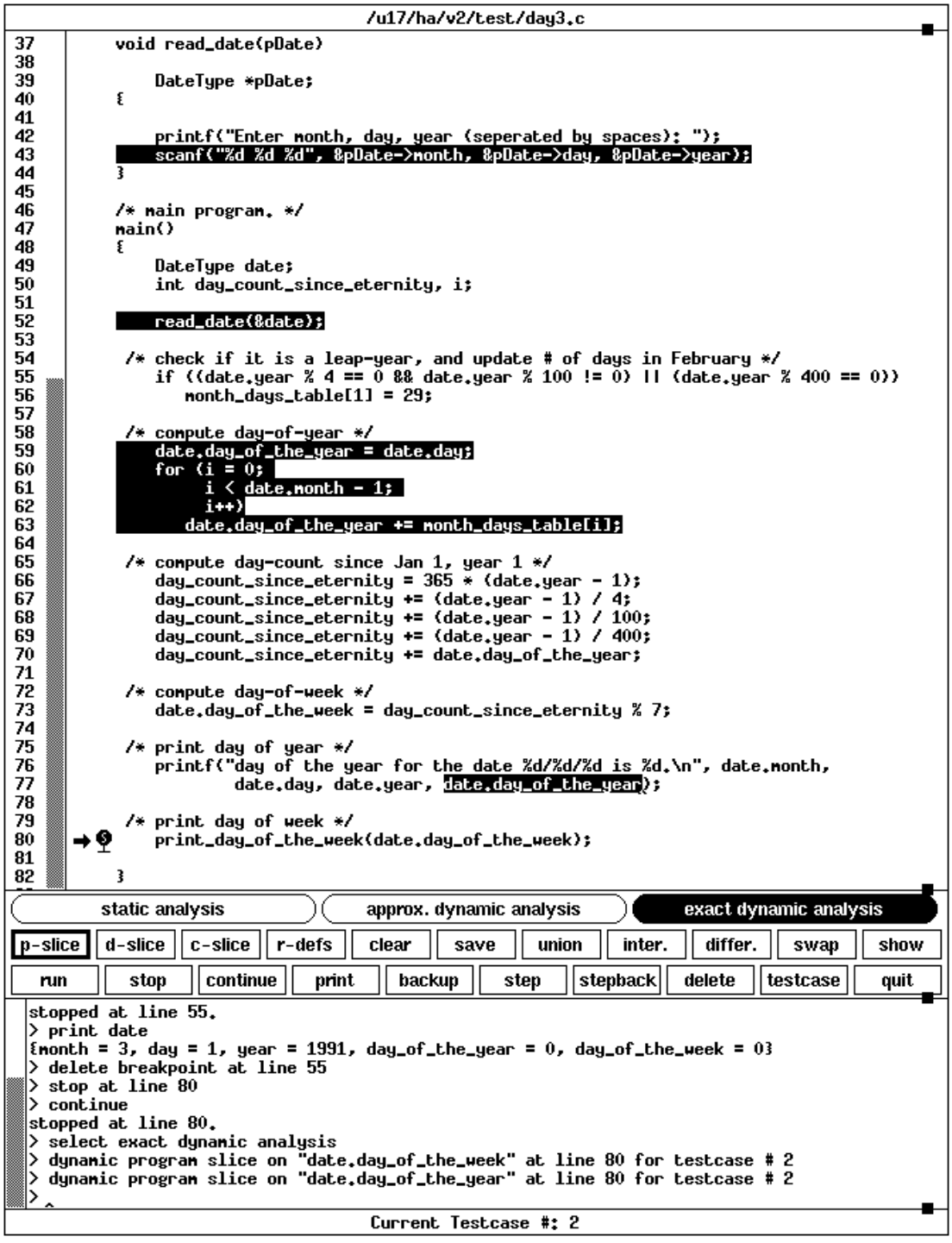

Figure 6.10 Dynamic program slice of date.day_of_the_year on line 80 for testcase (month=3, day=1, year=1991)

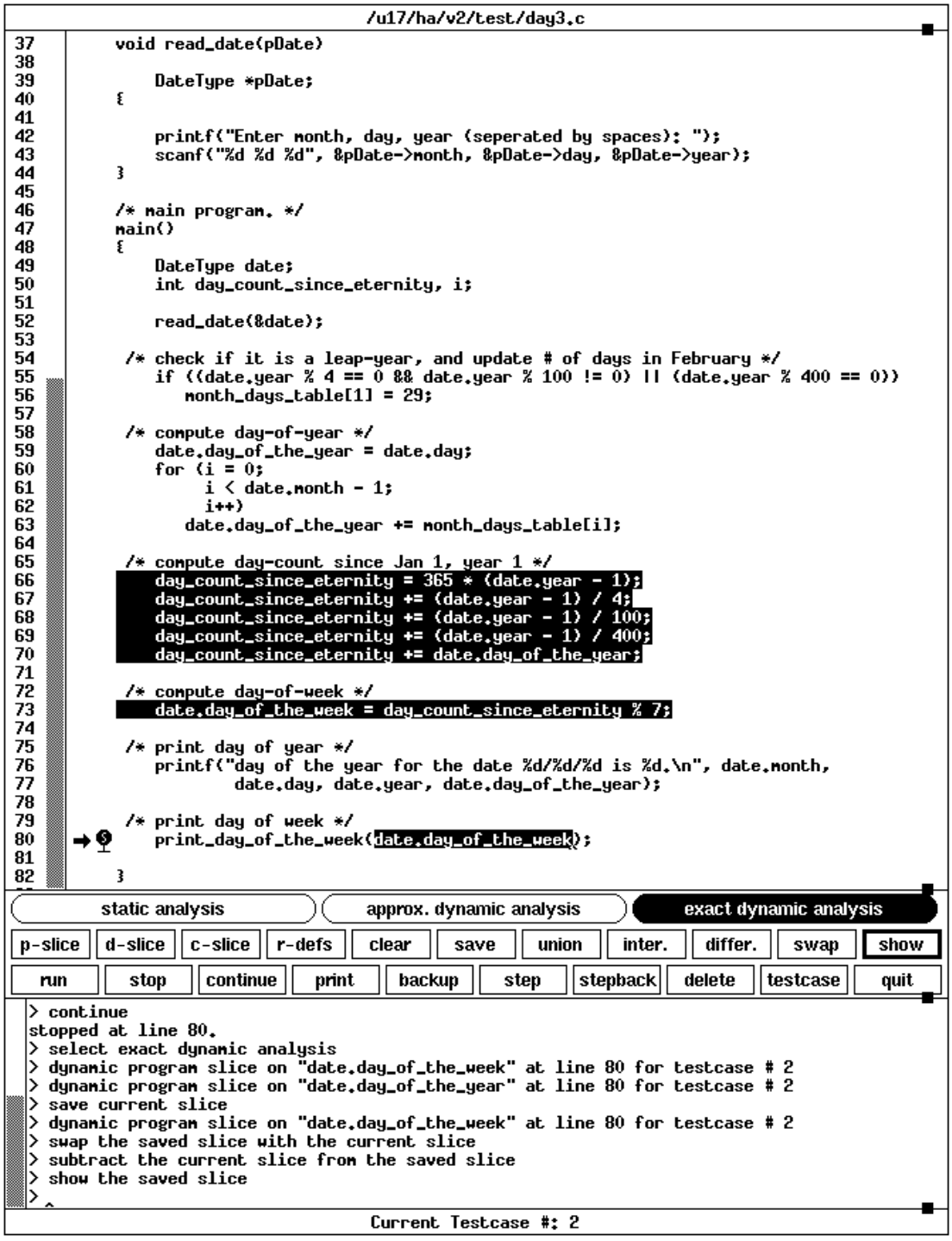

Figure 6.11 Result of subtracting the dynamic program slice of date.day_of_the_year on line 80 for testcase (month=3, day=1, year=1991) from the corresponding slice of date.day_of_the_week.

of the incorrect variable. Similarly, if we observe that values of several variables are incorrect, we may check the intersection of dynamic slices with respect to each of these variables.

6.1.3 Varying the Location Argument

Sometimes while debugging a program, we may observe that the value of a variable is correct at one instance but incorrect at another. For example, we may find that a variable being modified inside a loop has a correct value at the end of one iteration but the same variable assumes an incorrect value at the end of the next iteration. In this situation, we may want to examine the difference in the dynamic slices with respect to the two instances of the same variable for the same testcase. That is, if the instance when the variable in question assumes an erroneous value is denoted by loc_e and that when it assumes a correct value is denoted by loc_n, another strategy would be to examine:

$$DPS(\cdot, \cdot, loc_e, \cdot) - DPS(\cdot, \cdot, loc_n, \cdot)$$

For example Figure 6.12 shows the dynamic slice with respect to area on line 37 during the first loop iteration for testcase #1, and Figure 6.13 shows the corresponding slice during the second iteration. For this testcase, area is computed correctly during the first iteration but incorrectly during the second. Figure 6.14 shows the result after the slice in Figure 6.12 is subtracted from that in Figure 6.13.

As was done in Sections 6.1.1 and 6.1.2, we can also generalize the above strategy to subtracting dynamic slices of multiple correct instances of a variable for a given testcase from that of an incorrect instance of the same variable for the same testcase. Similarly it can also be generalized to taking intersections of the dynamic slices with respect to multiple incorrect instances of the same variable for the same testcase.

6.1.4 Varying the Program Argument

Oftentimes during program development, we encounter the following situation: We have a correct program, P_n, which when executed on a testcase, T, computes the

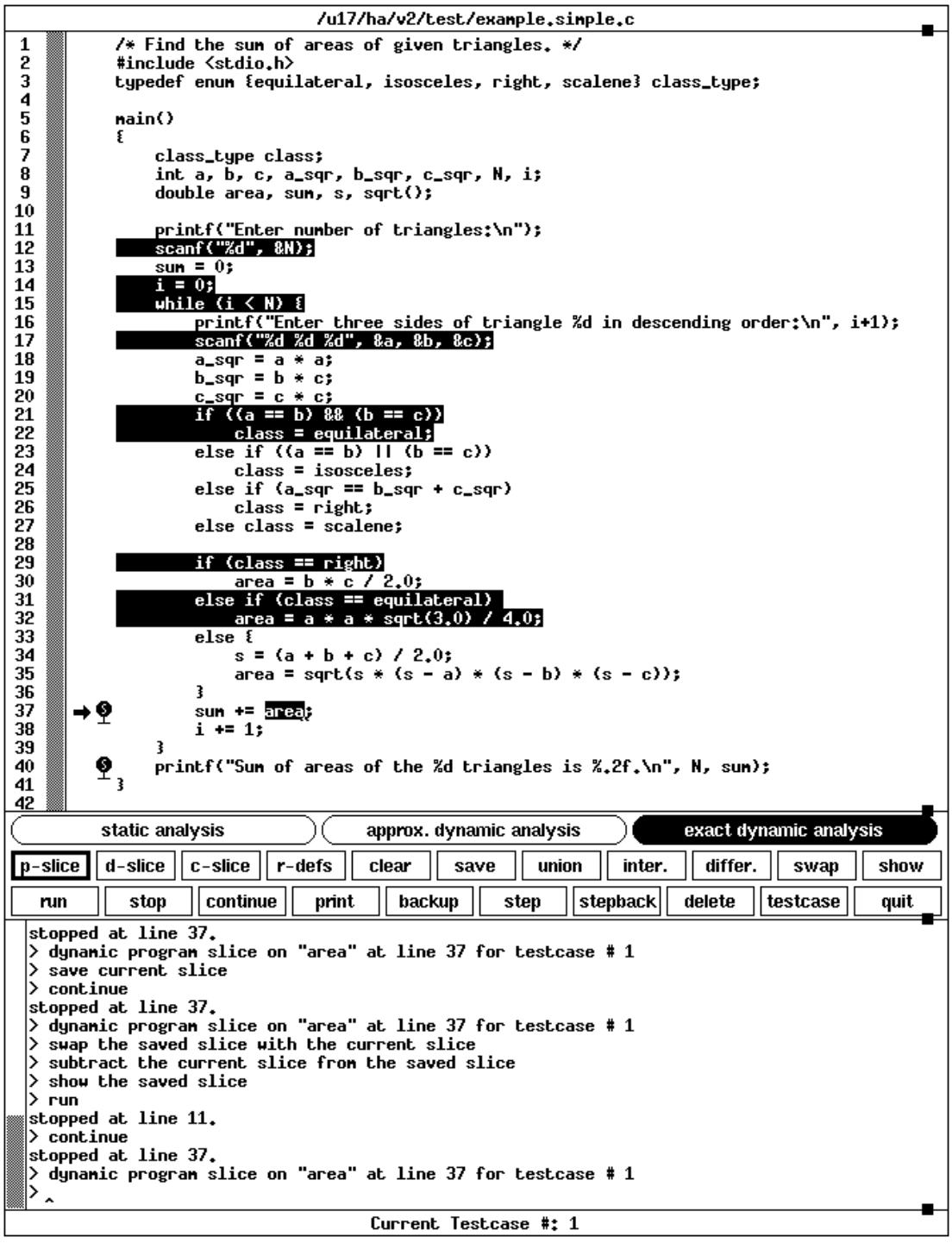

Figure 6.12 Dynamic program slice of area on line 37 during the first loop iteration for testcase #1.

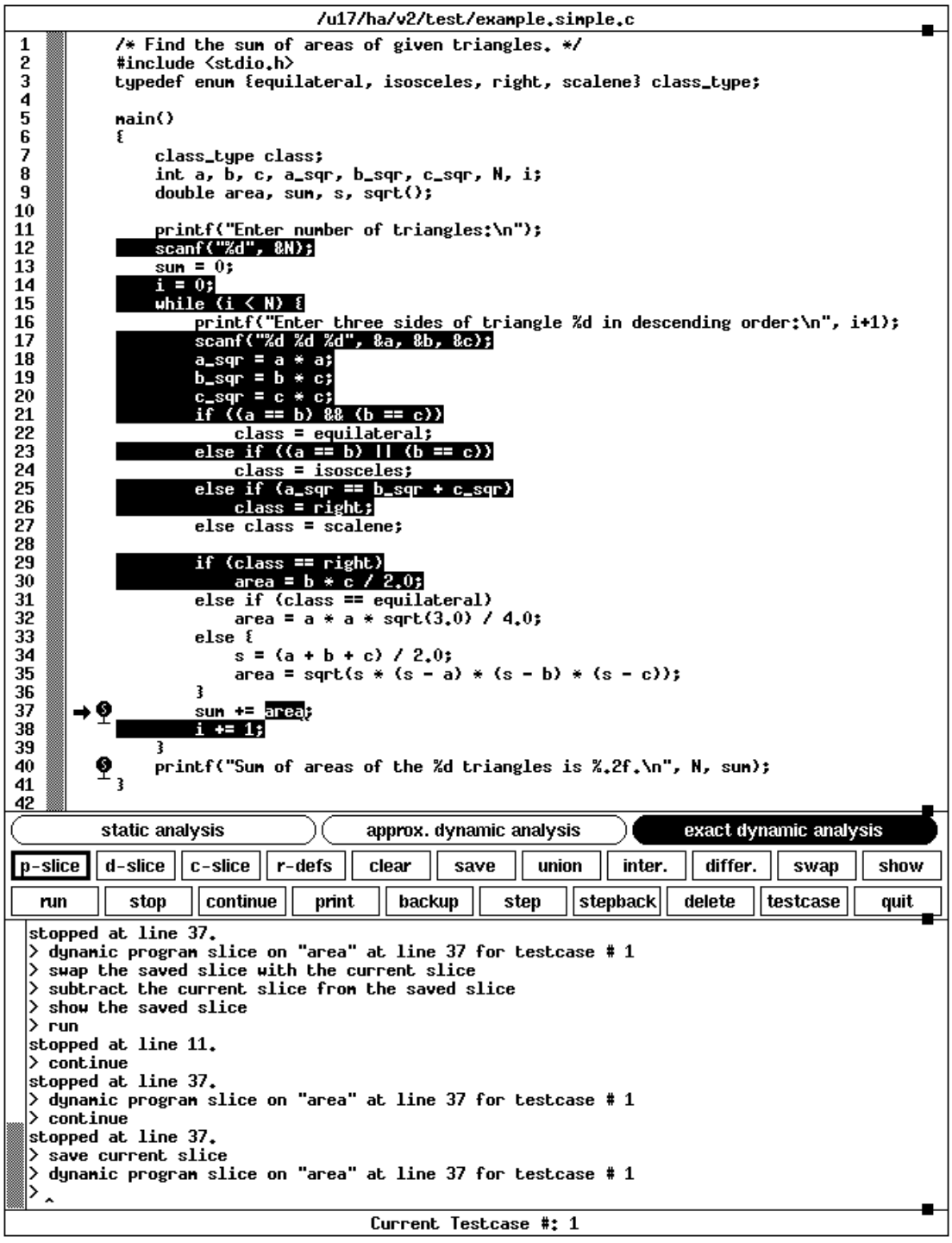

Figure 6.13 Dynamic program slice of area on line 37 during the second loop iteration for testcase #1.

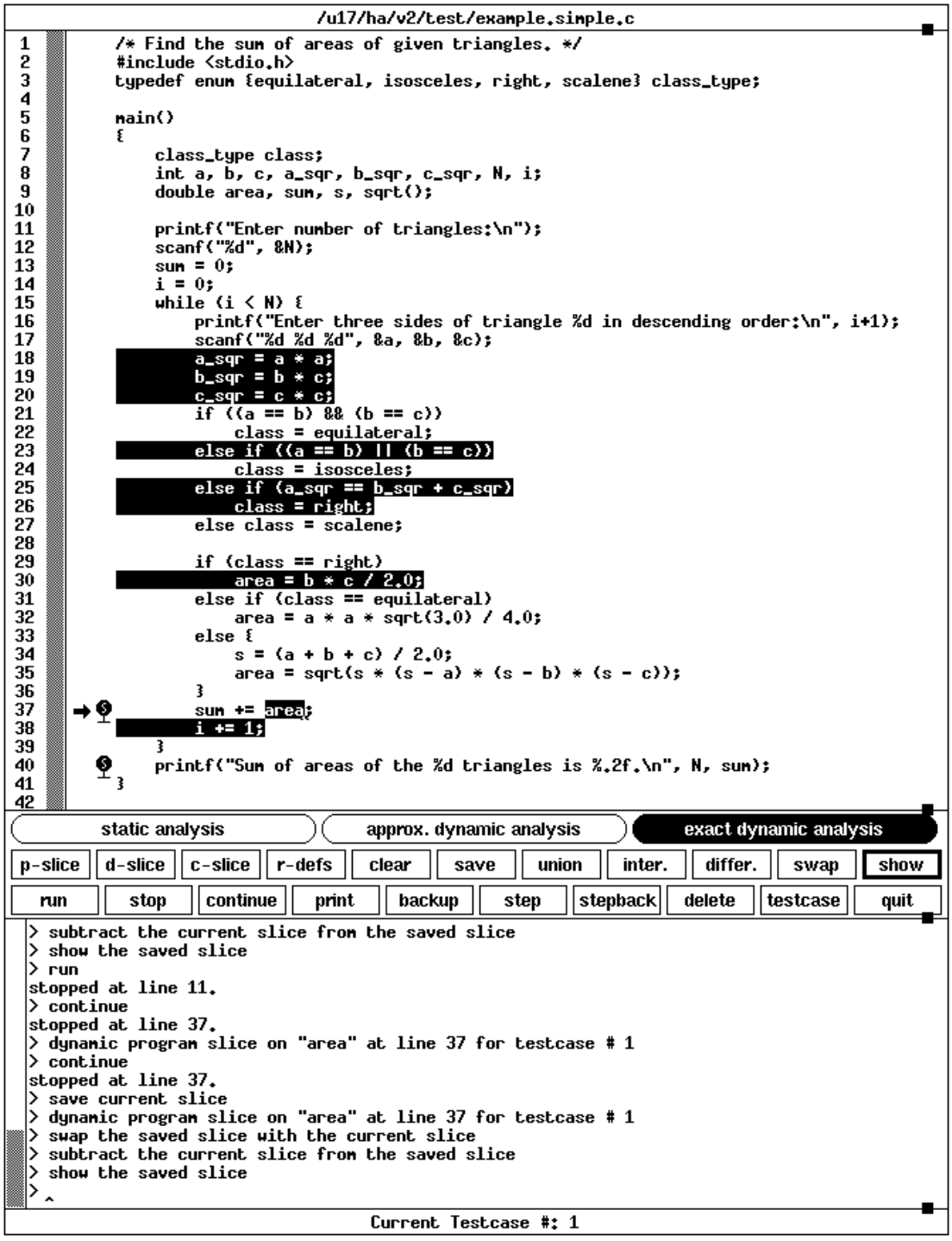

Figure 6.14 Result of subtracting the dynamic program slice of area on line 37 during the first loop iteration for testcase #1 from the corresponding slice during the second loop iteration.

correct value of an output variable, *var*. We then make some changes to P_n, e.g., to add some new functionality to it. But when the modified program, P_e, is executed on T, it produces a wrong value of *var*. In situations like this, another debugging strategy would be to examine the changes we made to P, especially those portions of the changes that affect the value of *var*. This can be facilitated by obtaining the difference in dynamic program slice of P_e with respect to *var* for testcase T from the corresponding slice of P_n. That is, we should examine:

$$DPS(P_e, \cdot, \cdot, \cdot) - DPS(P_n, \cdot, \cdot, \cdot)$$

This, of course, requires that we keep track of correspondence between statements in the old and the new program. If the changes made to P_n consist of addition of new statements, deletion of some old statements, or simple modifications of some old statements, this correspondence is easy to keep. But if the changes made are more complex, e.g., moving statements around and modifying them at the same time, the correspondence between statements in the two programs may not be clear any more. But in situations where the correspondence between statements in P_n and P_e remains largely intact, this strategy should be useful. Our prototype debugging tool, SPYDER, presently does not keep track of correspondence between statements in versions of the same program, hence it does not currently support this strategy.

6.2 Combining Data Slices

A program slice contains both assignments and predicate expressions. So the difference of two program slices may also contain both assignments and predicates. While it is relatively easy to reason about the presence or absence of an assignment statement from the difference set, doing the same for a predicate expression may become difficult. Suppose an assignment statement, A, is present in the dynamic slice of *var* at the program end for an error-revealing testcase, T_e. If A is absent from the difference set obtained by subtracting the corresponding dynamic program slice with respect to a non-error-revealing testcase, T_n, from the above slice, it means

that the value computed by A also contributed towards computation of *var* for T_n, and hence, baring a case of coincidental correctness, A is unlikely to contain a fault. Now suppose an **if**-predicate, P, is also present in the dynamic program slice with respect to T_e but absent from the difference set mentioned above. Does this mean P is unlikely to contain a fault? It may be the case that P incorrectly evaluated to *true* in the case of T_e thus producing the wrong output, but correctly evaluated to *true* in the case of T_n. As there are only two possible outcomes of evaluating a predicate, *true* or *false*, it is relatively easy for faulty predicates to achieve coincidental correctness.

Thus it may be helpful if, instead of examining differences of dynamic program slices, we examined differences of dynamic *data* slices. If we find that the value of *var* is incorrect at the end of the program execution for testcase T_e, we should first apply the strategies discussed in Section 6.1 but substituting dynamic data slices in place of program slices. Then, examining the resulting set, either we are able to localize the fault, or we find that an assignment is incorrectly present or absent in the resulting set. In the latter case, we can examine the control slice of that assignment. Again, either we are able to identify the fault with some predicate in the control slice or we are able to identify another variable, var', used in some predicate in the control slice to have a wrong value. We can then repeat the same process with respect to var'.

Let us apply this strategy for fault localization for the program in Figure 6.1 for testcase #1. Our first step would be obtain the dynamic data slice with respect to sum on line 40. Figure 6.15 shows this slice. Also, we know that the value of sum is printed correctly for testcase #2. So we also obtain the corresponding dynamic data slice with respect to testcase #2. Figure 6.16 shows this slice. We next subtract the latter data slice from the former. Figure 6.17 shows the resulting set. It contains only one assignment on line 30, indicating that as far as the value of sum is concerned the only thing that the program does for testcase #1 but not for testcase #2 is the assignment on line 30. Examining this assignment reveals that it computes the area of a right triangle. But testcase #1 does not contain any right triangle which means this assignment is incorrectly reached during the program execution. So we next examine

the control slice of line 30, shown in Figure 6.18. We find that line 30 is reached because the predicate on line 29 evaluates to true. That happens because class has an incorrect value at that instance. So we have localized the problem to class having an incorrect value on line 29. We then examine the dynamic data slice of class on line 29, shown in Figure 6.19. We find that only the assignment on line 26 is in the slice and that assignment is also incorrectly reached. Figure 6.20 shows the control slice of line 26. Examining this slice, we find that assignment on line 26 is reached because the predicate on line 25 evaluates to true. This indicates that the value of one of the variables referenced by the predicate must be incorrect. After examining these variables we find that the value of b_sqr is indeed incorrect. Figure 6.21 shows the dynamic data slice of b_sqr. Examining this data slice, we can quickly isolate the fault on line 19.

6.3 Summary

In this chapter we have proposed several strategies for fault localization based on examining the difference, union, intersection, or a combination of these operations, on dynamic program- or data slices. It should be emphasized that these strategies are only heuristics; they are not guaranteed to work in all situations. They provide mechanisms that are useful in many debugging situations, and it is up to the programmer to use them effectively.

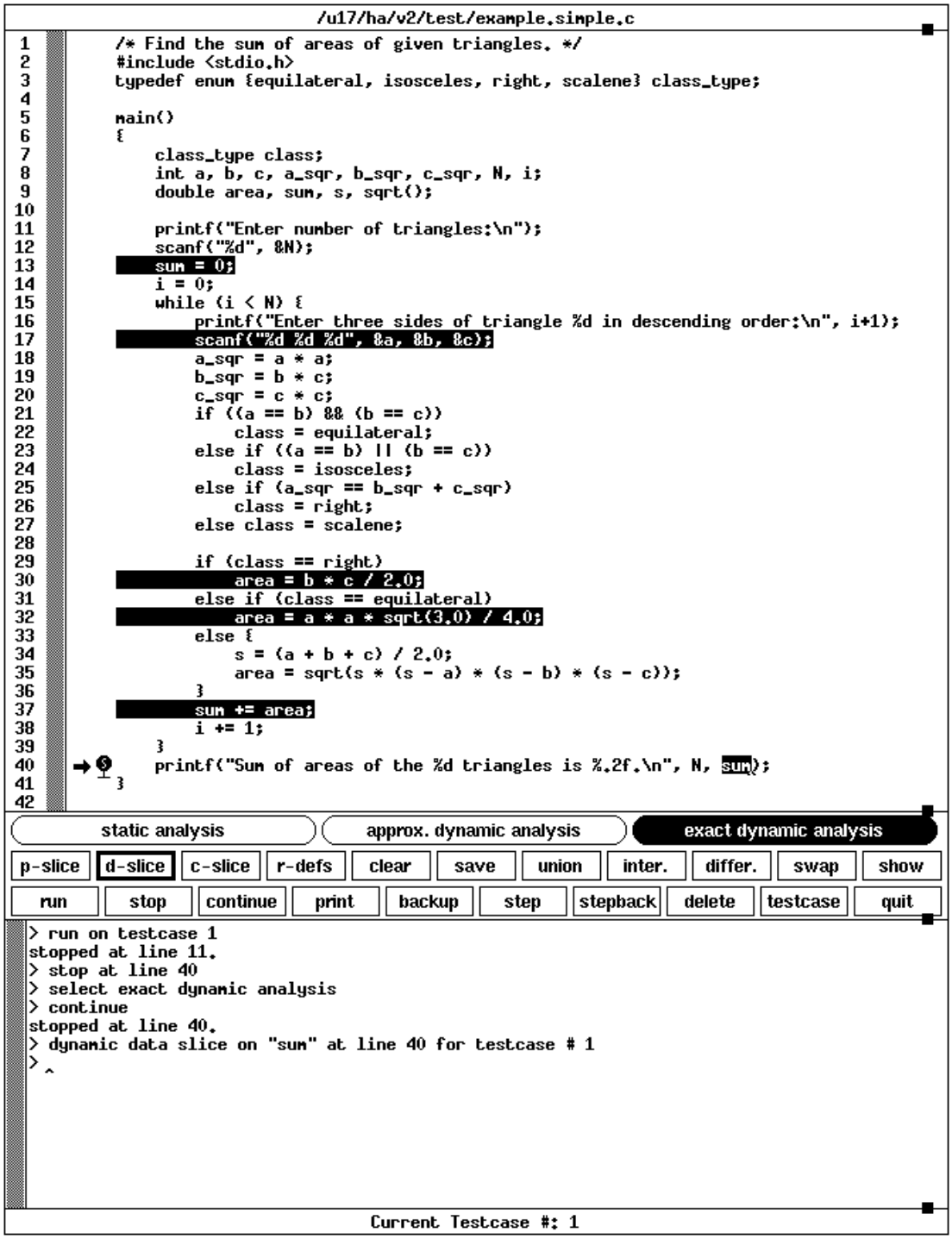

Figure 6.15 Dynamic data slice of sum on line 40 for testcase #1.

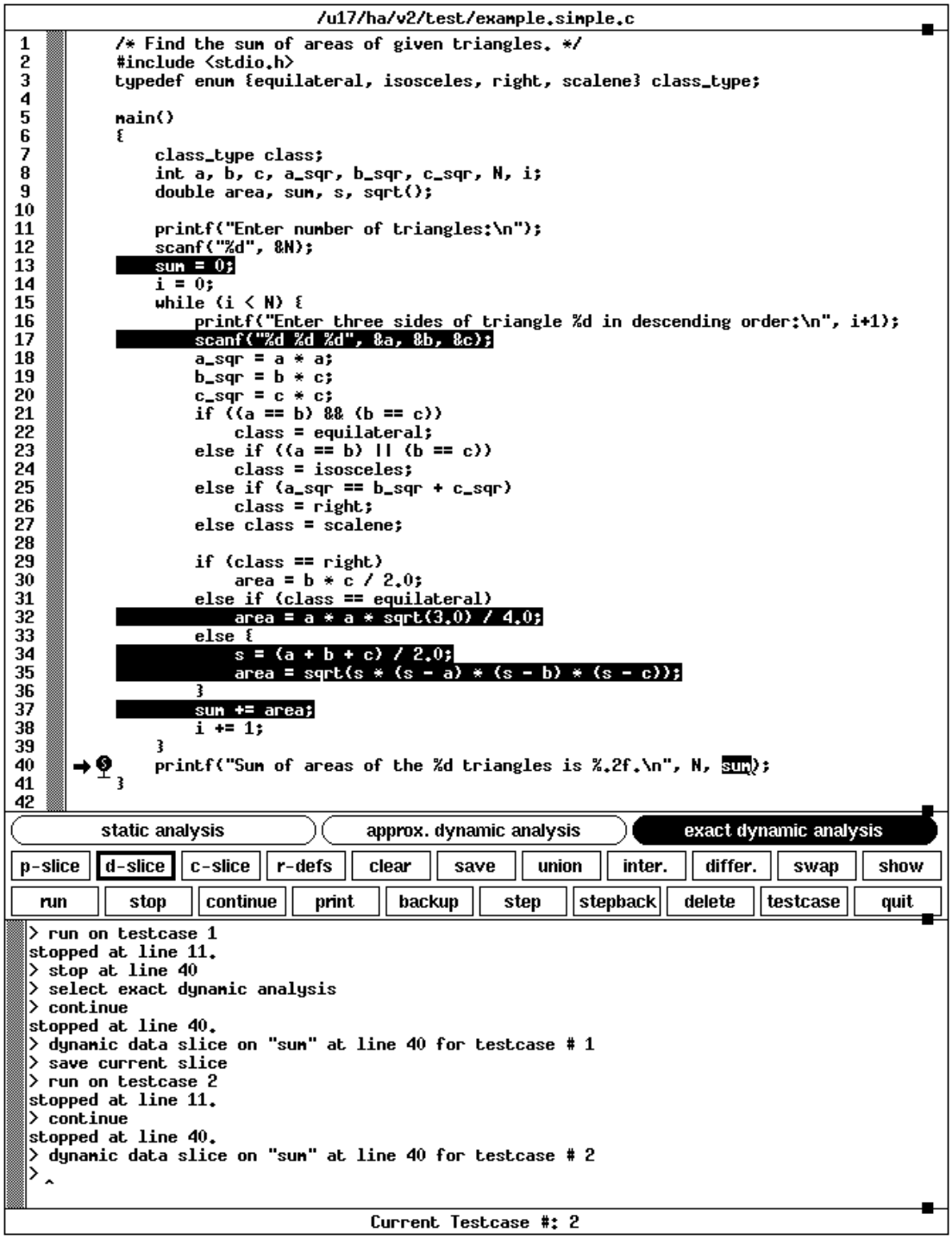

Figure 6.16 Dynamic data slice of sum on line 40 for testcase #2.

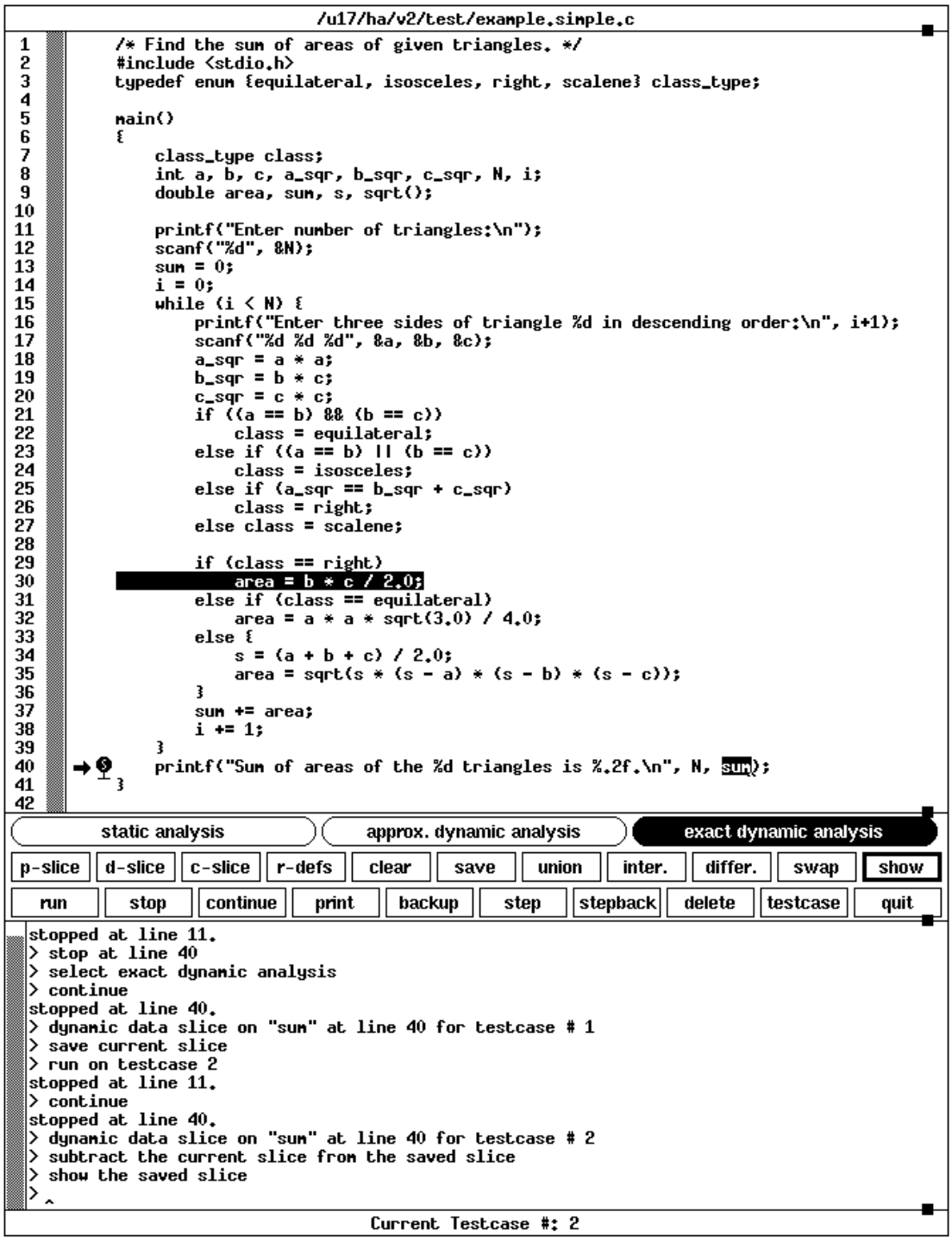

Figure 6.17 Result of subtracting the dynamic data slice of sum on line 40 for testcase #2 from the corresponding slice for testcase #1.

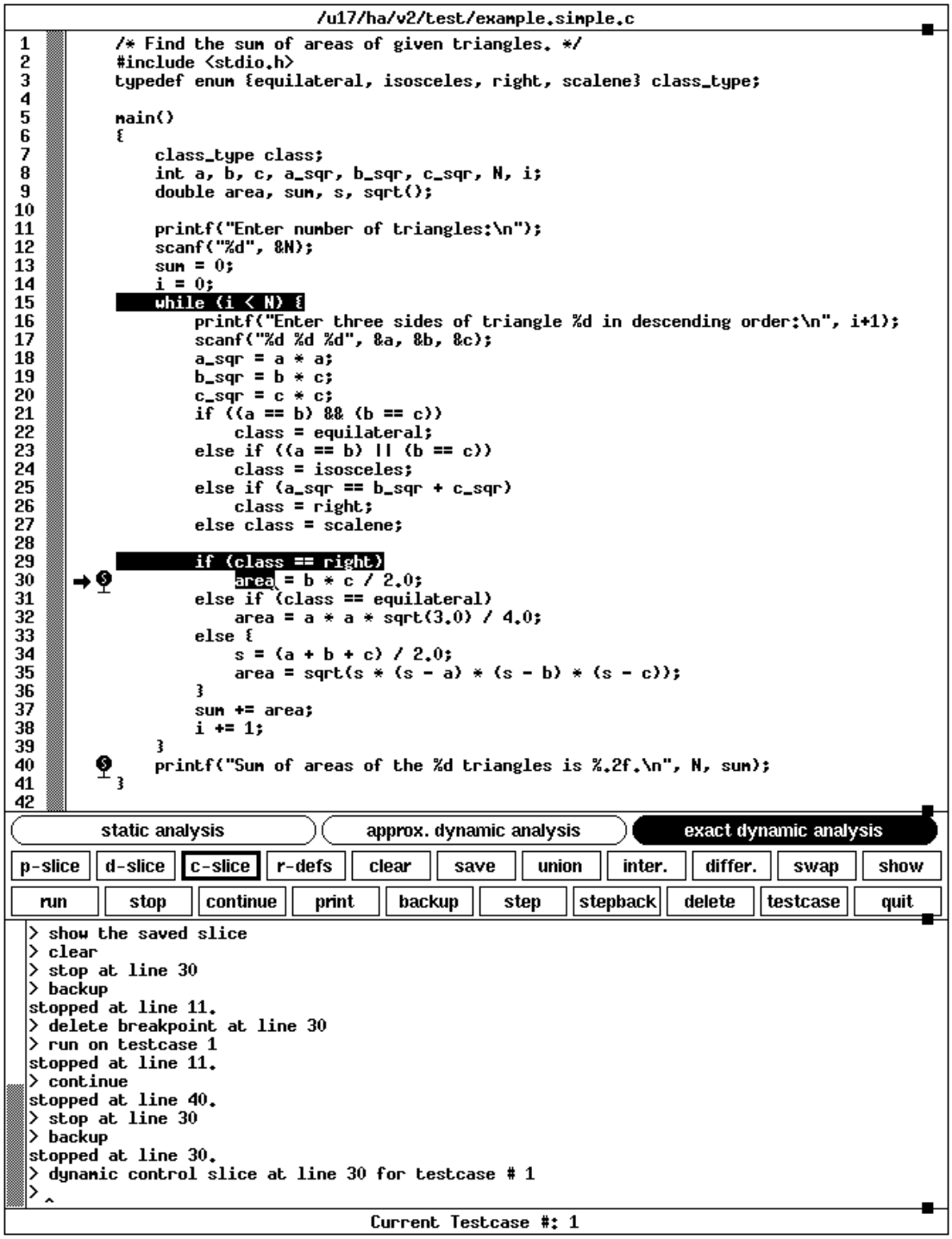

Figure 6.18 Control slice of line 30 for testcase #1.

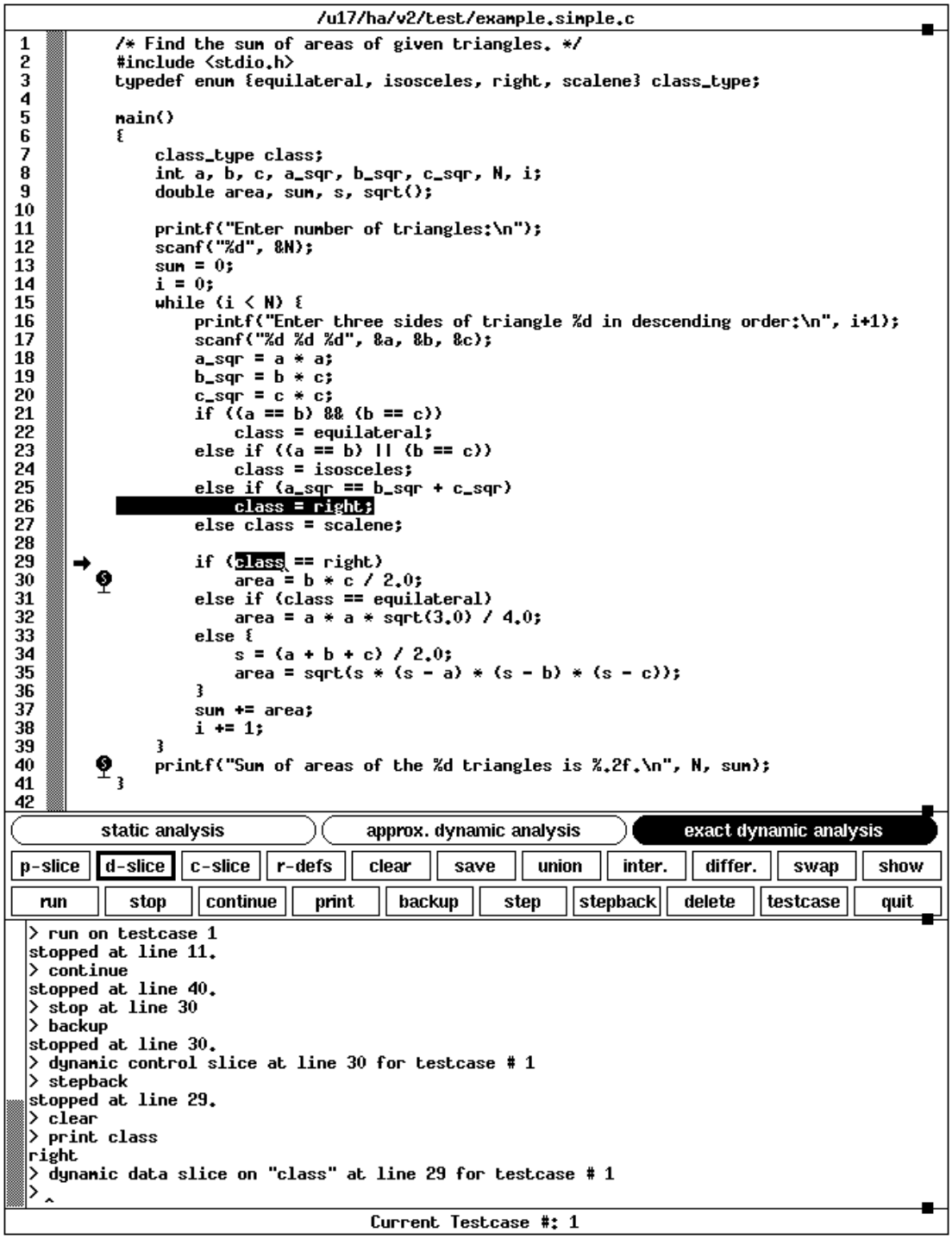

Figure 6.19 Dynamic data slice of class on line 29 for testcase #1.

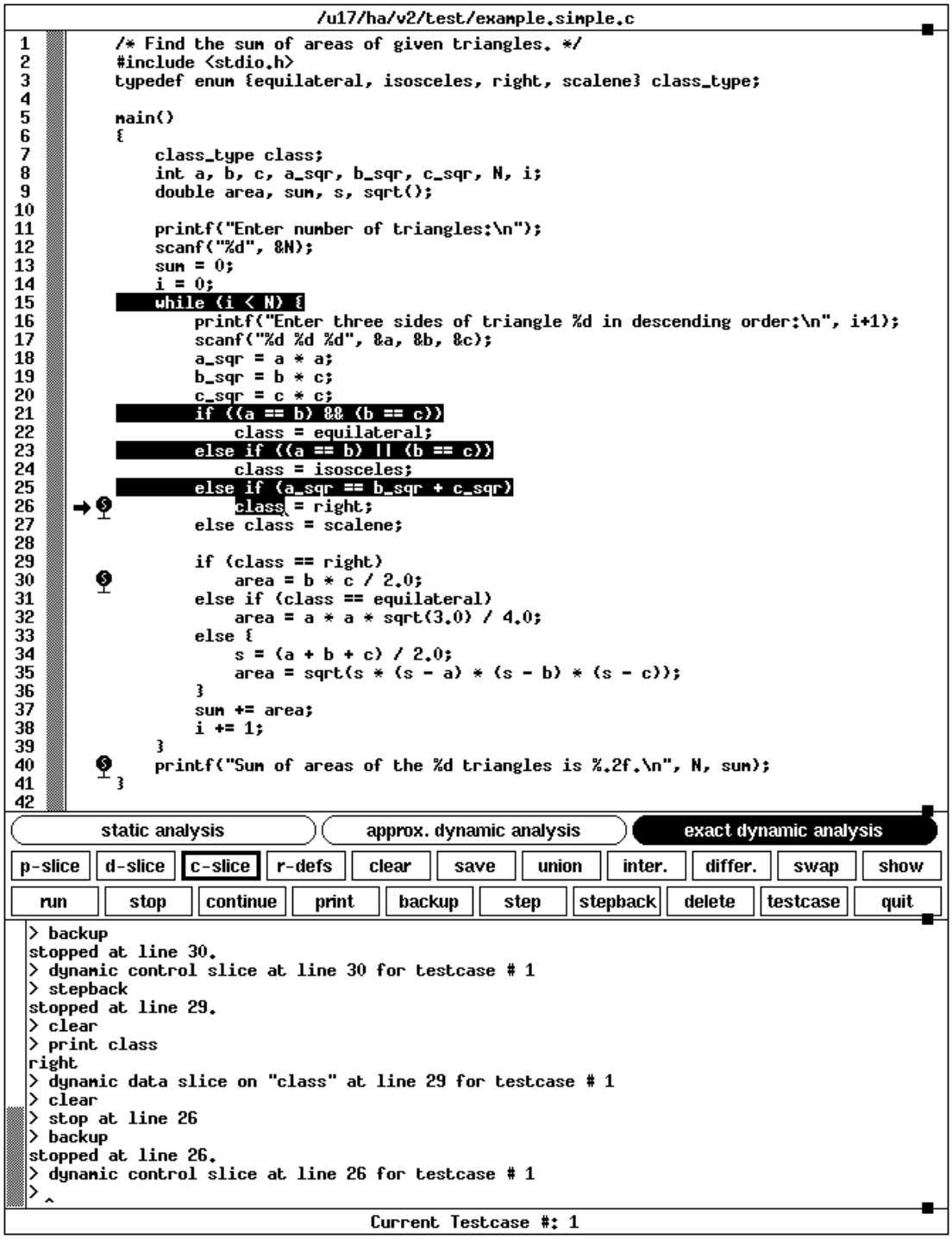

Figure 6.20 Control slice of line 26 for testcase #1.

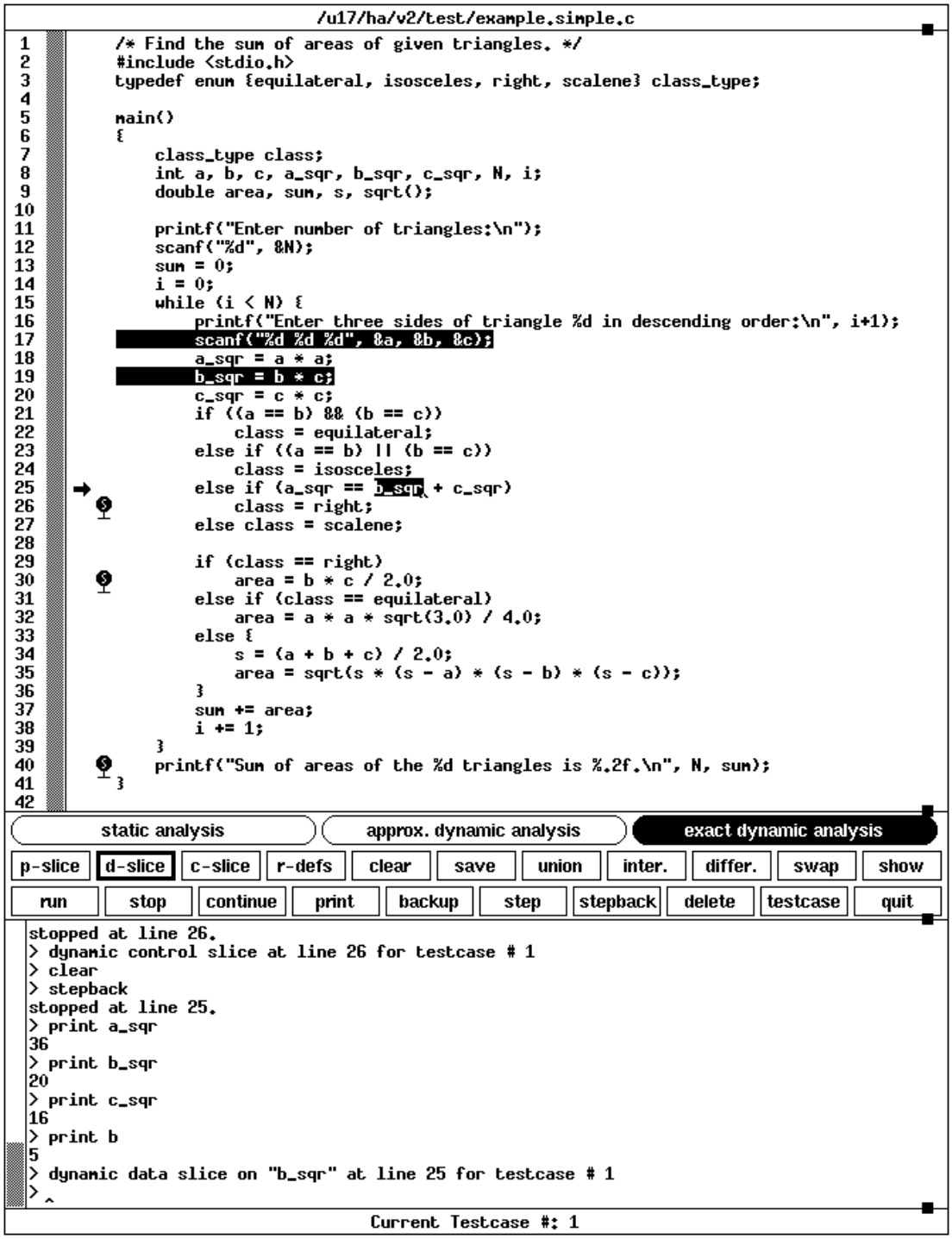

Figure 6.21 Dynamic data slice of b_sqr on line 25 for testcase #1.

7. EXECUTION BACKTRACKING

One of the three steps in the debugging paradigm outlined in Chapter 1 is to restore the program state to that attained when the program execution last reached a given earlier statement. This can be easily achieved if the debugging tool provided an execution backtracking facility. In this chapter, we outline two approaches to implement statement-level execution backtracking. We first consider, in Section 7.1, how such a facility may be implemented for a simple programming language. Then, in Section 7.2, we discuss how additional language features can be handled under the same approaches.

7.1 Simple Execution Backtracking

At any time during the program execution, the *state* of the program consists of two things: values of all variables in the program at the time, and the location of the program control. Executing a statement causes one program state to be transformed into another. The type of the transformation depends on the type of the statement. For simplicity, let us first consider the simple language used in Chapter 3 that consists of assignment, conditional (if-then-else), loop (while-do), and input-output (read, write) statements and their compositions. We also assume for the moment that only scalar variables are used and that expressions do not cause side-effects. An assignment statement modifies the program state so that the new state is identical to the previous state except for two things: the value of the variable on the left hand side of the assignment may be different, and the control location is modified to be the successor statement. The **if** and the **while** predicates, on the other hand, only modify the control-location.

Thus execution of a statement essentially causes two kinds of effects on the program state: it modifies control location, and it may change values of one or more variables. Backtracking over a statement would require some way of undoing these two effects. The first effect, modifying the control location, can easily be undone if we simply record the execution history of control locations—the sequence in which statements are visited during program execution. Then undoing the control-location effect would simply require traversing this sequence in the opposite direction. The second effect, viz., changing values of variables, can easily be undone if before executing the statement we save the current values of variables modified by the statement. Then undoing this effect would simply require restoring the previous values saved. In the following section we discuss the execution history saving approach to backtracking. Then in section 7.1.2 we show that by constraining backtracking in a particular way, it can be implemented much more efficiently.

7.1.1 The Execution History Approach

In Section 3.2.2 we associated a *def* set with each node in the flow-graph of a program. Nodes in a flow-graph correspond to simple statements (assignments, reads, writes) and predicate expressions (conditions in **if**, **while** statements) in the program. The *def* set of an assignment statement consists of the variable on the left-hand-side of the assignment, while the *def* set of a predicate expression is the empty set. If the language permits expressions with side-effects, then *def* sets of both assignments and predicates may contain several variables. The *def* set of a **read** statement includes variables read by the statement, and that of a **write** statement is an empty set. Henceforth, we also refer to **read** statements as assignment statements.

To be able to backtrack to any statement arbitrarily far back in execution, we need to record the complete execution history of statements and the corresponding previous values of variables in their *def* sets. Then we can backtrack to any statement by

restoring the previously saved values of *def* set variables starting at the current location and going backwards until that statement is encountered in the saved execution history.

For example, Figure 7.2 shows the execution history of the program in Figure 7.1 for the testcase ($x = 7$, $y = 3$), annotated with the saved *def* set values. The program state at the end of the execution is $x = 7$, $y = 3$, $r = 1$, $q = 2$, and $temp = 3$. If we wish to backtrack execution until just before the loop at statement S7 started execution, then we will have to restore all values in the *def* sets starting at the end of the execution history and going backwards up to (and including) entry 10. The program state will now become $x = 7$, $y = 3$, $r = 7$, $q = 0$, and $temp = 12$, just like it was when control first reached the while loop at statement S7.

Note that if a statement nested in a loop body is executed N times because of loop iteration, there will be N corresponding entries for that statement in the execution history. Hence, for programs with long-running loops, the execution history of the program can grow very long. Further, because the number of times a loop iterates may depend on run-time input, the length of the execution history may not be bounded at compile time. Thus, the space required to record the execution history and the corresponding *def* set variable values may not be allocated in advance. Besides having this space problem, the above approach is also time inefficient. If we have to backtrack up to a statement before a loop, then we have to backtrack individually over each iteration of the loop. In the next section we outline a different approach that does not have these disadvantages.

7.1.2 The Structured Backtracking Approach

Just as we defined *def* sets of assignment statements and predicate expressions, we can also define *def* sets of composite statements like **if** and **while** to be the set of all those variables whose values *could* be modified during the execution of that statement. For example, the *def* set of a **while** loop will consist of all variables that could be modified if the loop body is executed one or more times.

```
S1:     read (x, y);
S2:     r := x;
S3:     q := 0;
S4:     temp := y;
S5:     while (temp <= x) do
S6:             temp := temp * 2;
S7:     while (temp <> y) do begin
S8:             q := q * 2;
S9:             temp := temp div 2;
S10:            if (temp <= r) then begin
S11:                    r := r - temp;
S12:                    q := q + 1;
                end;
        end;
S13:    write (q, r);
```

Figure 7.1 Program to divide two integers.

1: S1, [x:?, y:?]
2: S2, [r:?]
3: S3, [q:?]
4: S4, [temp:?]
5: S5, [ ]
6: S6, [temp:3]
7: S5, [ ]
8: S6, [temp:6]
9: S5, [ ]
10: S7, [ ]
11: S8, [q:0]
12: S9, [temp:12]
13: S10, [ ]
14: S11, [r:7]
15: S12, [q:0]
16: S7, [ ]
17: S8, [q:1]
18: S9, [temp:6]
19: S10, [ ]
20: S7, [ ]
21: S13, [ ]

Figure 7.2 Execution history of the program in Figure 7.1 for the testcase X = 7, Y = 3, along with the saved *def* set values.

def sets of composite statements can be computed from the *def* sets of their constituent assignment statements. If we denote source statements by S, S_1, and S_2, and a boolean expression by *cond*, then the *def* sets of some composite statements may be computed as follows:

$$def(S_1; S_2) = def(S_1) \cup def(S_2)$$

$$def(\textbf{if } cond \textbf{ then } S) = def(cond) \cup def(S)$$

$$def(\textbf{if } cond \textbf{ then } S_1 \textbf{ else } S_2) = def(cond) \cup def(S_1) \cup def(S_2)$$

$$def(\textbf{while } cond \textbf{ do } S) = def(cond) \cup def(S)$$

For example, the *def* set of the **while** loop beginning at statement S7 in Figure 7.1 is {q, temp, r}, and that of the **if** statement at statement S10 is {r, q}.

Like assignment statements, we can also save values of all variables in the *def* set of a composite statement just before executing that statement. To backtrack over a **while** statement, we simply need to restore previous values of variables in its *def* set instead of undoing the effect of each iteration of the loop in the reverse order. If we also restrict backtracking such that one may *not* directly backtrack from a statement outside a composite statement to a statement nested inside it, then we can avoid both the space and time inefficiency problems of the execution history approach outlined above. Under the structured backtracking approach, for each statement—simple or composite—the debugger allocates space to save just one instance of values of all variables in its *def* set. Any time control reaches that statement, the debugger saves the current values of variables in its *def* set in the same space. So every time a statement in a loop body gets executed, the current values of variables in its *def* set overwrite the previously saved values. Thus, it is possible to backtrack from a statement in a loop body to an earlier statement in the same loop body for the current iteration, but it is not possible to directly backtrack to a previous iteration of that loop.

To illustrate how backtracking is constrained, consider the following program segment:

```
S1:     ....
S2:     while cond do begin
S3:             ....
S4:             ....
S5:             ....
        end;
S6:     ....
S7:     if cond then begin
S8:             ....
S9:             ....
        end else begin
S10:            ....
S11:            ....
        end;
S12:    ....
S13:    ....
```

All of the following instances of backtracking are *not* allowed under structured backtracking:

from S6 to S5

from S12 to S9

from S9 to S3

from S3 in iteration i to S5 in iteration $i-1$

Following are some valid instances of structured backtracking:

from S2 to S1

from S5 to S3 within the same iteration

from S6 to S2

from S4 to S1

from S9 to S8

from S13 to S7

Note that one can backtrack from a statement inside a loop to a statement outside it. These restrictions are analogous to those followed in structured programming and included in most modern language standards—disallowing jumps to a statement inside a loop from outside it, but allowing **breaks** from inside a loop to outside.

The restriction on backtracking over only complete statements is not an unduly constraining one. In a sense, it is similar to encouraging structured execution in the backward direction. As such, analyzing the effects of statements in reverse order should be much easier and logical because the user needs to consider only one complete statement at a time. If one needs to backtrack to a statement inside a composite statement from a statement outside it, one can always backtrack first to the beginning of the composite statement, and then execute forward to the desired statement.

Under the structured backtracking approach, we no longer need to save the execution history. Also, for each statement the amount of space required to save values of variables in its *def* set is fixed, so all the space required may be allocated in advance. In the next section we derive bounds on space requirements of the structured backtracking approach.

7.1.3 Bounds on Space Requirements

If an assignment statement is nested N levels deep, then the variables modified by it would belong to *def* sets of N statements—$N-1$ composite statements in which it is nested, and the assignment statement itself.[1] Let A be the total number of assignment statements in the program, s_i be the size of change set of the ith assignment statement, n_i be the nesting level of the ith assignment statement, and S

[1]For simplicity, an assignment statement not nested in any composite statement is assumed to be at level one; an assignment statement with a single enclosing **if** or **while** statement is assumed to be at level two; and so on.

be the sum of sizes of *def* sets of all statements in the program. Then we have:

$$S = \sum_{i=1}^{A}(n_i \times s_i) \tag{7.1}$$

Or,

$$S \leq A \times max_{i=1}^{A} n_i \times max_{i=1}^{A} s_i \tag{7.2}$$

Let

$$\alpha = max_{i=1}^{A} n_i \tag{7.3}$$

and

$$\beta = max_{i=1}^{A} s_i \tag{7.4}$$

That is, α represents the maximum nesting level in the program, and β represents the size of the largest change set of all assignment statements in the program. Then, from (7.2), (7.3), and (7.4), we have:

$$S \leq A \times \alpha \times \beta \tag{7.5}$$

Let L be the length of the program in number of source lines. Then, because there are A assignment statements in the program, there can be at most $L - A$ composite statements in the program. As only composite statements increase nesting levels of statements, the maximum nesting level of any assignment statement in the program can be $L - A + 1$. That is,

$$\alpha \leq L - A + 1 \tag{7.6}$$

From (7.5) and (7.6) we have:

$$S \leq A \times (L - A + 1) \times \beta \tag{7.7}$$

Or,

$$S \leq \beta \times (L \times A - A^2 + A) \tag{7.8}$$

For a given L, the right-hand side of this equation is a function of A. We find its maximum by differentiating it with respect to A and equating the derivative to zero.

That gives us:[2]

$$A = \frac{L+1}{2} \tag{7.9}$$

Substituting this value of A in (7.8), we get:

$$S \leq \frac{1}{4} \times \beta \times (L^2 + 2 \times L + 1) \tag{7.10}$$

Or,

$$S = O(L^2) \tag{7.11}$$

But this is only a theoretical worst-case upper bound. In practice, we have observed that both α and β are usually small constants. Denoting the product $\alpha \times \beta$ by c, we get from (7.5):

$$S \leq c \times A \tag{7.12}$$

But the number of assignment statements in the program is bounded by the program length, so we also have:

$$A \leq L \tag{7.13}$$

Combining (7.12) and (7.13) we get:

$$S \leq c \times L \tag{7.14}$$

Or,

$$S = O(L) \tag{7.15}$$

That is, in the usual case, the sum of the sizes of the *def* sets of all statements in the program is of the order of the length of the program. In particular, this size is independent of running time.

7.2 Extensions

In the previous section we used a simple programming language to describe two approaches to implement execution backtracking. In this section we examine how other language features like records, arrays, pointers, and procedures are handled.

[2]For simplicity of presentation, we treat the function as continuous, although it is a function of the discrete variable A. For this discrete function, the maximum occurs at $A = \lceil L/2 \rceil$.

Records are easy to handle: We simply need to treat each field of a record as a separate variable. Handling arrays and pointers, however, requires more work. Assignment to an array element, like $A[i] := \ldots$, or an indirect assignment through a pointer, like $pointer \uparrow := \ldots$, differs from an assignment to a scalar variable, like $var := \ldots$, in that the exact address of the memory location modified by the assignment in the latter case is fixed and known at compile time, whereas that in the former case is not. We can, however, easily overcome this problem by recording both the address and the contents of the memory location modified by the assignment just before it is executed. Then the execution can be backtracked over the assignment by restoring the contents at the address saved. Or, all—records, arrays, and pointers—can be uniformly handled by defining *def* sets in terms of memory cells, as was done in Chapter 4 the case of dynamic slicing. Now, just before executing an assignment, both its *def cell* and the contents of the *def* cell are saved. Backtracking over an assignment now means restoring the saved contents into the appropriate memory cell.

When an indirect assignment through a pointer appears within a loop, the address of the memory location assigned may vary from one iteration to another. In this case, the precise *def* set of the loop can not be determined at compile time. Thus it is constructed at run-time: each time around the loop, the address and the contents of the memory cell assigned are added to *def* set of the loop. If the *def* cell being added is already present in the *def* set of the loop, then it is simply ignored. The size of the *def* set of the loop, in this case, may not be bounded at compile time. But the space bounds derived in Section 7.1.3 would still hold if, in the case of an indirect assignment, we treat s_i to be the size of the complex data-structure modified incrementally by multiple occurrences of the same statement during the program execution.

Backtracking into a procedure call from outside it, in the execution history approach, requires recreating the stack frame of the call. In the structured backtracking approach, however, procedure calls are treated just like composite statements: one

may not backtrack into a procedure call from outside it. The side-effects of the procedure call constitute the *def* set of the call. Like a loop, when an indirect assignment is executed inside a procedure, the *def* set of the procedure call is updated appropriately. Recursion is handled by saving *def* sets of all statements inside a procedure on the current stack frame of the procedure.

Backtracking over I/O operations poses special problems. Any system can at most undo things that are directly under its control. If any of its actions have effects outside the boundary of the system, then the system, in general, cannot always retract them. For instance, we have no way to save the "state" of a line-printer to allow us to later backup to that state. One possible approach to handling I/O operations involves use of buffering along with *pushback* operations, similar to the "ungetc" operation in the C programming language standard library. Another way to handle file I/O is to record the current offset of the file pointer from the start of the file just before executing the file I/O statement. Then, backtracking over a read from a file also entails restoring the file pointer to the saved offset. SPYDER, our prototype debugging tool, employs the latter technique using the "lseek" system call in Unix.

7.3 Summary

In this chapter we have outlined two approaches to implementing execution backtracking. Our prototype debugging tool, SPYDER, currently supports backtracking the execution history approach outlined in Section 7.1.1 Just as the forward program execution is suspended whenever a breakpoint is encountered, SPYDER 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, SPYDER will have restored the program state to whatever it was when the execution last reached that point.

Most conventional debuggers provide facilities to step through the program execution, statement by statement. SPYDER also provides a back-stepping facility with which the user can *step back* through the program one statement at a time.

For example, consider again the program in Figure 1.1 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, as shown in Figure 7.3. 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 7.4 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, we find that all elements 0–9 of a have the value 0. We now set a breakpoint on line 16 and continue the program execution. 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. 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, the control 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.

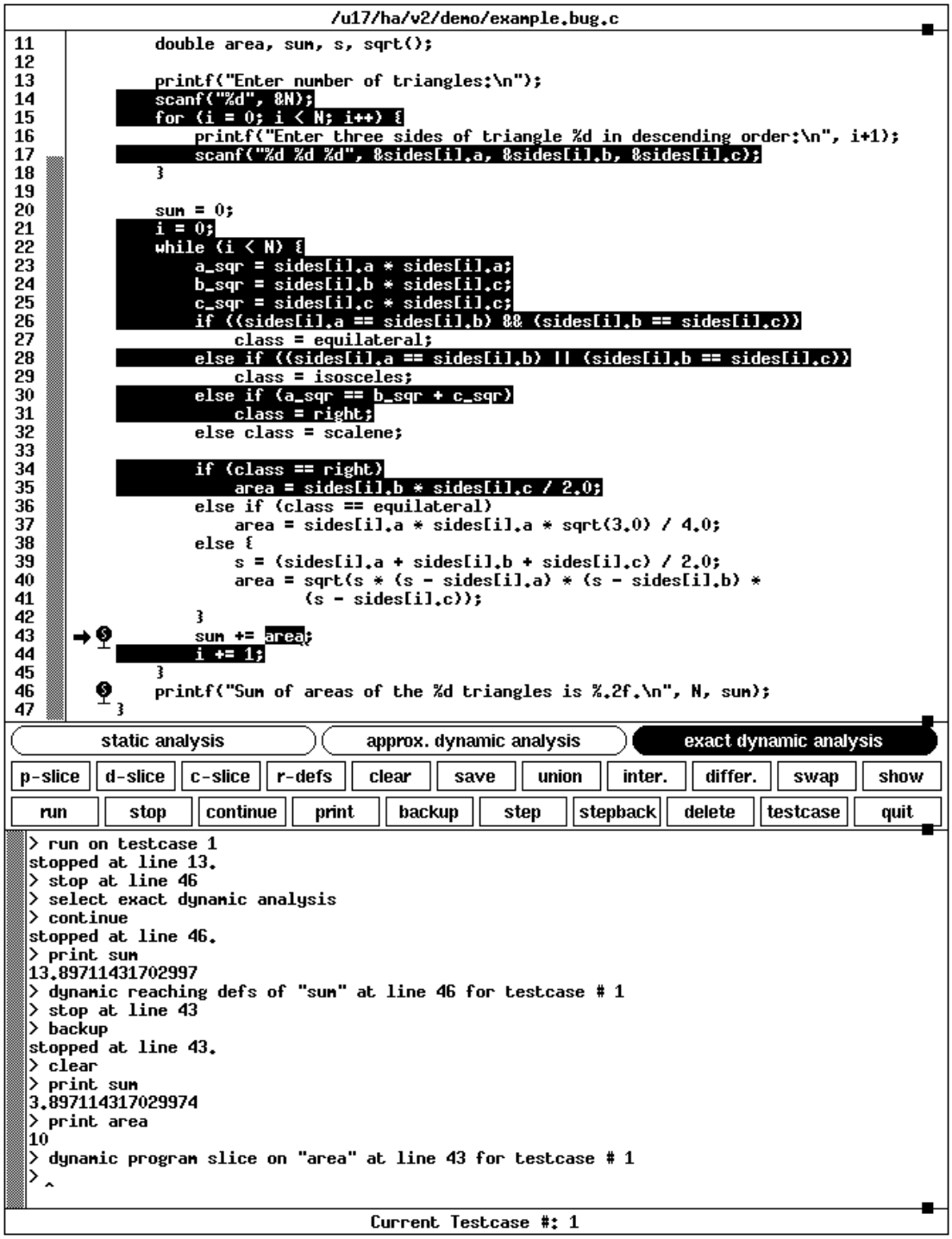

Figure 7.3 Tool screen after backtracking from line 46 to line 43

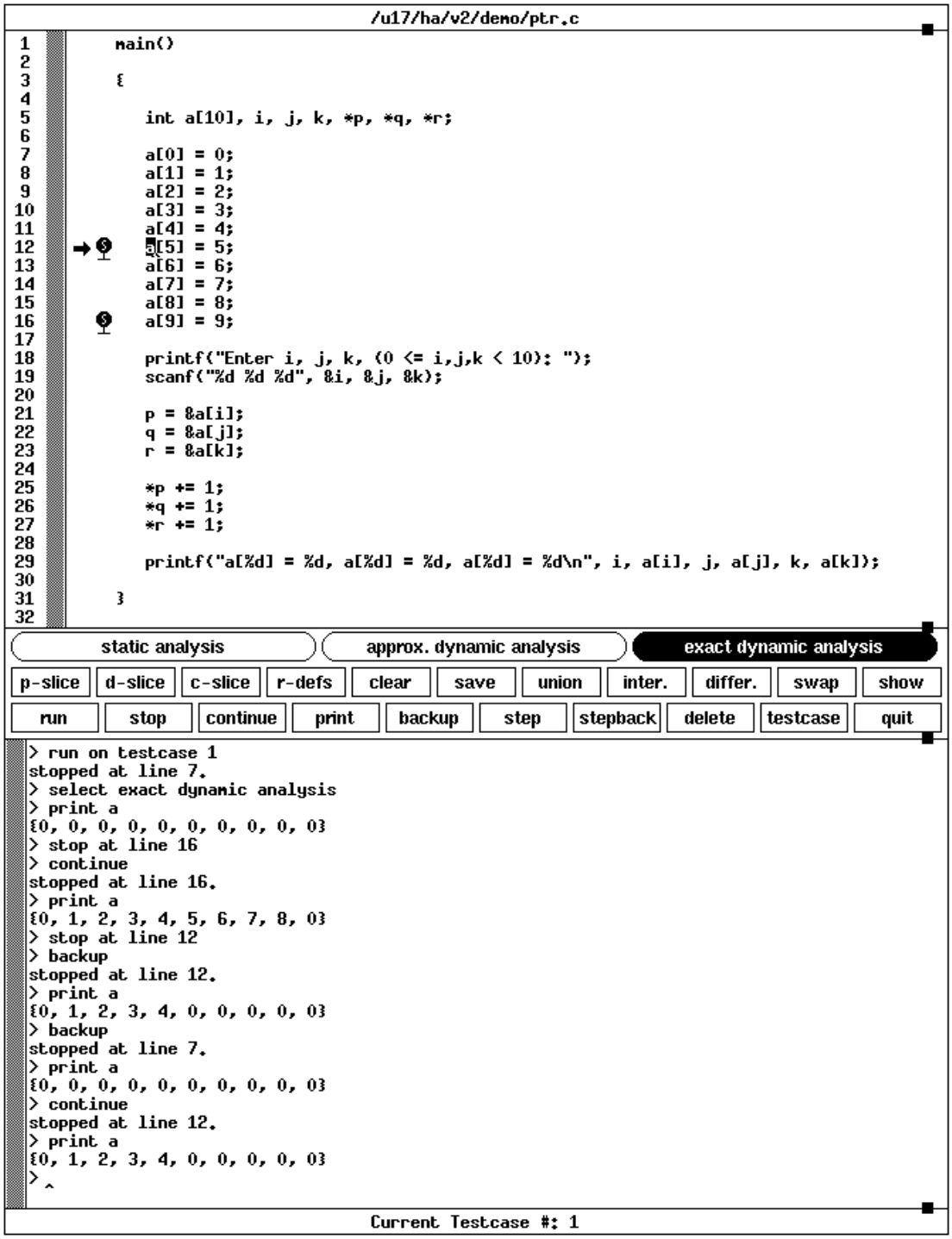

Figure 7.4 Execution backtracking from line 16 to line 12 to line 7

8. SPYDER: A PROTOTYPE IMPLEMENTATION

Throughout this dissertation we have used figures to present examples of various types of program slices. These figures are actually screen dumps of our prototype debugging tool, SPYDER, in action. In this chapter, we discuss functions provided by SPYDER in order to support our slicing–guessing–backtracking paradigm of debugging. We also briefly discuss the approach we used to implement SPYDER.

8.1 The Tool Screen

Figure 8.1 shows a snapshot of the SPYDER screen during a debugging session. The screen is divided into five windows tiled vertically in the following order (from top to bottom):

- File-Label Window
- Source Window
- Commands Window
- Output Window
- Testcase-Label Window

The file-label window simply displays the name of the source file currently displayed in the source window.

The source window provides a small viewing "window" into the source code of the program being debugging. Contents of this window can be scrolled up and down using the vertical scroll bar on the left of the source window. Another vertical bar, on the left of the scroll bar, displays source line numbers for reference. Line numbers

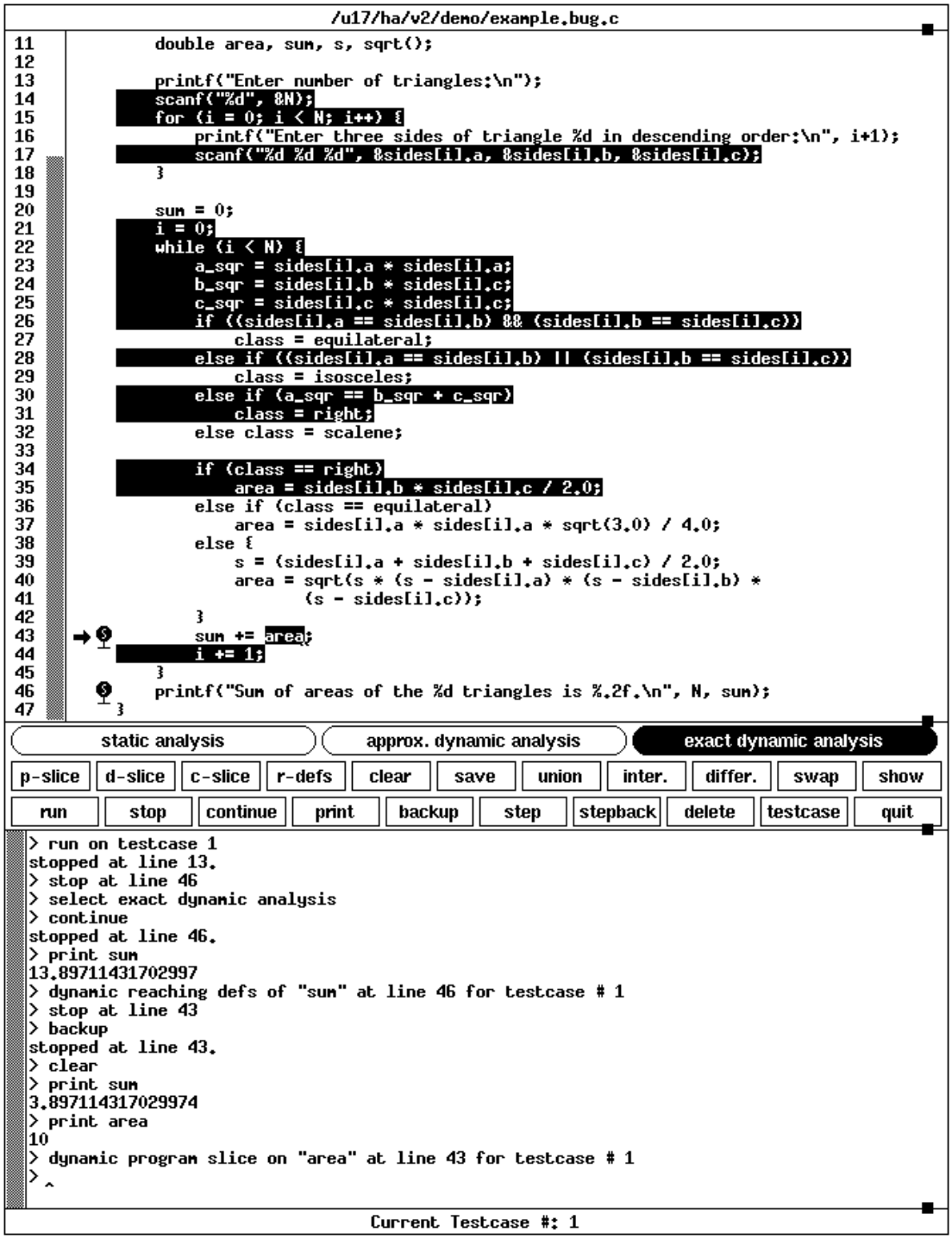

Figure 8.1 A snapshot of the SPYDER screen during a debugging session.

also scroll up and down along with the source code. Slices are displayed in the source window by highlighting the lines that belong to the slice in reverse "video." If a stop icon, in the shape of a stop sign, is displayed to the left of a source line, it indicates that a breakpoint is currently set there. Multiple stop icons indicate that multiple breakpoints are set at the same time. An arrow icon, in the shape of a right-pointing solid arrow, to the left of a source line indicates that the control is currently stopped at that line. There can be at most one arrow icon displayed at any time.

The commands window consists of three rows of buttons for issuing various slicing, backtracking, and traditional debugging commands discussed in the next section.

The output window echos the commands entered by clicking on buttons in the commands window, and displays SPYDER's response to some of these commands, most notably the print command.

Finally, the testcase-label window indicates the id-# of the testcase that is currently selected. All dynamic slicing commands are executed with respect to this testcase.

8.2 SPYDER Commands

Functions supported by SPYDER can be classified into the following five categories:

- Selection Setting Commands
- Slicing Commands
- Fault Guessing Commands
- Backtracking Commands
- Traditional Debugging Commands

8.2.1 Selection Setting Commands

Four commands fall into this category—three corresponding to the three toggle buttons in the top row of the commands window, and one corresponding to the button

labeled *testcase* in the bottom row. Exactly one of the three toggle buttons in the top row, labeled *static analysis, approximate dynamic analysis*, and *exact dynamic analysis*, is selected at any time, indicating the current slicing criterion selected. These three buttons correspond to static slicing, dynamic slicing using Approach 1, and dynamic slicing using Approach 3, discussed in Chapter 3, respectively. The criterion selected specifies whether the slices obtained with buttons labeled *p-slice, d-slice, c-slice*, and *r-defs* in the second row, are static slices, approximate dynamic slices, or exact dynamic slices.

In case one of the latter two toggle buttons is selected, we also need to specify the id-# of the testcase to use, as dynamic slices are always obtained with respect to a testcase. This can be done using the button labeled *testcase* in the bottom row of the commands window. 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. The testcase-label window is updated appropriately every time a testcase selection is made.

Testcases themselves are specified using a separate tool called *tcgen*. When this tool is invoked, it executes the program's object code, captures all input supplied to the program, and saves it as a testcase. It also assigns an id-# to this testcase that is used to refer to it during the current testcase selection process mentioned above.

8.2.2 Slicing Commands

Slicing functions of SPYDER are provided by the first four buttons in the middle row of the commands window, labeled *p-slice, d-slice, c-slice*, and *r-defs*. The button labeled *p-slice* is used to obtain program slices. The buttons labeled *d-slice* and *c-slice* are used to obtain data- and control-slices, discussed in Chapter 5, respectively. The button labeled *r-defs* is used to obtain immediate reaching definitions of variables and other l-valued expressions. The slices obtained are static, approximate dynamic, or exact dynamic, depending on which of the three toggle buttons in the first row of the commands window is currently selected. The variable argument to

these commands is specified by selecting appropriate text in the source window using the mouse. Any text there can be selected by clicking and dragging the mouse over it. The location argument is specified implicitly: In case of static and approximate dynamic slicing, it is the location associated with the current selection in the source window; in case of exact dynamic slicing, it is the location where the control is currently stopped, indicated by the right pointing arrow icon. The testcase argument in case of both approximate and exact dynamic slices is specified using the *testcase* button, as discussed above.

8.2.3 Fault Guessing Commands

SPYDER also provides mechanisms for guessing regions of the program that are likely to contain a fault by combining slices in various ways discussed in Chapter 6. Any slice currently displayed in the source window can be saved into an "accumulator" by clicking on the button labeled *save*. The button labeled *union* performs a union of the currently displayed slice with that saved in the accumulator and stores the result in the accumulator. That is, it "adds" the currently displayed slice to the accumulator. The currently displayed slice remains unaffected. The button labeled *inter* performs an intersection instead of union in a similar fashion. The button labeled *differ* takes the difference of the accumulator and the currently displayed slice and stores the result into the accumulator. That is, it "subtracts" the currently displayed slice from the accumulator. If, instead, one wishes to subtract the saved slice from the one currently displayed, one can first swap the two slices using the *swap* button, and then use the *differ* button. Contents of the accumulator can be displayed any time using the *show* button. This command will replace the currently displayed slice with that saved in the accumulator. The latter will remain unaffected. The currently displayed slice can also be cleared any time using the *clear* button. The accumulator is unaffected by this command. If one wishes to clear the accumulator, one can do so by first clearing the display with the *clear* command and then storing it in the accumulator using the *save* command.

8.2.4 Backtracking Commands

SPYDER provides two execution backtracking functions: *backup* and *stepback*. Clicking on the *backup* button starts execution backtracking. Backtracking continues until a breakpoint is reached when the control is returned to the debugger with the program state restored to what it was the last time control reached that location. The button labeled *stepback* provides reverse single-stepping. Currently, SPYDER uses the execution history approach to execution backtracking, discussed in Chapter 7, thus execution can also be backtracked to locations inside composite statements. At present, execution can not be backtracked into procedure calls from outside them because doing so requires that the corresponding stack frame be recreated. Thus back-stepping over a procedure call restores the program state to the state just before the procedure was invoked.

8.2.5 Traditional Debugging Commands

As we mentioned earlier, our slicing–guessing–backtracking paradigm works in conjunction with traditional interactive debugging commands. For this reason, we also implemented some of the basic traditional debugging functions such as breakpoints, single-stepping, and examining variable values, into SPYDER. A breakpoint can be easily set by selecting some text on the corresponding line in the source window using the mouse and clicking on the *stop* button. Clicking on the *delete* button removes a breakpoint from the currently selected line if one is set there. Values of variables or expressions can be printed by selecting the corresponding text in the source window and clicking on the *print* button[1]. Clicking on the *continue* button resumes forward execution. Execution can also be single-stepped in forward direction using the *step* command. Execution can be resumed at the beginning of the program using the *run* command. A debugging session with SPYDER is terminated by clicking on the *quit* button.

[1]Values printed are always "current" values, i.e., with respect to the location where control is currently stopped, not with respect to the location associated with the selection on the screen.

Many more standard debugging commands could be supported by SPYDER, but as our goal was to demonstrate usefulness of slicing, guessing, and backtracking functions, we implemented only the bare minimum of other functions we needed.

8.3 Implementation

SPYDER is built into versions of the GNU C compiler "Gcc" [Sta90] and the GNU source-level debugger "Gdb" [Sta89]. As mentioned earlier, our intent was not to write a production-quality tool but to demonstrate the feasibility as well as the usefulness 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.

8.3.1 Modifications to the Compiler

We modified Gcc to produce the program dependence graph along with the object code of the given program. This required making changes to the parser. The modified parser, apart from doing its normal functions, also builds the program's flow and control dependence graphs in a syntax directed manner, as described in Chapter 3. Both these graphs share the same set of nodes but different sets of edges. Each node is also annotated with its *use* and *def* sets, consisting of the l-valued expressions that are used and defined, respectively, by the node. After parsing is complete, the flow graph is traversed to compute data dependencies among nodes, and a third set of edges belonging to the data dependence graph is created. The aggregate graph, that now consists of flow, control, and data dependence subgraphs, is then written out. This graph is later read and used by the debugger to find the various static slices.

8.3.2 Modifications to the Debugger

Gdb was modified to read the aggregate graph consisting of flow, control, and data dependence subgraphs, produced by the modified Gcc. Code was added to traverse the aggregate graph to find static reaching definitions, and static data, control, and program slices.

To support dynamic slicing and execution backtracking, Gdb was also modified to record the execution history of the program as it executes. The execution history consists of a list of nodes in the aggregate graph appended in the order in which they are visited during program execution. Each entry in this list also constitutes a node in the dynamic dependence graph. Each such node is annotated with *use* and *def* sets consisting of memory-cells used and defined by the node, respectively. The contents of memory-cells that belong to *def* sets are also saved for each node. The modified Gdb captures all this information by setting numerous "transparent" breakpoints in the program that the user does not see. It associates appropriate callback functions with each of these transparent breakpoints. These callback functions perform all the work of recording execution history, determining *use* and *def* memory-cells, saving contents of the *def* memory-cells, etc., using the *use* and *def* set annotations of the corresponding nodes in the static program dependence graph. After all callback functions associated with a transparent breakpoint have been performed, program execution is resumed automatically.

Whenever the program execution stops, e.g., when a breakpoint is reached, the newly built portion of the execution history is traversed to create dynamic data and control dependence edges of all nodes newly added to the dynamic dependence graph. Now the dynamic dependence graph can be traversed to find dynamic reaching definitions and dynamic data, control, and program slices. To perform execution backtracking, the execution history is traversed in the reverse order and the previously saved contents of memory-cells in *def* sets of nodes encountered are restored into the corresponding memory-cells. Backtracking is stopped when a node is reached that has a breakpoint currently set there. Segments of the execution history may correspond to

program execution inside procedures. Thus care is taken while backtracking execution over these segments. *def* sets of nodes in these segments may contain memory-cells that belonged to the stack-frame of the procedure which may no longer be accessible. Contents of such memory cells are not restored during backtracking as we currently do not support backtracking into a procedure from outside it. But such *use* and *def* sets are still used for the purpose of finding dynamic slices.

Another major modification that was made to Gdb was to provide a window and mouse-based user interface for it so slices could be displayed by highlighting corresponding source lines. We also added hooks into the system so all the traditional debugging functions supported by SPYDER, such as setting breakpoints, could also be performed by simply selecting appropriate text in the source window using the mouse and clicking on appropriate command buttons. We used the Athena widget set and the Xt toolkit of the X Window System, Version 11, Release 4 to build this interface.

8.4 Summary

In this chapter, we discussed our prototype debugging tool, SPYDER, that explicitly supports our slicing–guessing–backtracking paradigm. We discussed various functions provided by SPYDER that make it possible for the programmer to follow this paradigm. We also briefly discussed the approach we took to quickly implement SPYDER.

9. CONCLUSIONS AND FUTURE DIRECTIONS

Debugging is a complex and difficult activity. The person debugging a program must determine the cause and the location of the program failure. The failure may be manifested far away from the fault itself—both textually (in terms of source lines) and temporally (in terms of execution time). Providing facilities that increase the ability of the programmer to identify the location or the nature of the fault involved will lead to more efficient debugging. In this dissertation, we have presented a debugging paradigm and a prototype tool that attempt to provide precisely these facilities. Our experience with both the paradigm and the tool so far has convinced us that they are quite useful, and when applied properly they can result in significant savings in debugging time. They are, however, no panacea. They only provide useful mechanisms; it is up to the user to use them effectively. In this chapter we list some limitations of the these techniques, discuss some lessons learned from the implementation, and finally conclude this dissertation with some ideas on future research directions.

9.1 Limitations of the Paradigm

In Chapter 5, we mentioned that a fault manifests itself, directly or indirectly, in terms of a data or a control inconsistency. While this is true, it requires that the programmer translates the externally visible symptom of the fault into an internal program symptom in terms of a data or a control problem before the techniques described here may be used. This translation may not always be an easy one to make. For example, if the external symptom is that some value in the program output is incorrect, the corresponding internal symptom—the value of a variable or an expression being incorrect at a print statement—may be easily determinable. But if the external symptom is a missing value in a list, the programmer may first have to

use traditional debugging facilities to find the corresponding internal symptom, and then only the techniques described here may be used.

Also, slices do not make dependences among multiple occurrences of the same statement explicit. For example, if a statement inside a loop body is included in a slice and the value it computes during one iteration depends on the value it computes during the previous iteration, this dependence is not made explicit in the slice as both occurrences are grouped together while displaying the slice.

Execution backtracking also has some limitations. Backtracking over a statement requires that all side-affects of executing the statement be undone. But, as we mentioned before, any system can at most undo things that are within its control. If executing a statement has effects outside the boundaries of the program, e.g., into the operating system, then the outside agents affected must cooperate with the debugging tool to enable execution backtracking. In other situations when executing a statement may have effects outside the controlling environment, it may not even be feasible to undo them. In such situations one may have to either accept "partial" backtracking, or resort to reexecuting the program from the beginning.

The programmer using these techniques must be aware of these limitations. The limitations are not restrictive enough so as to preclude their use. On the contrary, our experience has shown that despite these limitations, these techniques are extremely useful in quickly isolating program faults.

9.2 Limitations of the Current Implementation

As we mentioned before, we built our prototype tool on top of an existing compiler and a debugger. While this choice enabled us to quickly build a working system with which we could experiment with the proposed techniques and gain more insight into their usefulness, it also enforced some limitations on what could or could not be supported. For example, in Gdb, one cannot associate breakpoints with expressions or statements; one can only associate them with source lines. As we use transparent

breakpoints to capture all the information required for dynamic slicing and backtracking, the above limitation requires that there be a one-to-one correspondence between the smallest syntactic units used in slicing and backtracking—assignments and predicates—and source lines. Thus the current implementation requires that no source line contains more than one assignment, and that predicates and assignments appear on different lines. If the program does not follow these conventions, the results of backtracking and slicing may be unpredictable. Of course, it is easy to provide a preprocessor that converts a program into the "canonical" form acceptable to the tool.

It may be noted that these limitations are of the current implementation, not those of the techniques themselves. These limitations arose because both the compiler and the debugger were not originally written to support these techniques. They would not arise if we implemented them in the context of an interpretor, or if we wrote our own compiler and the debugger.

9.3 Lessons Learned from the Implementation

At the beginning of this research, we were faced with the following question: Should we build our own compiler and a debugger in order to implement the techniques we proposed, or should we use existing tools? The first alternative was rejected because it would have meant significant investment in time and energy in building things that were not the focus of our research. Having decided on the second alternative, the next question was: Which existing compiler and debugger should we use? One alternative was to build the prototype on top of an interpretor, but we couldn't do so because of the lack of availability of a good interpretor for C in the public domain. The Gnu tools, on the other hand, had several factors in favor of them:

- source code for these tools is freely available,
- they are quickly portable to multiple hardware platforms, and
- they support ANSI C.

They also had some factors against them:

- no documentation about their implementation was available,
- both these tools are big (e.g., in terms of number of source lines), and
- they are undergoing evolution with frequent new releases and bug fixes.

Despite the negative factors, we believed the positive factors were strong enough to warrant their use. In the hindsight, we certainly do not regret our decision to use the Gnu tools. We believe the rich functionality of Gdb was an important factor in enabling us to quickly develop our prototype.

Another decision we had to make was whether to modify the compiler to produce instrumented code to gather runtime information necessary for dynamic slicing and backtracking, or to have the debugger probe the program execution to collect the same information. Both approaches had their own advantages and disadvantages. We decided to use the latter approach because we believed it would be easier to implement and experiment with. One side benefit of using this approach was that by delaying instrumentation of probes until debugging time it becomes possible to interactively control which parts of the program to instrument and when to instrument. With this approach it would be possible to start without any instrumentation and first use static slicing techniques to narrow down the search for the fault to the extent possible to a smaller region of the program. Then we could successively increase program instrumentation to use approximate and exact dynamic slicing techniques but on successively smaller and smaller program regions. This way we don't pay the cost associated with instrumentation all the time. Also, once we have debugged a particular region—a procedure, a module, or some other program segment—it would be possible to instruct the debugger to "uninstrument" that region. If we had used the former approach, it would mean either to pay the cost of instrumentation throughout the program execution, or to repeatedly recompile the program with lesser and lesser instrumentation.

One consequence of our decision to let the debugger perform all the instrumentation and collect necessary runtime information was that a lot of time is spent by the system in context switching between the debugee and the debugger processes. As processes are heavyweight processes in Unix, and as a context switch from one process to another is a costly operation there, we can notice a significant slow-down in program execution in long-running programs because of the constant context switching. But in other systems that provide lightweight processes and fast context switches the overhead would be substantially smaller. Also, we can use the approach discussed in the paragraph above to restrict this context switching to small regions. Another promising approach to reduce the context switching overhead is to have the debugger "patch" the object code of the debugging process with instructions that do all the necessary logging of information required for dynamic slicing and backtracking purposes [Kra91, DKM91].

9.4 Future Directions

We conclude this dissertation by listing some ideas on extending our work and some preliminary thoughts on other research directions.

9.4.1 Fault Prediction Heuristics

In Chapter 6, we discussed several heuristics based on combining multiple dynamic slices in certain ways. The success of these heuristics depends on the judicious selection of the criteria used to generate the slices to be combined. The selection is usually made based on the programmer's knowledge of how the program has "behaved" on various testcases during program testing. A promising research direction is to investigate if the tool can capture the wealth of information generated during testing, and make recommendations about which criteria to use based on this information.

9.4.2 User Interfaces

SPYDER uses a simple user interface to display slices: The program source is displayed in a window and source lines belonging to the slice are highlighted in reverse "video". But at any instance only a fraction of the program is visible in the source window. If one wants to examine a portion of the program that is currently not visible, one must manually scroll up or down the text window. While this interface may be sufficient for small programs, it is obviously not suitable for large programs. An important line of research would be investigate how to display slices of large programs with thousands of lines of code. One possible approach is to have an interface that displays slices in a hierarchical manner. For example, at the top level, a slice may simply consist of the names of relevant modules that have an affect on the chosen slicing criterion. Then any module could be expanded to show the next level of the slice within the module. This, for example, may simply consist of the names of relevant functions or procedures within the module. Any procedure or function, in turn, can then be expanded to reveal the corresponding intra-procedural slice.

In the previous section, we mentioned that one limitation of slices is that they do not show inter-occurrence dependences of the same statement. One way to overcome this limitation would be to construct another interface where multiple occurrences of the same statement are not grouped together. But this interface may have some of the same disadvantages of traces—it may flood the user with too much information. One way to handle this could be to have a combination of the two interfaces.

9.4.3 Extensions to Other Domains

As we mentioned in Chapter 1, the scope of this research included examining the proposed debugging paradigm in the context of sequential and procedural programming languages such as Pascal and C. An obvious next step would be to examine how the techniques described here can be extended for use with parallel or distributed programs, and for programs written in other language domains such as functional, logical, or object oriented programming languages.

The idea of dynamic slicing is particularly appealing in the context of parallel programs because unlike the techniques used in static analysis, dynamic analysis will not face the problem of an exponential growth in the number of possible state transitions of a parallel program. Dynamic analysis will only analyze the events that did occur and in the order they actually occurred, not all possible events and their orderings. Of course, one must be aware that any dynamic instrumentation of a parallel program may modify its timing characteristics, thus possibly shielding some timing-related faults in the program. But this limitation should not prevent one from exploring other significant advantages of using these techniques.

Similarly, the idea of dynamic slicing is also appealing for object-oriented programs for the simple reason that an object-oriented program heavily relies on dynamic binding of messages sent to an object to methods that implement them, because there may be many message-handlers for the same message in the class hierarchy of the object. Dynamic slicing will highlight relevant code in the handlers that were actually used and not all possible handlers that could be used.

Dynamic slicing will have the same advantage in case of logic programs: There may be several clauses defining one predicate in a logic program, and dynamic slicing will analyze what actually occurred during unification instead of analyzing the possibly infinite search space.

9.4.4 Other Applications

Another research direction is to investigate if we could also put the information gathered for slicing and backtracking to other uses besides debugging. One possibility is to examine if we can perform some on-the-fly code optimizations based on all the dynamic information gathered. Another possibility is to investigate if, in cases when methods used to find static slices provide overly conservative information (such as including the whole program in the slice because the program used unconstrained pointers), we can use the union of dynamic slices with respect to a judiciously selected set of testcases to provide an estimate of the corresponding static slice. This approach

may be useful in program maintenance when a good regression testset may already be available.

BIBLIOGRAPHY

BIBLIOGRAPHY

[AB82] W. R. Adrion and M. A. Branstad. Validation, verification, and testing of computer software. *ACM Computing Surveys*, 14(2):159–192, June 1982.

[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.

[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.

[ANS83] ANSI/IEEE. IEEE standard glossary of software engineering terminology. IEEE Std 729–1983, IEEE, New York, 1983.

[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.

[Bea83] Bert Beander. Vax DEBUG: an interactive, symbolic, multilingual debugger. In *Proceedings of the Software Engineering Symposium on High-Level Debugging*, Pacific Grove, CA, March 1983. ACM SIGSOFT/SIGPLAN. Software Engineering Notes, 8(4):173–179, August 1983; SIGPLAN Notices, 18(8):173–179, August 1983.

[Boe81] B. W. Boehm. *Software Engineering Economics.* Prentice Hall, Englewood Cliffs, NJ, 1981.

[Bro88] M. H. Brown. *Algorithm Animation.* MIT Press, Cambridge, MA, 1988.

[BS85] M. H. Brown and R. Sedgewick. Techniques for algorithm animation. *IEEE Software*, pages 28–39, January 1985.

[Car83] Thomas A. Cargill. The Blit debugger. *Journal of Systems and Software*, 3(4):277–284, December 1983.

[Car86] Thomas A. Cargill. The feel of Pi. In *Proceedings of the Winter Usenix Conference*, pages 62–71, Denver, CO, January 1986.

[CC87a] Fun Ting Chan and Tsong Yueh Chen. AIDA—a dynamic data flow anomaly detection system for Pascal programs. *Software Practice and Experience*, 17(3):227–239, March 1987.

[CC87b] James S. Collofello and Larry Cousins. Towards automatic software fault location through decision to decision path analysis. In *Proceedings of the 1987 National Computer Conference*, pages 539–544, 1987.

[CF89] Robert Cartwright and Matthias Felleisen. The semantics of program dependence. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989. ACM SIGPLAN. SIGPLAN Notices, 24(7):13–27, July 1989.

[Cou88] Deborah S. Coutant. Doc: a practical approach to source-level debugging of globally optimized code. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, Atlanta, GA, June 1988. ACM SIGPLAN. SIGPLAN Notices, 23(7):125–134, July 1988.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. 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):296–310, June 1990.

[DE88] Mireille Ducasse and Anna-Maria Emde. A review of automated debugging systems: Knowledge, strategies and techniques. In *Proceedings of the Tenth International Conference on Software Engineering*, pages 162–171, Singapore, April 1988. IEEE.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, NJ, 1976.

[DKM91] Richard A. DeMillo, Edward W. Krauser, and Aditya P. Mathur. Compiler-integrated program mutation. In *Proceedings of the Fifteenth Annual Computer Software and Applications Conference.* IEEE, September 1991.

[DLP79] R. A. DeMillo, R. J. Lipton, and A. J. Perlis. Social processes and the proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979.

[DM91] Richard A. DeMillo and Aditya P. Mathur. On the use of software artifacts to evaluate the effectiveness of mutation analysis for detecting errors in production software. Technical Report SERC-TR-92-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, March 1991.

[DMMP87] R. A. DeMillo, W. M. McCracken, R. J. Martin, and J. F. Passafiume. *Software Testing and Evaluation.* Benjamin/Cummings Publishing Co., Menlo Park, CA, 1987.

[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.

[Fet88] J. H. Fetzer. Program verification: the very idea. *Communications of the ACM*, 31(9):1048–1063, September 1988.

[FO76] Lloyd D. Fosdick and Leon J. Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys*, 8(3):305–330, September 1976.

[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.

[GL89] Keith B. Gallagher and James R. Lyle. A program decomposition scheme with applications to software modification and testing. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, volume II, Software Track, pages 479–485, January 1989.

[Gou75] J. D. Gould. Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies*, 7(1):151–182, January 1975.

[Hen82] John Hennessy. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems*, 4(3):323–344, 1982.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.

[How87] W. E. Howden. *Functional Program Testing and Analysis.* McGraw-Hill, New York, 1987.

[HPR89a] Susan Horwitz, Phil Pfeiffer, and Thomas Reps. Dependence analysis for pointer variables. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989. ACM SIGPLAN. SIGPLAN Notices, 24(7):28–40, July 1989.

[HPR89b] 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.

[HR67] Jr. Hartley Rogers. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, New York, 1967.

[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.

[Hua79] J. C. Huang. Detection of data flow anomaly through program instrumentation. *IEEE Transactions on Software Engineering*, SE-5(3):226–236, May 1979.

[ISO87] Sadahiro Isoda, Takao Shimomura, and Yuji Ono. Vips: a visual debugger. *IEEE Software*, 4(3):8–19, May 1987.

[JS85] W. L. Johnson and E. Soloway. Proust: Knowledge-based program understanding. *IEEE Transactions on Software Engineering*, 11(3):267–275, March 1985.

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

[KKL+81] D. J. Kuck, R. H. Kuhn, B. Leasure, D. A. Padua, and M. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, pages 207–218, Williamsburg, VA, January 1981. ACM SIGPLAN.

[KL88a] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29:155–163, October 1988.

[KL88b] Bogdan Korel and Janusz Laski. Stad—a system for testing and debugging: User perspective. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 13–20, Banff, Canada, July 1988. ACM.

[KL90] Bogdan Korel and Janusz Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195, November 1990.

[KL91] Bogdan Korel and Janusz Laski. Algorithmic software fault localization. In *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, volume II, pages 246–252, 1991.

[Kra91] Edward W. Krauser. *Compiler-Integrated Software testing.* PhD thesis, Department of Computer Sciences, Purdue University, West Lafayette, IN, 1991.

[LD85] R. L. London and R. A. Duisberg. Animating programs using Smalltalk. *IEEE Computer*, pages 61–71, August 1985.

[LH88] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation.* ACM SIGPLAN, July 1988. SIGPLAN Notices, 23(7):21–34, July 1988.

[LOS86] H. Longworth, L. Ott, and M. Smith. The relationship between program complexity and slice complexity during debugging tasks. In *Proceedings of COMPSAC.* IEEE, 1986.

[LST91] David Luckham, Sriram Sankar, and Shuzo Takahashi. Two-dimensional pinpointing: Debugging with formal specifications. *IEEE Software*, pages 74–84, January 1991.

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

[LW87] James R. Lyle and Mark Weiser. Automatic program bug location by program slicing. In *Proceedings of the Second International Conference on Computers and Applications*, Beijing, China, June 1987.

[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.

[Moh88] Thomas G. Moher. Provide: a process visualization and debugging environment. *IEEE Transactions on Software Engineering*, 14(6):849–857, June 1988.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, MA, 1990.

[NR69] P. Nauer and B. Randell, editors. *Software Engineering*. Scientific Affairs Div., NATO, Brussels, Belgium, January 1969.

[OCHW91] Ronald A. Olsson, Richard H. Crawford, W. Wilson Ho, and Christopher E. Wee. Sequential debugging at a high level of abstraction. *IEEE Software*, pages 27–36, May 1991.

[OF76] Leon J. Osterweil and Lloyd D. Fosdick. DAVE—a validation error detection and documentation system for Fortran programs. *Software Practice and Experience*, 6:473–486, 1976.

[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.

[OT89] L. Ott and J. Thuss. The relationship between slices and module cohesion. In *Proceedings of the Eleventh International Conference on Software Engineering*. IEEE, May 1989.

[Par85] D. L. Parnas. Software aspects of strategic defense systems. *Communications of the ACM*, 28(12):1326–1335, December 1985.

[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.

[RW88] Charles Rich and R. C. Waters. The Programmer's Apprentice: A research overview. *IEEE Computer*, pages 10–25, November 1988.

[RY88] Thomas Reps and W. Yang. The semantics of program slicing. Technical Report TR-777, Computer Science Department, University of Wisconsin, Madison, WI, June 1988.

[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.

[SD60] T. G. Stockham and J. B. Dennis. Flit—flexowriter interrogation tape: a symbolic utility for TX-O. Memo 5001-23, Dept. of Electical Engineering, Massachusetts Institute of Technology, July 1960.

[Sel89] R. P. Selke. A rewriting semantics for program dependence graphs. In *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages*, pages 12–24. ACM, January 1989.

[Sev87] Rudolph E. Seviora. Knowledge-based program debugging systems. *IEEE Software*, 4(3):20–32, May 1987.

[Sha83] Ehud Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, Cambridge, MA, 1983.

[SI91] Takao Shimomura and Sadahiro Isoda. Linked-list visualization for debugging. *IEEE Software*, pages 44–51, May 1991.

[SKF90] Nahid Shahmehri, Mariam Kamkar, and Peter Fritzson. Semi-automatic bug localization in software maintenance. In *Proceedings of the IEEE Conference on Software Maintenance*, San Diego, CA, November 1990. IEEE.

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

[Sta90] Richard M. Stallman. *Using and Porting GNU CC, version 1.37.* Free Software Foundation, Cambridge, MA, January 1990.

[STJ83] Robert L. Sedlmeyer, William B. Thompson, and Paul E. Johnson. Knowledge-based fault localization in debugging. In *Proceedings of the Software Engineering Symposium on High-Level Debugging*, Pacific Grove, CA, March 1983. ACM SIGSOFT/SIGPLAN. Software Engineering Notes, 8(4):25–31, August 1983; SIGPLAN Notices, 18(8):25–31, August 1983.

[SW65] R. Saunders and R. Wagner. On-line debugging systems. In *Proceedings of IFIP Congress*, volume 2, pages 545–546, 1965.

[Tei72] Warren Teitelman. Automated programming: The Programmer's Assistant. In *Afips Proceedings, Fall Joint Computer Conference*, volume 41, pages 917–921, Montvale, New Jersey, 1972. AFIPS Press.

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

[TR81] Tim Teitelbaum and Thomas Reps. The Cornell Program Synthesizer: a syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, September 1981.

[Ven91] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, Toronto, Ontario, Canada, June 1991. ACM SIGPLAN. SIGPLAN Notices, 26(6):107–119, June 1991.

[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.

[Zel78] Marvin V. Zelkowitz. Perspectives on software engineering. *ACM Computing Surveys*, 10(2):197–216, June 1978.

[Zel84] Polle T. Zellweger. Interactive source-level debugging of optimized programs. Technical Report CSL-84-5, Xerox Palo Alto Research Center, Palo Alto, CA, May 1984.

VITA

VITA

Hiralal Agrawal was born on June 6, 1963 in Motipur, India. He received his junior high school education at Bal Vidya Mandir, Lucknow, India and high school education at Vidya Niketan (Birla Public School), Pilani, India where he passed his Secondary School and Senior School Certificate Examinations in 1979 and 1981, respectively. He was placed on the All-India Merit Lists and awarded gold medals for securing the highest marks in the school on both these exams. Then he pursued his undergraduate studies at Birla Institute of Technology and Science (BITS), also in Pilani, India where he obtained his Master of Science (Technology) degree in Computer Science in 1985. There too he received a gold medal for securing the highest cumulative GPA—10 on a scale of 10—a first-ever achievement in the history of the institute. After graduating from BITS, he worked in the R & D Division of CMC Ltd, Secunderabad, India for one year before starting his graduate studies at Purdue University in August, 1986. He received his Master of Science and Doctor of Philosophy degrees in Computer Science at Purdue in 1988 and 1991, respectively. While at Purdue, he received the Purdue University Fellowship for the years 1986–1988 and was awarded the Maurice Halstead Award in Software Engineering in 1991.