

Debugging with Dynamic Slicing and Backtracking

HIRALAL AGRAWAL

Bellcore, 445 South Street, Morristown, NJ 07962-1910, U.S.A.

AND

RICHARD A. DEMILLO AND EUGENE H. SPAFFORD

Software Engineering Research Center, Purdue University, W. Lafayette, IN 47907-1398, U.S.A.

SUMMARY

Programmers spend considerable time debugging code. Symbolic debuggers provide some help but the task 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 for a given test case, and what was the value of a given variable when the control last reached a given program location. If debugging tools provided explicit support for these tasks, the debugging process could be automated to a significant extent.

In this paper we present a debugging model, based on dynamic program slicing and execution backtracking techniques, that easily lends itself to automation. This model is based on experience with using these techniques to debug software. We also present a prototype debugging tool, SPYDER, that explicitly supports the proposed model, and with which we are performing further debugging research.

KEY WORDS: Program debugging Execution backtracking Reverse program execution Program slicing
Dynamic program slicing

1. INTRODUCTION

The importance of good debugging tools cannot be overemphasized. Average programmers may spend considerable amounts of their program development time debugging. Several tools are available to help them in this task,^{1,2} varying from hexadecimal dumps of program state at the time of failure to window- and mouse-based interactive debuggers using bit-mapped displays.^{3,4} Most interactive debuggers provide breakpoints, traces, and some facilities to examine the program state as their main debugging aids. Unfortunately, these traditional mechanisms are often inadequate for the task of quickly isolating specific program faults.

One approach to address the above problem is to provide users more control over actions they may take when a breakpoint or a tracepoint is reached, and to provide them with explicit mechanisms to construct high-level abstractions by correlating the low-level events.⁸ In this paper, we propose a different approach based on a new

debugging model. The importance of this model lies in the fact that each step in it can be largely automated, thus removing much of the tedium from the debugging process. It also provides a systematic approach to debugging, thus attempting to introduce an element of science into something that otherwise has largely been considered an art.

How does one debug?

Given that a program has failed to produce a desired output, how does one go about finding where it went wrong? Other than the program source, the only important information usually available to the programmer is the input data and the erroneous output produced by the program. If the program is sufficiently simple, it can be analyzed manually on the given output. For many large programs, however, 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.

Consider, for example, the program in the main window panel of [Figure 1](#).^{*} For now, ignore the fact that some lines of the program have been highlighted. 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 its area 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 test case[†] when $N = 2$ and sides of the two triangles are (3, 3, 3) and (6, 5, 4), respectively. The sum of the areas of the two triangles for this test case is incorrectly printed as 13.90 instead of 13.82, thereby implying the presence of a fault in the program. How should we go about locating this bug? Looking backwards from the `printf` statement on line 46, we find there are several possibilities: `sum` is not being updated properly; one (or more) of the formulas for computing the area of a triangle is incorrect; the triangle is being classified incorrectly; or the values for the three sides of the triangle are not being read correctly.

The statement on line 43 adds `area` to `sum`, so that the first thing we may want to do is to examine the program state at that point. We can set a breakpoint at that line and re-execute the program up to that statement to examine the values of variables `sum` and `area` at that point. Suppose we find that the area is computed correctly during the first loop iteration, but is computed incorrectly during the second iteration. To discover why, we may wish to examine the values of `class` for the second triangle to determine which formula for computing the area was used. If we discover `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:

^{*} The figures are X Window System window dumps of our prototype debugging tool, SPYDER, in operation.

[†] A test case consists of a specific set of run-time input values.

We will refer to this test case as the test case no. 1 in later sections.

```

                                /u17/ha/v2/demo/example.bug.c
1  /* Find the sum of areas of given triangles. */
2  #define MAX 100
3  typedef enum {isosceles, equilateral, right, scalene} class_type;
4  typedef struct {int a, b, c;} triangle_type;
5
6  main()
7  {
8      triangle_type sides[MAX];
9      class_type class;
10     int a_sqr, b_sqr, c_sqr, N, i;
11     double area, sum, s, sqrt();
12
13     printf("Enter number of triangles:\n");
14     scanf("%d", &N);
15     for (i = 0; i < N; i++) {
16         printf("Enter three sides of triangle %d in ascending order:\n", i+1);
17         scanf("%d %d %d", &sides[i].a, &sides[i].b, &sides[i].c);
18     }
19
20     sum = 0;
21     i = 0;
22     while (i < N) {
23         a_sqr = sides[i].a * sides[i].a;
24         b_sqr = sides[i].b * sides[i].b;
25         c_sqr = sides[i].c * sides[i].c;
26         if ((sides[i].a == sides[i].b) || (sides[i].b == sides[i].c))
27             class = equilateral;
28         else if ((sides[i].a == sides[i].b) || (sides[i].b == sides[i].c))
29             class = isosceles;
30         else if (a_sqr == b_sqr + c_sqr)
31             class = right;
32         else class = scalene;
33
34         if (class == right)
35             area = sides[i].b * sides[i].c / 2.0;
36         else if (class == equilateral)
37             area = sides[i].a * sides[i].a * sqrt(3.0) / 4.0;
38         else {
39             s = (sides[i].a + sides[i].b + sides[i].c) / 2.0;
40             area = sqrt(s * (s - sides[i].a) * (s - sides[i].b) *
41                 (s - sides[i].c));
42         }
43         sum += area;
44         i += 1;
45     }
46     printf("Sum of areas of the %d triangles is %.2f.\n", N, sum);
47 }
48
49
50

```

static analysis
 approx. dynamic analysis
 exact dynamic analysis


```

> delete breakpoint at line 35
> continue
stopped at line 43.
> clear
> clear
> dynamic program slice on "area" at line 43
> ^

```

Current Testcase #: 1

Figure 1. Dynamic program slice with respect to area on line 43 during the second loop iteration for test case no. 1

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 *re-execute* the code until the control stopped at the breakpoint. Our debugging job would become much easier if the debugger provided direct assistance in performing all three of these tasks. With explicit support for these activities, we will more easily be able to pursue software debugging in a systematic fashion.

We have built a prototype debugger, named SPYDER, that provides the user with exactly this assistance. The first task—given a variable and a program location, determining which statements in the program affected the value of that variable at that location when the program is executed for a given test case—is referred to as *dynamic program slicing*.^{12,13} SPYDER can find dynamic slices for us automatically. It can also restore the program state at any desired location by *backtracking* the program execution without having to re-execute the program from the beginning.¹⁴ In this paper we discuss these two functions, slicing and backtracking, combined in our prototype debugging tool.

Figure 2 depicts our proposed debugging model. It may be thought of as a

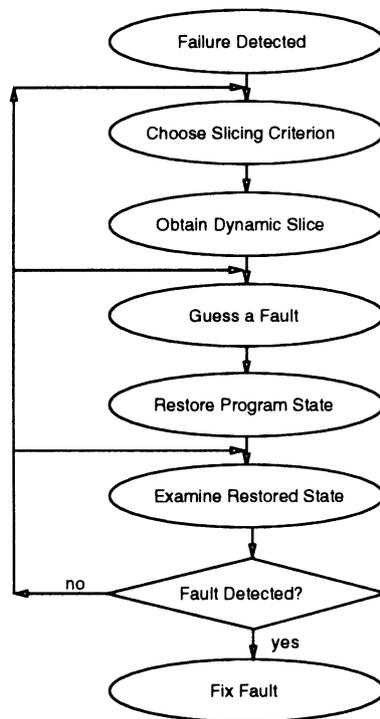


Figure 2. A debugging model

specialization of the general model of the debugging process proposed by Araki, Furukawa and Cheng.¹⁵ The first step of the model after 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 the state when the 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, guess a new fault, or select a new slicing criterion and repeat the cycle until the fault is localized.

In the next section we examine various notions of dynamic slices and show how SPYDER supports them. In Section 3 we discuss the execution backtracking facility of SPYDER. Then in Section 4 we present an example debugging session with SPYDER using both slicing and backtracking facilities. In Section 5 we briefly discuss some implementation issues. Section 6 concludes the paper with a discussion of some of the limitations of the techniques described here and outlining some of the lessons learned from this experience.

2. DYNAMIC SLICING

The concept of program slicing was first proposed by Weiser.^{16,17} But this original notion of a program slice envisioned a static slice obtained irrespective of the values of the input variables. A static program slice with respect to a given variable occurrence evaluates the variable occurrence identically to the original program when executed for any test case. Weiser also presented an algorithm to compute static slices based on an iterative solution of data-flow equations.¹⁷ Ottenstein and Ottenstein later presented an algorithm to find static program slices based on graph reachability in the program dependence graph, but they only considered the intra-procedural case.¹⁸ Horwitz, Reps and Binkeley have proposed extending the program dependence graph representation to what they call the ‘system dependence graph’ to find interprocedural static slices using the same graph-reachability framework.¹⁹ Bergeretti and Carré have also defined information-flow relations somewhat similar to data and control dependence relations, that can be used to obtain static program slices (referred to as ‘partial statements’ by them).²⁰ Several studies investigating the semantic basis of program slicing have also been reported. Uses of program slicing have been suggested in many applications, e.g. program verification, maintenance, automatic parallelization of program execution, automatic integration of program versions, software metrics, etc.^{17,20} In this paper, we are primarily concerned with its use in debugging.

Often during debugging, the value of a variable, *var*, at a program statement, *S*, is observed to be incorrect. A static program slice with respect to *var* at *S* identifies the relevant subset of the program that one should examine to determine the cause of the error. But, as mentioned earlier, the static notion of a program slice does not make any use of the input values that reveal the error. It is concerned with finding statements that *could* affect the value of a variable occurrence for any input, not statements that *did* affect its value for a specific input. During debugging, programmers generally analyse the program behavior under the test case that revealed the

error, not under a generic test case. The concrete test case that exercises the bug helps them focus their attention on the ‘cross-section’ of the program that contains the fault.*

Korel and Laski extended the static notion of a program slice to the dynamic case that also takes into consideration the test case that reveals a fault.²⁹ They define a dynamic slice with respect to a variable occurrence and a test case to be a subset of the program that evaluates the variable occurrence identically to the original program when executed for the given test case. Unfortunately, the requirement that a dynamic slice be executable for the given test case may cause many statements that do not really ‘affect’ the value of the variable occurrence in question to be included in the dynamic slice. The extra statements included solely for the purpose of making the dynamic slice executable may add considerable noise, and in some cases may be larger than the statements really needed for the simple dynamic slice. We believe that from the debugging perspective, the value 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 affect the value of a particular variable at a particular program location.

For example, in the program in [Figure 1](#), each loop iteration computes the value of the variable `area`. Note that the computation of `area` during one iteration is totally independent of the computation performed during any other iteration. If the value of `area` at the end of one iteration is found to be incorrect and we obtain the dynamic slice with respect to `area` at the end of that iteration, we would like only those statements to be included in the slice that affected its value at the end of that iteration, not during all previous iterations. On the other hand, if we find that the value of the variable `sum` is incorrect at the end of some iteration, and we obtain the corresponding dynamic slice, we would not only want to see those statements that affected it during that iteration but also those that affected it during all the previous iterations. This is because unlike the value of `area`, the value of `sum` at the end of one iteration is also affected by computations performed during previous iterations. Korel and Laski’s definition of a dynamic slice fails to make the above important distinction.

Agrawal and Horgan independently proposed another notion of a dynamic slice that does not have the above-mentioned behavior.¹² For example, [Figure 1](#) shows the dynamic slice with respect to `area` at the end of the second loop iteration for the test case no. 1 according to their definition. Note that only the statements that affect the `area` of the second triangle are included in the slice. Korel and Laski’s definition, on the other hand, would have required the statements on lines 27, 36, and 37, that affect the `area` of the first triangle, to also be included in that slice even though they do not in any way affect the `area` of the second triangle.

A dynamic program slice presents a summary of the relevant computation by simply showing the relevant statements involved. The underlying reasons, however, for selecting those statements are not at all apparent by looking at the slice. In fact, for many large programs, looking at the complete program slice may often be overwhelming. Thus, knowing the analysis performed to arrive at the program slice

* When we say the slice contains the fault, we do not necessarily mean that the fault is textually contained in the slice; the fault could correspond to the absence of something from the slice—a missing `if` statement, a statement outside the slice that should have been inside it, etc. We can discover that something is missing from the slice only after we have found the slice.

is often crucial in determining the fault. Our prototype tool, SPYDER, not only lets us see the global view of the relevant computation in terms of the dynamic program slice, but also lets us obtain various local views in terms of the data and control slices, or even finer views in terms of the reaching definitions and controlling predicates. These local views enable us to ‘see’ the rationale used in arriving at the global view. The remainder of this section derives the notion of a dynamic program slice in the bottom-up manner and shows how SPYDER provides explicit support to make this derivation visible. The analysis presented below also is useful in understanding the purpose and the usefulness of the various functions provided by SPYDER.

2.1. Local analysis

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

- Case 1. The value of an expression, exp , at some program location, L , is observed to be incorrect.
- Case 2. Control has incorrectly reached a location, L , it should never have reached.
- Case 3. Control did not reach the desired location, L .

We now examine each of these three cases in detail.

Case 1

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

- 1.1. The 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.
- 1.2. The value of at least one variable, var , used in exp is incorrect, e.g. the value of the expression $x + y$ is incorrect because the value of x is incorrect. In this case, we must find the statement that last assigned a value to var , i.e. 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.*

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

- 1.2.1. The value of the expression, exp' , computed at R , is incorrect. In this case, we are back to Case 1 with respect to the value of exp' computed at R .
- 1.2.2. R is the wrong reaching definition of var . If this is the case, there are four further possibilities:
 - 1.2.2.1. Control should not have reached R , in which case we are back to Case 2 with respect to the location of R .
 - 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

* 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.

L , or one of the assignments along this path has incorrectly assigned a value to the wrong variable. If we have determined this, we have localized the fault.

- 1.2.2.3. Control correctly reached R , but it did not reach the correct definition, R' , 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 R' .
- 1.2.2.4. R should not 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 the control should not have reached the location L , there are two possibilities: L is immediately enclosed by a predicate, $pred$, or there is no predicate enclosing L :

- 2.1. L is immediately enclosed by a predicate, $pred$, e.g. an if-then-else or a while predicate. Again, there are two possibilities:
 - 2.1.1. Control should not have reached $pred$ either. Then we are back to Case 2 with respect to the location of $pred$.
 - 2.1.2. Control should have reached $pred$ but not L . In this case, there are two more possibilities:
 - 2.1.2.1. $Pred$ evaluated incorrectly, in which case we are back to Case 1 with respect to the value of $pred$.
 - 2.1.2.2. There is a missing predicate* along the path between $pred$ and L . If we have found this, we have localized the fault.
- 2.2. L is not enclosed by any predicate. If the control should not have reached L , it should have been enclosed by a predicate that would have prevented the control from reaching it. This means there is a missing predicate enclosing L . In this case, again, we have localized the fault.

Case 3

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

- 3.1. Control did not reach $pred$ either. In this case, we are back to Case 3 with respect to the location of $pred$.
- 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 $pred$.
- 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.

* Or a missing goto. For simplicity in analysis, we will assume that the program has no unconditional flow-of-control operations.

2.2. Global analysis

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

1. An assignment statement is found to compute an incorrect function.
2. A predicate expression is found to compute an incorrect function.
3. An assignment statement is found to assign a value to the wrong variable.
4. The desired assignment to a variable is missing.
5. The desired predicate expression guarding a given statement is missing.
6. An extraneous assignment is found to be present in the program.
7. An extraneous predicate is found to be present in the program.

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

1. We find the dynamic reaching definition of a variable at a given location.
2. We find the predicate that immediately encloses a given location.

Sometimes, the fault localization may be expedited if, in the above analysis, we combine 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:

1. 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.
2. An assignment whose computation should have propagated into the value of exp at L never got executed.
3. An assignment whose computation should have propagated into the value of exp at L is missing from the program.
4. An assignment whose computation did propagate into the value of exp at L should never have been included in the program.
5. An assignment whose computation propagates into the value of exp at L got executed an incorrect number of times (more or less than necessary).

Case 2 revisited

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

1. One of the predicates enclosing the given location has evaluated incorrectly.
2. There is a missing predicate that should enclose the given location (not necessarily enclosing it immediately; it may be missing at some 'outside' nesting level).

Case 3 revisited

If the control did not reach a given location, L , there are again two possibilities:

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

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 the dynamic reaching definitions of the variables used in exp at the given location. We call the set of all assignments that belong to this closure the *dynamic data slice* with respect to the given expression, location, and the test case. 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 from the program, or if one of the assignments in it computes an incorrect function. Similarly, by examining the dynamic data slice, we can also check if one of the assignments that should not 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 closer to finding the fault: we should continue our search using Cases 2 or 3 of the global analysis with respect to 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 must resort to the local analysis described above to localize such faults.

Figure 3 shows the dynamic data slice with respect to sum at line 46 for test case no. 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 test case are right triangles. This suggests that we should pursue Case 2 with respect to line 35.

Dynamic control slice

In Cases 2 and 3 of the global analysis, we need to find all predicates that enclose a given location. This can be done by taking the transitive closure of the enclosing predicates starting with the given location. We call the set of all predicates that belong to this closure the *dynamic control slice* with respect to the given location. If we know that the 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 the global analysis with respect to the incorrect argument.

```

/u17/ha/v2/deno/example.bug.c
1  /* Find the sum of areas of given triangles. */
2  #define MAX 100
3  typedef enum {isosceles, equilateral, right, scalene} class_type;
4  typedef struct {int a, b, c;} triangle_type;
5
6  main()
7  {
8      triangle_type sides[MAX];
9      class_type class;
10     int a_sqr, b_sqr, c_sqr, N, i;
11     double area, sum, s, sqrt();
12
13     printf("Enter number of triangles:\n");
14     scanf("%d", &N);
15     for (i = 0; i < N; i++) {
16         printf("Enter three sides of triangle %d in ascending order:\n", i+1);
17         scanf("%d %d %d", &sides[i].a, &sides[i].b, &sides[i].c);
18     }
19
20     sum = 0;
21     i = 0;
22     while (i < N) {
23         a_sqr = sides[i].a * sides[i].a;
24         b_sqr = sides[i].b * sides[i].b;
25         c_sqr = sides[i].c * sides[i].c;
26         if ((sides[i].a == sides[i].b) && (sides[i].b == sides[i].c))
27             class = equilateral;
28         else if ((sides[i].a == sides[i].b) || (sides[i].b == sides[i].c))
29             class = isosceles;
30         else if (a_sqr == b_sqr + c_sqr)
31             class = right;
32         else class = scalene;
33
34         if (class == right)
35             area = sides[i].b * sides[i].c / 2.0;
36         else if (class == equilateral)
37             area = sides[i].a * sides[i].a * sqrt(3.0) / 4.0;
38         else {
39             s = (sides[i].a + sides[i].b + sides[i].c) / 2.0;
40             area = sqrt(s * (s - sides[i].a) * (s - sides[i].b) *
41                 (s - sides[i].c));
42         }
43         sum += area;
44         i += 1;
45     }
46     printf("Sum of areas of the %d triangles is %.2f.\n", N, sum);
47 }
48
49
50
static analysis  approx. dynamic analysis  exact dynamic analysis
program slice  data slice  control slice  reaching defs  new testcase  clear
run  stop  continue  print  backup  step  stepback  delete  quit
stopped at line 47.
> stop at line 46
> backup
stopped at line 46.
> select exact dynamic analysis
> dynamic data slice on "sum" at line 46
>
Current Testcase #: 1

```

Figure 3. Dynamic data slice with respect to sum on line 46 for test case no. 1

For example, consider again the program in Figure 3. When it is executed for test case no. 1, we find that during the second loop iteration, the control incorrectly reaches the statement on line 35. The corresponding control slice contains the two predicates on lines 22 and 34. Examining the slice reveals that the statement is incorrectly reached because the second predicate above evaluates incorrectly: it

evaluates to true instead of false because the value of class examined by the predicate is incorrect at that instance. This suggests that we pursue Case 1 with respect to class on line 34.

Dynamic program slice

Note that during the global analysis discussed above, we alternate between the data and control slices until we have localized a fault. Often, a fault manifests itself several levels of indirection away from the fault itself. In such situations it may be possible to localize the fault more quickly if we knew the ‘sum’ of all the 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 the 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 dynamic control slice. Next, for each predicate in the control slice we find its dynamic data slice, and so on, until we reach the situation when no new statements can be added to this set. We call the set of statements so obtained the *dynamic program slice* with respect to the given expression at the given location for the given test case. Similarly, if we find that the control has incorrectly reached a given location or if it did not reach the desired location, we repeat the above analysis but this time starting with the control slice of the relevant location. We call the resulting set the dynamic program slice with respect to the given location.

Thus a dynamic program slice is the closure of all the data and control slices with respect to all expressions and locations in its constituent dynamic data and control slices. Figure 1, as described earlier, shows the dynamic program slice with respect to area at line 43 when the execution is stopped there during the second loop iteration for test case no. 1. Note that the erroneous assignment on line 24 is included in the slice. Also note that the assignment on line 37 that computes the area of the first triangle is not included in the slice because the computation of area during one loop iteration does not affect that during another iteration in this case. 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.

The above discussion suggests that, depending on the size and complexity of the program, and on the extent to which the person debugging the program is familiar with the program code, sometimes it may be more efficient to perform the local step-by-step analysis, whereas at other times examining the data, control, or the program slice may expedite the fault localization. SPYDER provides mechanisms to allow local as well as global dynamic analysis to be performed.

After a program is written, it is usually run against several input data sets designed to test specific aspects of the program behavior. If the program works correctly on one test case but fails on another, it may be helpful to analyze the program behavior under both these test cases. The inclusion or exclusion of a statement in dynamic slices with respect to the two test cases may provide valuable debugging clues. The same idea may be generalized to examining the dynamic program behavior under several test cases. SPYDER allows the user to save several test cases and select any test case for the dynamic analysis at any time.

We are currently building another experimental version of SPYDER that also

supports facilities to show the differences in dynamic slices with respect to two or more test cases. This version also provides support for the dynamic analogue of the static *program dicing* technique proposed in Reference 30. A dynamic program slice uses the fact that the value of a variable at a given program location is incorrect under a test case to narrow down the search for the fault. It does not, however, use the information that the value of another variable at the same location may be correct for the same test case. A dynamic program dice attempts to further narrow the search for the fault by removing the statements that belong to the dynamic slice with respect to the correct variable from those that belong to the dynamic slice with respect to the incorrect variable. Our experimental version also provides facilities to save the currently displayed slice, obtain another slice and perform operations such as difference and intersection with the saved slice. The results of these operations are then displayed and used as an aid in narrowing the search for program faults. Thus, in effect, we give the user the ability to obtain dynamic program dices (and more). The interested reader is referred to References 31 and 32 for details of the above and some other extensions of the program dicing approach.

3. EXECUTION BACKTRACKING

As described earlier in Section 1, once we have obtained a dynamic slice with respect to an erroneous variable and/or location, we are either able to identify the fault, or we choose one or more statements in the slice at which to examine the program state. In the latter case, if we were debugging using a conventional debugger, we would need to set a breakpoint at the selected statement and re-execute the program from the beginning. When the execution is suspended at the breakpoint, the program state there can be examined. After examining some variable values, we may discover that the error occurred at an earlier location. We may then decide to set another breakpoint at an earlier statement and restart program execution. We may need to repeat this process of setting breakpoints in backward order and re-executing the program several times before we are able to localize the fault. For large programs such repeated program execution may be very cumbersome.

SPYDER provides an execution backtracking facility with which the program state can be restored to an earlier location without having to re-execute the entire program. 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 wish to examine the program state at some earlier 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 during the forward execution.

For example, consider again the program in [Figure 4](#) and test case no. 1. If the program execution is stopped at line 46, and we discover the value of sum is incorrect there, we may set a breakpoint on line 43 and start the backward execution. As the loop was iterated twice for this test case, the second iteration will be reached first during the 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 the forward execution. If we examine the value of sum there, we will get its value

```

                                /u17/ha/v2/demo/example.bug.c
11      double area, sum, s, sqrt();
12
13      printf("Enter number of triangles:\n");
14      scanf("%d", &N);
15      for (i = 0; i < N; i++) {
16          printf("Enter three sides of triangle %d in ascending order:\n", i+1);
17          scanf("%d %d %d", &sides[i].a, &sides[i].b, &sides[i].c);
18      }
19
20      sum = 0;
21      i = 0;
22      while (i < N) {
23          a_sqr = sides[i].a * sides[i].a;
24          b_sqr = sides[i].b * sides[i].b;
25          c_sqr = sides[i].c * sides[i].c;
26          if ((sides[i].a == sides[i].b) && (sides[i].b == sides[i].c))
27              class = equilateral;
28          else if ((sides[i].a == sides[i].b) || (sides[i].b == sides[i].c))
29              class = isosceles;
30          else if (a_sqr == b_sqr + c_sqr)
31              class = right;
32          else class = scalene;
33
34          if (class == right)
35              area = sides[i].b * sides[i].c / 2.0;
36          else if (class == equilateral)
37              area = sides[i].a * sides[i].a * sqrt(3.0) / 4.0;
38          else {
39              s = (sides[i].a + sides[i].b + sides[i].c) / 2.0;
40              area = sqrt(s * (s - sides[i].a) * (s - sides[i].b) *
41                      (s - sides[i].c));
42          }
43          sum += area;
44          i += 1;
45      }
46      printf("Sum of areas of the %d triangles is %.2f.\n", N, sum);
47  }
48

```

static analysis			approx. dynamic analysis			exact dynamic analysis		
program slice	data slice	control slice	reaching defs	new testcase	clear			
run	stop	continue	print	backup	step	stepback	delete	quit

```

> select exact dynamic analysis
> stop at line 46
> run on testcase 1
stopped at line 13.
> continue
stopped at line 46.
> print sum
13.89711431702997
> dynamic reaching defs of "sum" at line 46
> clear
> stop at line 43
> backup
stopped at line 43.
> print sum
3.897114317029974
> print area
10
> dynamic program slice on "area" at line 43
> ^

```

Current Testcase #: 1

Figure 4. Tool screen after backtracking from line 46 to line 43

just before the last assignment was executed, as shown in the Figure. Notice the value of sum before and after the 'backup' command, as displayed in the bottom (output) window. 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 43 during the previous iteration.

Most debuggers provide facilities to let the user step forward through the program execution one line at a time. But during debugging, as described earlier, we often ‘think backwards’ from the location where a fault is manifested, so it would be helpful if the debugger also lets the user step-back through the program execution, statement by statement. SPYDER also provides such a back-stepping facility. When execution is stopped at a breakpoint, issuing the step-back command undoes the effect of the last statement executed.*

Execution backtracking is easily implemented if the tool already has mechanisms to support dynamic slicing. While recording the execution path of the program to enable the dynamic slicing (we will discuss this in Section 5), SPYDER also records the previous values of variables modified by each statement. Then, backtracking over a statement simply requires restoring the previously saved values of the variables modified by that statement. SPYDER, however, currently does not support restoration of activation records of completed procedure calls. Thus it does not allow backtracking *into* a procedure from outside it. This, however, may not be a serious problem as one can backtrack completely over a procedure call and then forward execute into it up to the desired location.

The above execution backtracking approach is similar to the execution replay facility in EXDAMS, an interactive debugging tool for Fortran developed in the late 1960s.³³ In that system, first the complete history tape of the program being debugged under a test case is saved. Then, the program is ‘executed’ through a ‘playback’ of this tape. At any point, the program execution can be backtracked to an earlier location using the information saved on the history tape. Under this approach, however, when the control is stopped at some program location it is not possible to change values of variables before executing forward again because EXDAMS simply replays the program behavior recorded earlier.

Miller and Choi’s *ppd* also performs flow-back analysis like EXDAMS but it uses a notion of incremental tracing.³⁴ Under that approach, instead of saving the values of changed variables at the statement level, they are saved only at the start and end of the program segments called emulation blocks. Then, during debugging, finer traces of execution within the emulation blocks are obtained, on demand, by re-executing the corresponding blocks using the values of the variables saved at the start of the blocks. This approach implicitly assumes that the memory location referenced by an l-value expression does not change during program execution within a block. But in the presence of pointer variables and dynamic memory allocations, the above assumption may not hold true. Thus, this approach may not work well in such situations.

Wilson and Moher’s demonic memory approach also involves hierarchical checkpointing to allow regeneration of program history at multiple levels of resolution.³⁵ But unlike Miller and Choi’s approach that relies on static analysis to determine which variables may be changed during execution, their approach relies on dynamic detection of changes to the process memory. Thus it is also able to handle pointers

* SPYDER indicates the next statement to be executed by an arrow in the left margin of the code window. See, e.g., Figure 4.

and dynamic memory allocations, like our approach. It is not clear, however, if it would be suitable in the context of the fine-grained backtracking required to support features like the reverse single stepping facility described above.

Alternatively, it is also possible to use the structured backtracking approach described in Reference 14. This approach provides the user with almost the same capabilities, but requires much less storage. Additionally, the storage requirements may be bounded at compile time in most cases. If we were using structured backtracking in our example above, we would not have been able to backtrack from the second iteration of the loop to the first. Rather, we would have had to backtrack to the beginning of the loop itself and execute forwards up to the given statement. This may also be implemented in a manner transparent to the user.

Other approaches to backtracking have also been proposed in different contexts. Zelkowitz incorporated a backtracking facility within the programming language PL/1 by adding a RETRACE statement to the language.³⁶ That approach, however, does not provide an interactive control over backtracking during debugging. INTERLISP³⁷ and the Cornell Program Synthesizer³⁸ also provide facilities to undo operations. These systems maintain a fixed-length history of side-effects caused by operations. As new events occur, the existing events on the list are aged, with oldest events 'forgotten'. Thus backtracking to locations arbitrarily far back in the execution may not be possible with these systems. IGOR³⁹ and COPE⁴⁰ also provide execution backtracking by performing periodic checkpointing of memory pages and file blocks, respectively, modified during program execution. This approach, while suitable for undoing effects of complete programs, may be inefficient to support statement-level backtracking.

4. AN EXAMPLE DEBUGGING SESSION WITH SPYDER

Let us illustrate the usage of SPYDER as well as the proposed model by presenting a small debugging session. We will again use the program in Figure 4 and the test case no. 1 ($N = 2$, and the sides of the two triangles being (3, 3, 3) and (6, 5, 4), respectively) from our discussion above. When the program is executed for this test case, the final value of sum is printed as 13.9 instead of the correct value, 13.82 (the area of the first triangle being 3.9 and that of the second being 9.92).

We first enable exact dynamic analysis by clicking on the toggle button labeled *exact dynamic analysis*.^{*} We then set a breakpoint on line 46 and run the program for the above test case. When the execution stops at the breakpoint,[†] we select the variable sum using the mouse and print its value by clicking on the *print* button. The incorrect value, 13.9, is printed.

We then click on the *reaching defs* button to find the last definition of sum and find that the assignment of line 43 last assigned a value to sum. We set another breakpoint on that line and click on the *backup* button to restore program state to what it was just before the assignment on that line was last executed. We then print

^{*} SPYDER also provides support to obtain static and approximate dynamic slices. The approximate dynamic slices are computationally less expensive to obtain compared to the exact dynamic slices, but they may also contain extraneous statements that do not really affect the value of the variable in question. The interested reader is referred to References 12 and 31 for a detailed discussion on various approaches to computing exact as well as approximate dynamic slices and the relative trade-offs involved.

[†] Represented by the little stop sign. See, e.g., Figure 4.

the value of `sum` at that location, and find that its correct value at that point, 3.9, is printed.

Thus far, `sum` contains only the area computed during the previous iteration. This implies that area was correctly computed for the first triangle and that it must be wrong for the second triangle. We print the current value of `area` and find that it is indeed incorrect—10.0 instead of 9.92. We next click on the *program slice* button to obtain the dynamic program slice for the current value of `area`, as shown in Figure 4. To our surprise, we find that the assignment on line 35 that computes the area of a right triangle belongs to the slice instead of that on line 40 that computes the area of a scalene triangle. We also find that the assignment on line 31 that assigns `right` to `class` is in the slice.

To check if the values of the sides of the triangle are being read correctly, we print `sides[i]` and find that the current values of the three sides of the triangle are indeed correct—(6, 5, 4). Looking at the program slice, we notice that the `if` statement on line 30 determines if the triangle is a right triangle, and if so, it sets the value of `class` accordingly. Therefore, we set another breakpoint at line 30 and further backup execution to that point. We next print the values of `a_sqr`, `b_sqr`, and `c_sqr`, and find that the value of `b_sqr` is printed incorrectly—20 instead of 25. Next, examining the data slice of `b_sqr` reveals the error on line 24.

5. IMPLEMENTATION

SPYDER is comprised of a modified version GCC, the GNU C compiler,⁴¹ and GDB, the GNU source-level debugger.⁴² We chose the GNU tools because of their easy availability and their ability to run on different hardware platforms. Although this choice has led to some problems (see Section 6.2), it has allowed us to rapidly develop a prototype and experiment with the techniques described earlier. Figure 5 gives a high-level view of the structure of SPYDER and shows how the modified GNU tools fit into it. In the next two subsections we briefly discuss the changes we made to these tools and how the changes help SPYDER provide its functions.

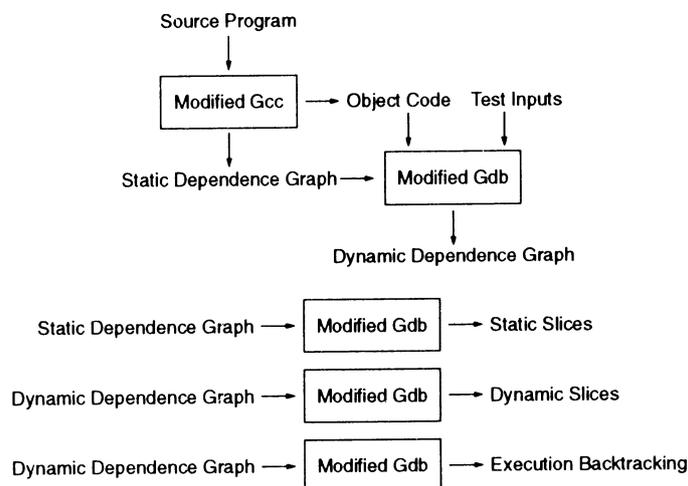


Figure 5. A high-level view of the structure of SPYDER

5.1. Modifications to the compiler

We modified GCC to produce the program dependence graph along with the object code of the given program. This simply required making changes to the parser. The modified parser, apart from its normal functions, also builds the control flow graph of the program in a syntax-directed manner. Each node in the graph is also annotated with its *use* and *def* sets in terms of the *l*-valued expressions used and defined, respectively, by the node. These expressions are later used by the debugger to determine the actual memory cells used and defined by the corresponding node during execution (see the next subsection). After the parsing is complete, the flow graph is traversed to compute the data and control dependencies among its nodes, and two more sets of edges belonging to the data and control dependence graphs, respectively, are created. The three graphs—flow graph, data dependence graph, and the control dependence graph—share the same nodes but have their own edges. The aggregate graph consisting of these three graphs is then written out to a file. It is later read and used by the debugger to determine static slices. The reader is referred to References 31 and 43 for details of the algorithms used to compute the data and control dependencies.

5.2. Modifications to the debugger

GDB was modified to read the aggregate graph consisting of the flow, control, and data dependence subgraphs, produced by the modified GCC. Code was added to traverse the aggregate graph to find the static reaching definitions, and the 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 the program execution. Each entry in the list also constitutes a node in the dynamic dependence graph. Each node is annotated with its dynamic *use* and *def* sets. The two sets are saved in terms of memory cells, which are (start-address, length) tuples, used and defined, respectively, by each node. The start addresses of the memory cells are obtained by evaluating the addresses of the *l*-valued expressions that constitute the *use* and *def* sets of the node. The *l*-valued expressions, as mentioned earlier, are determined by the compiler during parsing and made available to the debugger in the form of annotations to the nodes in the program dependence graph. The lengths of the memory cells are easily determined from the types of the expressions. To enable execution backtracking, the debugger also reads the contents of the *def* memory cells of each node and saves them along with the node in the dynamic dependence graph.

The modified GDB captures the above information by setting numerous ‘transparent’ breakpoints in the program that the user does not see. It associates callback functions with each of these transparent breakpoints to perform all the work of recording execution history, determining the starting memory addresses of the *use* and *def* memory cells, and saving the contents of the *def* memory cells. GDB provides a facility that lets one associate a callback list of debug commands with a breakpoint. We extended this facility so that, in addition to the debug commands, arbitrary functions may be included in the callback list. During the program execution, whenever a transparent breakpoint is reached, execution is interrupted and the control

is transferred to the debugger. The debugger then invokes all the callback functions associated with that breakpoint that in turn update the execution history, record the memory cells used and defined by the node, and save the contents of the *def* memory cells. After all the callback functions associated with the transparent breakpoint have been performed, the program execution is resumed.

Whenever the program execution normally stops, e.g. when a user-defined breakpoint (as opposed to a transparent breakpoint described above) is reached, the dynamic data and control dependence edges of all nodes added to the execution history since the last normal stop are determined and added to the dynamic dependence graph. The updated dynamic dependence graph may then be traversed to find the dynamic reaching definitions and the dynamic data, control, and program slices. The reader is referred to References 13 and 31 for details of the algorithms used to compute the data and control dependencies.

One of the main benefits of making the *use* and *def* sets consist of memory cells, or $\langle \text{start-address, length} \rangle$ tuples, is that it enables us to perform precise analysis even in the presence of pointers and dynamic memory allocations. It is their presence in a program that makes interstatement dependencies hard to visualize by the manual examination of the program text. The main difficulty with static analysis in the presence of an indirect reference through a pointer is that the memory location referenced by such an expression may vary during the course of the program execution, unlike that of a scalar variable that is fixed. The difficulty is compounded if the language used is not strongly-typed and permits integer arithmetic over pointer variables, as in the programming language C. In this case, if we analyze the program statically, we are forced to make the most conservative assumption—an indirect assignment through a pointer 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 program. On the other hand, our approach of defining the dynamic *use* and *def* sets in terms of $\langle \text{start-address, length} \rangle$ tuples, or memory cells, enables us to perform precise dynamic analysis even in the presence of unconstrained pointers. All the usual problems associated with static analysis in the presence of pointers, e.g. detection of aliases, are automatically taken care of by this approach because there is no ambiguity in determining if two memory cells overlap.

For example, consider the simple program in Figure 6. It initializes all elements of an array *a* and then prompts the user for values of *i*, *j* and *k*. It increments the *i*th, *j*th and *k*th elements of the array and prints out the new values of these elements. *p*, *q*, and *r* are pointer variables that point to the *i*th, *j*th and the *k*th elements of the array *a*, respectively. Consider the test case when this program is executed with input values (*i* = 1, *j* = 3, *k* = 3). Figure 6 also shows the dynamic slice with respect to *a*[*i*] on line 29 for this test case. Of the three indirect assignments on lines 25–27 the last two are included in the dynamic slice because values of *j* and *k* are the same for this test case, making *q* and *r* aliases to the same array element *a*[3]. The corresponding static slice contains the entire program.

Figure 7 shows a variant of the above program where a loop is used to initialize the array instead of having a separate assignment for each array element. If we execute this program for the same test case (*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

```

/u17/ha/v2/demo/ptr.c
1  main()
2
3  {
4
5      int a[10], i, j, k, *p, *q, *r;
6
7      a[0] = 0;
8      a[1] = 1;
9      a[2] = 2;
10     a[3] = 3;
11     a[4] = 4;
12     a[5] = 5;
13     a[6] = 6;
14     a[7] = 7;
15     a[8] = 8;
16     a[9] = 9;
17
18     printf("Enter i, j, k, (0 <= i,j,k < 10): ");
19     scanf("%d %d %d", &i, &j, &k);
20
21     p = &a[i];
22     q = &a[j];
23     r = &a[k];
24
25     *p += 1;
26     *q += 1;
27     *r += 1;
28
29     printf("a[%d] = %d, a[%d] = %d, a[%d] = %d\n", i, a[i], j, a[j], k, a[k]);
30
31 }

```

static analysis approx. dynamic analysis **exact dynamic analysis**

program slice data slice control slice reaching defs new testcase clear

run stop continue print backup step stepback delete quit

> dynamic program slice on "a[j]" at line 29

Current Testcase #: 1

Figure 6. Dynamic slice with respect to $a[j]$ on line 29

program execution. If we obtained 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 7. This suggests that the variable k was clobbered during the execution of the loop. Further examination reveals that the fault indeed lies with the loop predicate—it iterates ten times when the array is declared to be only eight elements long. Alternatively, it implies that the fault lies in the array declaration—it is declared to be eight elements long instead of ten. Figure 8 shows the memory allocation made by the compiler for all variables along with their contents at the end of the program execution for the above test case.* Note that the memory location that corresponds to k indeed overlaps with that of $a[9]$! The precise dynamic analysis enabled by our approach is invaluable in revealing such subtle faults.

The above approach also enables precise interprocedural dynamic slices to be obtained. Consider, for example, the program in Figure 9. It is the same program as that in Figure 1 except that the segment of code that determines the class of a triangle has been moved into a procedure called `find_class`. The same Figure also shows the dynamic program slice with respect to `area` on line 53 during the second loop iteration in the main program for the test case: $N = 2$ and the sides of the two triangles being (3, 3, 3) and (6, 5, 4). Our approach of resolving *def* and *use* sets of statements in terms of memory cells implies that there is no need to determine

* Memory allocation will vary from compiler to compiler.

address	contents	symbolic name
1000	1068	r
1004	0	
1008	1044	q
1012	0	
1016	1036	p
1020	0	
1024	10	l
1028	0	
1032	0	a[0]
1036	2	a[1]
1040	2	a[2]
1044	4	a[3]
1048	4	a[4]
1052	5	a[5]
1056	6	a[6]
1060	7	a[7]
1064	8	
1068	10	k
1072	0	
1076	3	j
1080	0	
1084	1	i

Figure 7. Dynamic slice with respect to k on line 27

which global variables are referenced inside a procedure, or which parameters may be aliases to each other or to global variables, nor do we need to eliminate name conflicts among variables in different procedures.

The same approach also enables us to easily implement execution backtracking. While recording the *use* and *def* memory cells of nodes in the execution history during the forward execution, as detailed earlier, the previous contents of *def* cells are also saved before they are modified. To backtrack execution, the saved contents of the *def* cells are restored into the corresponding memory locations while traversing the execution history backwards. The addresses of the memory locations to be restored are obtained from the corresponding memory cells. The backwards traversal and restoration of saved values continues until a user-defined breakpoint is encountered or the beginning of the execution history is reached. In the former case, the execution history is truncated at the breakpoint location; in the latter case, it is reinitialized to being empty.

Note that segments of the execution history may correspond to execution inside procedures. *Def* sets of nodes in these segments may contain memory cells that belonged to the activation records of old procedure cells that are no longer accessible. Contents of such memory cells are not restored during backtracking because, as indicated earlier, SPYDER currently does not support backtracking into a procedure from outside it. For example, in the program in Figure 9, SPYDER will not allow backtracking from the statement on line 44 to that on line 17. One may, however,

```

1      main()
2
3      {
4
5          int i, j, k, a[8], l, *p, *q, *r;
6
7          printf("Enter i, j, k, (0 <= i,j,k < 10): ");
8          scanf("%d %d %d", &i, &j, &k);
9
10         p = a;
11         l = 0;
12         while (l < 10)
13         {
14             *p = l;
15             p++;
16             l++;
17         }
18
19         p = &a[i];
20         q = &a[j];
21         r = &a[k];
22
23         *p += 1;
24         *q += 1;
25         *r += 1;
26
27     → printf("a[%d] = %d, a[%d] = %d, a[%d] = %d\n", i, a[i], j, a[j], k, a[k]);
28
29     }
30
31

```

static analysis approx. dynamic analysis **exact dynamic analysis**

program slice data slice control slice reaching defs new testcase clear

run stop continue print backup step stepback delete quit

> dynamic program slice on "k" at line 27

Current Testcase #: 1

Figure 8. Storage layout of the program in Figure 7 at the end of the program execution for the test case ($i = 1, j = 3, k = 3$)

backtrack from line 44 to line 42 over the procedure call on line 43, and then resume the forward execution into the procedure call up to any statement there.

The last major change that was made to GDB was to provide it a window- and mouse-based user interface so that slices could be displayed by highlighting the corresponding source lines. We also added hooks into the system so the traditional debugging functions supported by SPYDER, such as setting breakpoints, could be performed by simply selecting appropriate text in the source window using the mouse and clicking on appropriate command buttons. As our intention was simply to show how slicing and backtracking can be usefully combined with standard debugging functions like breakpoints, single-stepping, examining values, etc., we only included these most common debugging functions in our windowed interface rather than provide hooks for every function that GDB supports. The interface was written using the Athena widget set and the Xt toolkit of the X Window System, Version 11, Release 4.

CONCLUDING REMARKS

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 from the fault itself—both textually (in terms of the source lines) and temporally (in terms of the execution time). Providing facilities that increase

```

      /u17/ha/v2/demo/inter-proc.c
  8  void find_class(triangle, class_ptr)
  9      triangle_type triangle;
 10      class_type *class_ptr;
 11  {
 12      int a_sqr, b_sqr, c_sqr;
 13
 14      a_sqr = triangle.a * triangle.a;
 15      b_sqr = triangle.b * triangle.b;
 16      c_sqr = triangle.c * triangle.c;
 17      if ((triangle.a == triangle.b) && (triangle.b == triangle.c))
 18          *class_ptr = equilateral;
 19      else if ((triangle.a == triangle.b) || (triangle.b == triangle.c))
 20          *class_ptr = isosceles;
 21      else if (a_sqr == b_sqr + c_sqr)
 22          *class_ptr = right;
 23      else
 24          *class_ptr = scalene;
 25  }
 26
 27  main()
 28  {
 29      class_type class;
 30      double area, sum, s, sqrt();
 31      int N, i;
 32
 33      printf("Enter number of triangles:\n");
 34      scanf("%d", &N);
 35      for (i = 0; i < N; i++) {
 36          printf("Enter three sides of triangle %d in ascending order:\n", i+1);
 37          scanf("%d %d %d", &sides[i].a, &sides[i].b, &sides[i].c);
 38      }
 39
 40      sum = 0;
 41      i = 0;
 42      while (i < N) {
 43          find_class(sides[i], &class);
 44          if (class == right)
 45              area = sides[i].b * sides[i].c / 2.0;
 46          else if (class == equilateral)
 47              area = sides[i].a * sides[i].a * sqrt(3.0) / 4.0;
 48          else {
 49              s = (sides[i].a + sides[i].b + sides[i].c) / 2.0;
 50              area = sqrt(s * (s - sides[i].a) * (s - sides[i].b) *
 51                      (s - sides[i].c));
 52          }
 53          sum += area;
 54          i += 1;
 55      }
 56      printf("Sum of areas of the %d triangles is %.2f.\n", N, sum);
 57
  
```

static analysis
 approx. dynamic analysis
 exact dynamic analysis


```

> dynamic program slice on "area" at line 56
> clear
> stop at line 53
> backup
stopped at line 53.
> dynamic program slice on "area" at line 53
>
  
```

Current Testcase #: 1

Figure 9. Interprocedural dynamic program slice with respect to area on line 53 during the second loop iteration

the ability of the programmer to identify the location or the nature of the fault involved leads to a more efficient debugging. In this paper, we have presented a debugging model and a prototype tool that attempts to provide precisely these facilities. Our experience with both the model 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 employ them effectively. We conclude this paper with a discussion of some of the limitations of these techniques and outlining some of the lessons learned from this experience.

Limitations of the model

We mentioned at the beginning of this paper that a fault manifests itself, directly or indirectly, in terms of a data or a control discrepancy. While this is true, it requires that the programmer translate the externally visible symptoms 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 determined. But if the external symptom is more complex, e.g. a missing value in a list, the programmer may first have to use the traditional debugging facilities to find the corresponding internal symptom, and only then 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. SPYDER, nevertheless, attempts to partially overcome this problem by providing the local analysis facilities described earlier.

Execution backtracking also has its limitations. Backtracking over a statement requires that all side-effects of executing the statement be undone. But 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 be feasible to undo them. In such situations one may have to either accept ‘partial’ backtracking, or resort to re-executing 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 the use of these techniques. On the contrary, our experience has shown that despite these limitations, our techniques are extremely useful in quickly isolating program faults.

Limitations of the current implementation

As we mentioned before, we built our prototype tool on top of an existing computer and a debugger. While this choice enabled us to quickly build a working

system using which we could experiment with the proposed techniques and gain more insight into their usefulness, it also introduced some limitations on what could or could not be supported. For example, in GDB, and any other debugger that we know of, 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. It is, however, easy to provide a preprocessor that converts a program into the ‘canonical’ form acceptable to the tool. Another limitation of the current implementation, as described earlier, is that it does not permit backtracking into a procedure from outside it.

It may be noted that these limitations are of the current implementation, and not those of the techniques themselves. They arose largely 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 interpreter, or if we wrote our own compiler and the debugger.

Lessons learned from the implementation

One question we were faced with at the beginning of our implementation effort was whether to modify the compiler we used, GCC, to produce instrumented code that gathers run-time information necessary for dynamic slicing and backtracking, or to have the debugger, GDB, probe the program execution to collect the same information. Both approaches have their advantages and disadvantages. We decided to use the latter approach because we believed it would be easier to implement and experiment with.

One benefit of using our approach is that delaying the instrumentation of probes until the debugging time makes it possible to interactively control which parts of the program to instrument and when to instrument them. With this approach it is also possible to start without any instrumentation and use the static slicing techniques to narrow the search to the extent possible to a relatively smaller region of the program. Then we can successively increase the program instrumentation to use approximate and exact dynamic slicing techniques but on successively smaller and smaller program regions. This way, we need not 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 may be possible to instruct the debugger to ‘uninstrument’ that region. The alternative approach, on the other hand, would require having to pay the cost of instrumentation for the entire program execution, or require repeated recompilations of the program with less and less instrumentation.

One consequence of our decision to let the debugger perform all the instrumentation and collect necessary run-time information was that a great deal of time is spent by the system in context switching between the debuggee 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 the 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 should be substantially smaller. We may also use the approach discussed in the paragraph above to restrict the context switching to small regions of the program.

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 the necessary logging of the information required.^{44,45} It is possible that this approach, combined with the use of lightweight ‘thread’ processes, might support a very fast debugger for single-thread programs. We intend to examine this and other approaches, and run experiments to quantify and compare their relative overheads.

ACKNOWLEDGEMENTS

We would like to thank Bob Horgan and Guda Venkatesh for our discussions with them, Ed Krauser for his help during the implementation, and Ryan Stansifer, Hsin Pan, Nok Viravan, and the anonymous referees for their comments on earlier drafts of this paper.

This research was supported, in part, by a grant from the Software Engineering Research Center at Purdue University, a U.S. National Science Foundation Industry/University Cooperative Research Center (NSF Grant ECD-8913133), and by U.S. National Science Foundation Grant CCR-8910306.

REFERENCES

1. Hiralal Agrawal and Eugene H. Spafford, ‘A bibliography on debugging and backtracking’, *ACM Software Engineering Notes*, **14**, (2) 49–56 (1989).
2. Charles E. McDowell and David P. Helmbold, ‘Debugging concurrent programs’, *ACM Computing Surveys*, **21**, (4) 593–623 (1989).
3. Evan Adams and Steven S. Muchnick, ‘Dbxtool: a window-based symbolic debugger for Sun workstations’, *Software—Practice and Experience*, **16**, (7), 653–669 (1986).
4. Thomas A. Cargill, ‘Pi: a case study in object-oriented programming’, *OOPSLA’86 Conference Proceedings*, Portland, Oregon, September 1986. ACM Press. *SIGPLAN Notices*, **21**, (11) 350–360 (1986).
5. H. Katsoff, ‘Sdb: a symbolic debugger’, *Unix Programmer’s Manual*, 1979.
6. J. Maranzano and S. Bourne, ‘A tutorial introduction to ADB,’ *Unix Programmers Manual*, 1979.
7. Kevin J. Dunlap, ‘Debugging with Dbx’, *Unix Programmers Manual, Supplementary Documents 1*, 4.3 Berkeley Software Distribution, Computer Science Division, University of California, Berkeley, California, April 1986.
8. Ronald A. Olsson, Richard H. Crawford, W. Wilson Ho and Christopher E. Wee, ‘Sequential debugging at a high level of abstraction’, *IEEE Software*, May 1991, pp. 27–36.
9. Jacob T. Schwartz, ‘An overview of bugs’, in Randall Rustin (ed.), *Debugging Techniques in Large Systems* Prentice-Hall, Englewood Cliffs, New Jersey, 1971, pp. 1-16.
10. J. D. Gould, ‘Some psychological evidence on how people debug computer programs’, *International Journal of Man–Machine Studies*, **7**, (1) 151–182 (1975).
11. F. J. Lukey, ‘Understanding and debugging programs’, *International Journal of Man–Machine Studies*, **12**, (2) 189–202 (1980).
12. Hiralal Agrawal and Joseph R. Horgan, ‘Dynamic program slicing’, *Proc. SIGPLAN’90 Conference on Programming Language Design and Implementation*, White Plains, New York, June 1990. ACM Press. *SIGPLAN Notices*, **25**, (6), 246–256 (1990).
13. Hiralal Agrawal, Richard A. DeMillo and Eugene H. Spafford, ‘Dynamic slicing in the presence of unconstrained pointers’, *Proc. Fourth Symposium on Testing, Analysis and Verification (TAV4)*, ACM Press, October 1991, pp. 60–73.
14. Hiralal Agrawal, Richard A. DeMillo and Eugene H. Spafford, ‘An execution backtracking approach to program debugging’, *IEEE Software*, May 1991, pp. 21–26.

15. Keijiro Araki, Zengo Furukawa and Jingde Cheng, 'A general framework for debugging', *IEEE Software*, May 1991, pp. 14–20.
16. Mark Weiser, 'Programmers use slices when debugging', *Communications of the ACM*, **25**, (7), 446–452 (1982).
17. Mark Weiser, 'Program slicing', *IEEE Trans. Software Engineering*, **SE-10**, (4), 352–357 (1984).
18. Karl J. Ottenstein and Linda M. Ottenstein, 'The program dependence graph in a software development environment', *Proc. of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, Pittsburgh, Pennsylvania, April 1984, ACM Press. *SIGPLAN Notices*, **19**, (5), 177–184 (1984).
19. Susan Horwitz, Thomas Reps and David Binkeley, 'Interprocedural slicing using dependence graphs', *ACM Trans. Programming Languages and Systems*, **12**, (1), 26–60 (1990).
20. Jean-Francois Bergeretti and Bernard A. Carré, 'Information-flow and data-flow analysis of while programs', *ACM Trans. Programming Languages and Systems*, **7**, (1), 37–61 (1985).
21. Thomas Reps and W. Yang, 'The semantics of program slicing', *Technical Report TR-777*, Computer Science Department, University of Wisconsin, Madison, Wisconsin, June 1988.
22. Robert Cartwright and Matthias Felleisen, 'The semantics of program dependence', *Proc. ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, Portland, Oregon, June 1989. ACM Press. *SIGPLAN Notices*, **24**, (7), 13–27 (1989).
23. R. P. Selke, 'A rewriting semantics for program dependence graphs', *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages*, ACM Press, January 1989, 12–24.
24. G. A. Venkatesh, 'The semantic approach to program slicing', *Proc. SIGPLAN'91 Conference on Programming Language Design and Implementation*, Toronto, Ontario, Canada, June 1991. ACM Press. *SIGPLAN Notices*, **26**, (6), 107–119 (1991).
25. Susan Horwitz, Jan Prins and Thomas Reps, 'Integrating noninterfering versions of programs', *ACM Trans. Programming Languages and Systems*, **11**, (3), 345–387 (1989).
26. Keith Brian Gallagher and James R. Lyle, 'Using program slicing in software maintenance', *IEEE Trans. Software Engineering*, **17**, (8), 751–761 (1990).
27. L. Ott and J. Thuss, 'The relationship between slices and module cohesion', *Proc. Eleventh International Conference on Software Engineering*, IEEE Computer Society Press, May 1989.
28. H. Longworth, L. Ott and M. Smith, 'The relationship between program complexity and slice complexity during debugging tasks', *Proc. COMPSAC*, IEEE Computer Society Press, 1986.
29. Bogdan Korel and Janusz Laski, 'Dynamic slicing of computer programs', *J. Systems and Software*, **13**(3), 187–195 (1990).
30. James R. Lyle and Mark Weiser, 'Automatic program bug location by program slicing', *Proc. Second International Conference on Computers and Applications*, Beijing, China, July 1987.
31. Hiralal Agrawal, 'Towards automatic debugging of computer programs', *Ph.D. thesis*, Department of Computer Sciences, Purdue University, West Lafayette, IN, 1991.
32. Hsin Pan and Eugene H. Spafford, 'Towards automatic localization of software faults', *Proc. 10th Pacific Northwest Software Quality Conference*, October 1992, pp. 192–209.
33. R. M. Balzer, 'Exdams: extendible debugging and monitoring system', *AFIPS Proceedings, Spring Joint Computer Conference*, Vol 34, AFIPS Press, Montvale, New Jersey, 1969, pp. 567–580.
34. Barton P. Miller, and Jong-Deok Choi, 'A mechanism for efficient debugging of parallel programs', *Proc. SIGPLAN'88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988. ACM Press. *SIGPLAN Notices*, **23**, (7), 135–144, (1988).
35. Paul R. Wilson and Thomas G. Moher, 'Demonic memory for process histories', *Proc. SIGPLAN'89 Conference on Programming Language Design and Implementation*, Portland, Oregon, June 1989. ACM Press. *SIGPLAN Notices*, **24**, (7), 330–343, (1989).
36. M. V. Zelkowitz, 'Reversible execution as a diagnostic tool', *Ph.D. Thesis*, Dept. of Computer Science, Cornell University, January 1971.
37. Warren Teitelman, *Interlisp Reference Manual, Fourth Edition*, Xerox Palo Alto Research Center, Palo Alto, California, 1978.
38. Tim Teitelbaum and Thomas Reps, 'The Cornell Program Synthesizer: a syntax-directed programming environment', *Communications of the ACM*, **24**, (9), 563–573 (1981).
39. Stuart I. Feldman and Channing B. Brown, 'Igor: a system for program debugging via reversible execution', *Proc. Workshop on Parallel and Distributed Debugging*, Madison, Wisconsin, May 1988. ACM Press. *SIGPLAN Notices*, **24**, (1), 112–123 (1989).

40. James E. Archer, Jr., Richard Conway and Fred B. Schneider, 'User recovery and reversal in interactive systems', *ACM Trans. Programming Languages and Systems* **6**, (1), 1–19 (1984).
41. Richard M. Stallman, *Using and Porting GNU CC*, Version 1.37, Free Software Foundation, Cambridge, Massachusetts, January 1990.
42. Richard M. Stallman, *GDB Manual, Third Edition, Version 3.4*, Free Software Foundation, Cambridge, Massachusetts, October 1989.
43. Jeanne Ferrante, Karl J. Ottenstein and Joe D. Warren, 'The program dependence graph and its uses in optimization', *ACM Trans. Programming Languages and Systems*, **9**, (3), 319–349 (1987).
44. Edward W. Krauser, 'Compiler-integrated software testing', *PhD thesis*, Department of Computer Sciences, Purdue University, West Lafayette, IN, 1991.
45. Richard A. DeMillo, Edward W. Krauser and Aditya P. Mathur, 'Compiler-integrated program mutation', *Proc. Fifteenth Annual Computer Software and Applications Conference (COMPSAC)*, IEEE Computer Society Press, September 1991.