An Execution Backtracking Approach to Program Debugging*

Technical Report SERC-TR-22-P

Hiralal Agrawal
Richard A. DeMillo
Eugene H. Spafford †

Software Engineering Research Center
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907–1398

Abstract

An execution backtracking facility in interactive source debuggers allows users to mirror their thought processes while debugging — working backwards from the location where an error is manifested and determining the conditions under which the error occurred. Such a facility also allows a user to change program characteristics and reexecute from arbitrary points within the program under examination — a "what-if" capability.

This paper describes an experimental debugger that provides such a backtracking function. We describe why the facility is useful, and why other current techniques are inadequate. We show how execution backtracking can be efficiently implemented by saving only the latest values of variables modified by a statement, and allowing backtracking only over *complete* program statements. We also describe how this approach relates to our work on dynamic program slicing.

Keywords: debugging, backtracking, reverse execution.

1 Introduction

The importance of good debugging tools cannot be overemphasized. Programmers spend considerable amounts of their program development time debugging. Several tools are available to help them in this task, varying from hexadecimal dumps of program state at the time of failure to window and mouse-based

---

*An early version of this paper appeared as [1]; a revised, later version appeared as [2].

This research was supported, 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), and by National Science Foundation Grant CCR–8910306.

†Authors' e-mail addresses: {ha,rad,spaf}@cs.purdue.edu.

interactive debuggers using bit-mapped displays. Most interactive debuggers provide breakpoints and traces as their main debugging aids. The work described in this paper involves the development of an execution backtracking facility to be used in association with breakpoints and traces as an aid in debugging. We believe that such a facility will be a significant addition to the debugging help provided by conventional debuggers today. Systems that have provided similar facilities in the past (e.g., [3, 4, 5, 6]) have all used a technique that, in general, may require an unbounded amount of storage. To overcome this space problem, some systems restrict execution backtracking only over a fixed number of most recent operations (see Section 4). In this paper, we also propose another approach to execution backtracking, called *structured backtracking*, that alleviates this problem.

1.1 How Does One Debug?

Given that a program has failed to produce the desired output, how does one go about finding where it went wrong? Other than the program source, the only important information usually available to the programmer is the input data (if any), 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 that it did[7].

Consider, for example, the following code segment:

```
S1:   read (X);
S2:   if (X < 0) then begin
S3:        Y := f1(X);
S4:        Z := f2(X);
      end
S5:   else if (X > 0) then begin
S6:        Y := g1(X);
S7:        Z := g2(X);
      end
      else begin (* X = 0 *)
S8:        Y := h1(X);
S9:        Z := h2(X);
      end;
S10:  write (Y);
S11:  write (Z);
```

The above code reads a value for X, and depending on the value read, whether $X < 0$, $X > 0$, or $X = 0$, it computes two values Y and Z as some functions of X. The functions themselves ($f1, f2, g1, g2, h1, h2$) are not relevant for the current discussion. Suppose this code is executed and we discover that the value printed for the variable Z is incorrect. Debugging the above code for this error would mean asking the following question: what caused the value of Z at statement S11 to be wrong?

Looking backwards from statement S11 we find that one of the three statements, S4, S7, or S9 last assigned a value to Z. We would have to examine the value read for X for the current testcase to determine which of these three statements actually assigned a value to Z. Suppose the value read for X was 3, so statements S6 and S7 were executed. Now there are several possibilities: there could be an error in the code for the function $g2$ used in statement S7; one of the predicates in statements S2 or S5 may be erroneous; or there could be an error in reading the value of X at statement S1. We may decide to set a breakpoint at statement S7 to check if the execution indeed reached there, and whether the value of X is correct before statement S7 is executed. If the breakpoint at statement S7 is reached and we find the value of X to be correct there, we may proceed to examine the code for the function g2.

There are three distinct tasks we performed above: 1) determining which statements in the code had an influence on the value of Z at statement S11, 2) selecting one (or more) of these statements at which to examine the program state, and then 3) restoring the program state at that statement. In this case, we performed the first two tasks ourselves by examining the code, without any assistance from the debugger. For the third task we had to set a breakpoint and *reexecute* the code until the control stopped at the breakpoint. Our debugging job would become much easier if the debugger provided direct assistance in performing all three tasks.

We have built a prototype debugging tool, named *Spyder*, to assist the user with all three of these tasks. The first — given a variable and a program location, determining which statements in the program really affected the value of that variable as observed at that location, when the program is executed for the given testcase (particular input values) — is referred to as *Dynamic Program Slicing* [8]. *Spyder* can automatically find dynamic slices for us. It is also possible for *Spyder* to help us make judicious decisions about which statements within the slice we should examine first, based on, e.g., how the program has "behaved" on other testcases. And last, it can also help us automatically restore the program state at any desired location by *backtracking* the program execution to that location without having to reexecute the program from the beginning. As it is not feasible to discuss all three of these functions in one paper, we discuss the last of the above tasks here: building execution backtracking support into debuggers. The reader is referred to [8] for a preliminary discussion of dynamic program slicing.

1.2 Why Backtrack While Debugging?

As we mentioned above, one logical way to go about debugging a program is to think backwards from the statement where the error is first manifested. Conceptually, this *backwards execution* can be mentally performed by analyzing the effect of each source statement, proceeding backwards one statement at a time. For large programs, such analysis at the statement level would be too tedious to perform. What we seek is to first narrow the user's focus to a small region of the program that is likely to contain an error, and then do statement level analysis within this region.

How can a user determine the small program region that is most likely to contain an error? If we consider small regions of the program as "logically atomic" blocks, then this problem is analogous to locating an erroneous statement in the program — only now the program is viewed as a sequence of logical blocks instead of individual statements. One need only to think backwards at the program block level instead of the statement level.

To perform such backward analysis using conventional interactive debuggers, one would first set a breakpoint just before the last logical block. If the program state is found to be correct at this point, it would imply that the error occurred within the last block. Otherwise another breakpoint would be set before the second-last block, and the program *reexecuted*. If the program state is found to be correct at that point, then we may conclude that the error resides within that second-last block. Otherwise this process of setting breakpoints in backward order and reexecuting the program continues until the erroneous block is discovered. If N is the number of blocks in the program, then clearly this method of setting breakpoints successively in backwards order and reexecuting the program every time leads to an $O(N^2)$ cost for execution. Using an execution backtracking facility, the backward debugging strategy discussed above would require only $O(N)$ block executions. This is because, from the point an error is manifested, program execution can be backtracked one block at a time, checking for errors in program state after each backtrack. The program would not have to be repeatedly reexecuted from the beginning for each block under examination.

The scenario presented above is obviously an idealized one. Most of the time, much more information about the program behavior, internal and external, is known to the programmers. With that information, the programmers do not always have to follow the rigid path outlined above. However, in large and complicated programs, especially those developed by teams of programmers and maintained by people who were not involved with the code development, strategies somewhat similar to the one mentioned above are often employed to detect the lurking bugs most frustrating in real systems development.

A backtracking facility within a debugger is also valuable in other situations besides its use in the debugging strategy outlined above. When debugging using conventional interactive debuggers, the user has to determine the regions that are likely to contain errors and then set breakpoints in those regions. Often, when the program execution is suspended at a breakpoint, the user discovers that the error occurred at an earlier location. In such cases, there is no other choice but to set another breakpoint at an earlier location and start the program execution again. With an execution backtracking facility, the program execution can simply be backtracked to the earlier location, and there is no need to rerun the program from the beginning.

As yet another motivating example, consider a user single-stepping through a program, observing its behavior or tracking down a bug. The user may “step over” a statement by mistake or by not realizing its importance until some statements later in the execution. In the absence of a backtracking facility, the only way to recover from this situation is to restart the program execution and take care that the same mistake is not made again. With an execution backtracking facility, any single-step command could easily be undone by simply backtracking the execution over the last statement executed.

If the debugger also allows the user to modify the program state, then the execution backtracking facility could be used to do some *what-if* analysis over sections of the program. Starting from a particular program state, the user could execute a section of the program, inspect the results of this execution, backtrack the execution to the same earlier state, change this state, and reexecute over that program section. Different execution paths taken by the program could be easily examined using this backtracking and state changing facility.

From these examples, it is apparent that having some form of backtracking function would be quite useful in a debugger. At the least, users will feel more comfortable and confident if they know they can undo their actions. At best, the presence of such a facility could make a significant difference in the time spent debugging a large amount of code.

In the next section we outline two approaches to implement statement-level execution backtracking in a simple language. Then in Section 3 we discuss how additional language features can be handled using the approaches outlined earlier. In Section 4 we discuss related work and ideas and explain how they are different or inappropriate for our purposes.

2 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 a language that only includes assignment, conditional (**if-then-else**), loop (**while-do**), and input-output (**read**, **write**) statements and their composition. Further let us assume 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 alter 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 — changing values of variables — can easily be undone if before executing a 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 2.2 we show that by constraining backtracking in a particular way, it can be implemented much more efficiently.

2.1 The Execution History Approach

With each assignment statement we associate a *change-set*. The change-set consists of all variables whose values are modified by the statement. For the simple language we mentioned above, the change-set of an assignment statement consists of the single variable that appears on the left-hand-side of the assignment.

For languages that allow expressions with side-effects, the change-set of an assignment statement may have more than one variable. The change-set of a **read** statement includes all variables read by the statement. Henceforth, we also refer to **read** statements as assignment statements. A **write** statement has an empty change-set. Similarly, change-sets of **if** and **while** predicates are empty-sets.

Consider, for example, the program fragment in Figure 1. This program reads two integers x and y ($x \geq 0, y > 0$), and finds their quotient, q, and remainder, r ($0 \leq r < y$), so that $x = q * y + r$. The change-set of statement S1 is {x, y} and that of S6 is {temp}. Statement S13, on the other hand, has an empty change-set.

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 change-sets. Then we can backtrack to any statement by restoring the previously saved values of change-set variables starting at the current location and going backwards until that statement is encountered in the saved execution history. For example, Figure 2 shows the execution history of the program in Figure 1 for the testcase ($x = 7$, $y = 3$).

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 change-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 change-set variable values may not be allocated in advance. Besides having this space problem, the above approach is also time

```
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 1: Program to divide two integers.

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.

2.2 The Structured Backtracking Approach

In the previous section we associated change-sets with assignment statements. We may also define the change-set of a composite statements like **if** or **while** to be the set of all those variables whose values *could* be modified during the execution of that statement. For example, the change-set of a **while** loop will consist of all variables that could be modified if the loop body is executed one or more times.

Change-sets of composite statements can be computed from the change-sets of their constituent assignment statements. If we denote the function computing the change-set of a statement by C, source statements by S, S_1, and S_2, and a boolean expression by $cond$, then the change-sets of some composite statements may be computed as follows:

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

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 2: Execution history of the program in Figure 1 for the testcase X = 7, Y = 3, along with the saved change-set values.

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

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

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

For example, the change-set of the **while** loop beginning at statement S7 in Figure 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 change-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 change-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* 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 new approach, for each statement — simple or composite — the debugger allocates space to save just one instance of values of all variables in its change-set. Any time control reaches that statement, the debugger saves the current values of variables in its change-set in the same space. So every time a statement in a loop body gets executed, the current values of variables in its change-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 loop body for the same iteration, but it is not possible to 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 this scheme:

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 valid instances of 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.

Note that the use of change-set restoration puts all involved variables back into their prior state. It is then possible to forward execute from that point, possibly after the user changes some data values. This provides the *what-if* capability referred to earlier.

Under this 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 change-set is fixed, so all the space required may be allocated in advance. In the next section we derive bounds on space requirements of this approach.

2.3 Bounds on Space Requirements

If an assignment statement is nested N levels deep, then the variables modified by it would belong to change-sets of N statements — $N - 1$ composite statements in which it is nested and the assignment statement itself (an assignment statement not nested in any composite statement is assumed to be at level one). Let A be the total number of assignment statements in the program, s_i be the size of the change-set of the ith assignment statement, n_i be the nesting level of the ith assignment statement, and S be the sum of sizes of change-sets of all statements in the program. Then we have:

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

Here,α represents the maximum nesting level in the program, and β represents the size of the largest change-set among all assignment statements in the program. Then, we have:

$$S \leq A \times \alpha \times \beta$$

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 $L - A + 1$. This means, $S \leq A \times (L - A + 1) \times \beta$. For a given L, the right-hand side of this equation is a function of A. The maximum of this function occurs at $A = \lceil L/2 \rceil$. For this value of A, we get $S \leq c_1 \times L^2$ for some constant c_1, giving us $S = O(L^2)$.

But this is only a theoretical worst-case upper bound. In practice, both α and β are usually small constants. Thus, usually $S \leq c_2 \times A$ for some small constant c_2, giving us $S = O(A)$. As the number of assignment statements in the program is also bounded by the program length, we also have $S = O(L)$. Thus, in the usual case, the sum of the sizes of the change-sets of all statements in the program is of the order of the length of the program. In particular, this size is independent of the length of the execution history, or the running time of the program.

3 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. Records are easy to handle: We simply need to treat each field of a record as a separate variable!

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. When such an assignment statement appears within a loop, the address of the memory location assigned may vary from one iteration to another. In this case, the precise change-set of the loop cannot 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 location assigned are added to the change-set of the loop. The size of the change-set of the loop, in this case, may not be bounded at compile time. But the space bounds derived in Section 2.3 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 loop 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 change-set of the call. Like a loop, when an indirect assignment is executed inside a procedure, the change-set of the procedure call is updated appropriately. Recursion is handled by saving change-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.

4 Related Work

Many systems have used variations of the execution history approach, outlined in Section 2.1, to provide some form of execution backtracking, and thus all have suffered from the same space problem. EXDAMS[3], an interactive debugging tool developed for FORTRAN in the late 60's, provided an execution replay facility where the program to be debugged is first executed in entirety and the complete execution history is saved. Then the program is "reexecuted" through a "playback" of this tape. This reexecution can be backtracked any time using the information saved on the tape. The user, however, cannot change values of variables before resuming forward execution because EXDAMS simply replays the program behavior recorded earlier.

Zelkowitz incorporated a backtracking facility within the programming language PL/1 by adding a RETRACE statement to the language[4]. With this statement, execution can 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. Incorporating backtracking facility within a programming language, however, does not provide *interactive* control over backtracking required during debugging, as the user must program the RETRACE statements along with the code in advance. INTERLISP[5] and the Cornell Program Synthesizer[6] also provide facilities to undo operations. All these systems maintain a history list of operations while recording their side effects. Thus they too suffer from the same space problem discussed in Section 2.1. To overcome this problem, they all use bounded history lists: 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 these systems.

IGOR[9] and COPE[10] 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, would be highly space-inefficient for performing statement-level backtracking.

5 Conclusions

An execution backtracking facility in interactive debuggers would be of significant help to programmers. With this facility, they would be able to match the debugging mechanism with their thought process of working backwards from the location where an error is manifested and determining conditions under which the error could occur. Previous systems that provided facilities for execution backtracking have all suffered from a space problem: they either had to maintain arbitrarily long execution histories, or had to restrict backtracking only over a fixed number of most recent events. We have proposed a new approach to execution backtracking called structured backtracking that alleviates these problems.

We have a prototype debugging tool, named *Spyder*, that incorporates backtracking as well as dynamic program slicing techniques. Figure 3 shows the main window panel of the tool in operation. The buttons labeled "backup" and "stepback" provide the reverse analogues of continuing forward execution and single-stepping (functions provided by buttons labeled "continue" and "step"), respectively. The current prototype works for the C programming language and has been implemented on a SUN SPARCstation 1 running SunOS 4 and the X Window System (version 11, release 4). Readers interested in more information or a copy of the prototype should contact E. Spafford.

References

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

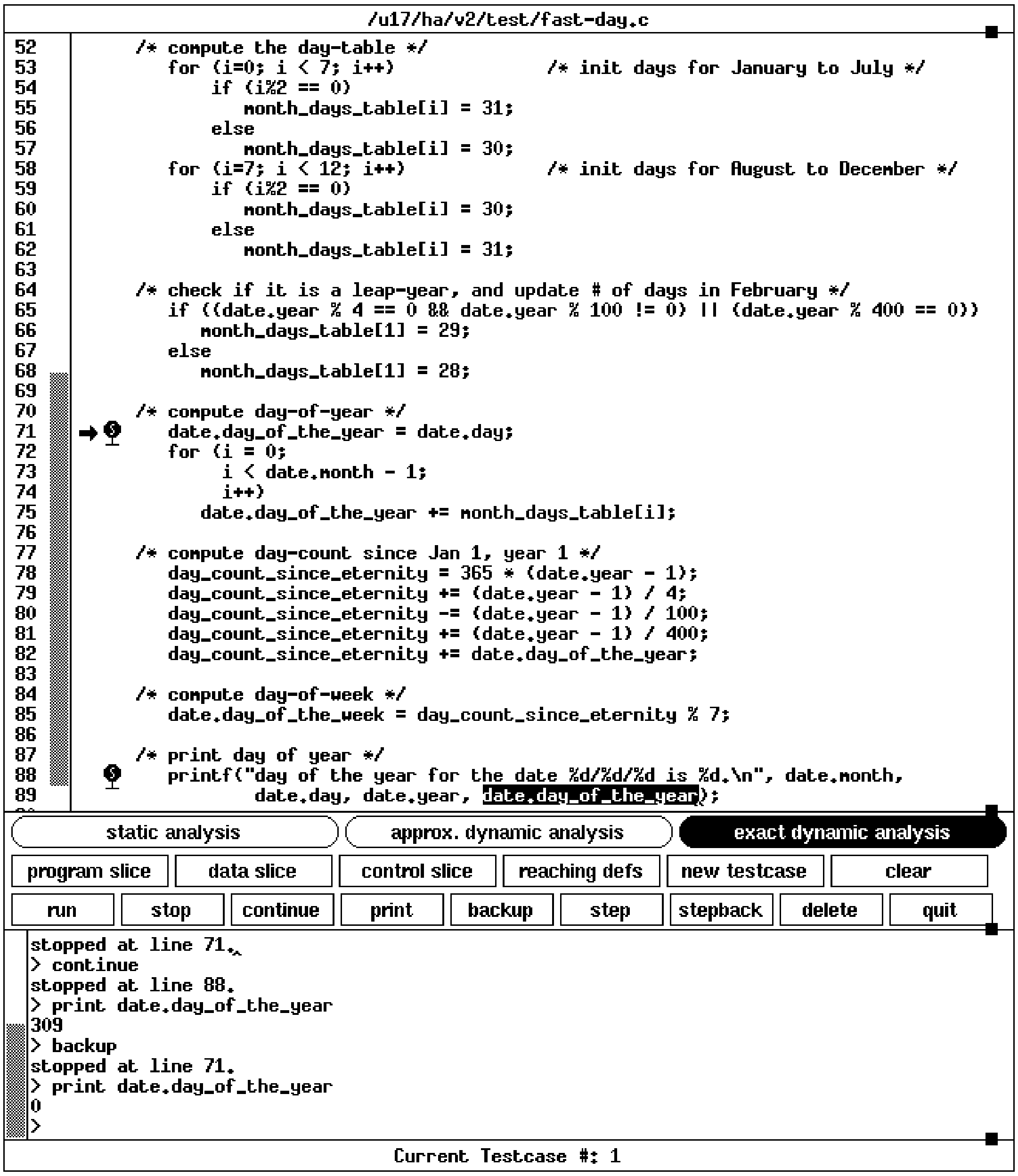

Figure 3: Main window panel of *Spyder*

[2] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. An execution backtracking approach to program debugging. *IEEE Software*, pages 21–26, May 1991.

[3] 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.

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

[5] Warren Teitelman and Larry Masinter. The Interlisp programming environment. *IEEE Computer*, pages 25–33, April 1981.

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

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

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

[9] 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.

[10] 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.