ABSTRACT

A recently developed, experimental, integrated System for Testing And Debugging is presented. Its testing part supports three data flow coverage criteria. The debugging part guides the programmer in the localization of faults by generating and interactively verifying hypotheses about their location.

1. INTRODUCTION

STAD is a recently developed prototype program testing and debugging system based on the philosophy of integration of verification activities. Its testing part supports three data flow strategies: Chain testing, involving single variables in the program, and U- and L-context testing that involve tuples of variables [3,8,25]. A novel feature of the debugger is its capability to guide the user in the localization of faults by generating hypotheses about their location.

A structural testing coverage criterion requires that certain parts of the code, or combinations thereof, be exercised. Ultimately, it is expressed in terms of a set of paths which activate those parts when executed. We use the generic term Required Elements (RES) to refer to those parts (this is a generalization of that term as originally introduced in [5]). The synthesis of test data for the RES is, however, difficult. Therefore, under STAD black-box testing is the basis for the preliminary testing, and the structural coverage is measured as a side-effect of it, rather than being applied independently. When the coverage is not satisfactory, the programmer can synthesize new structural tests.

Otherwise, the actual coverage can serve as a measure of program reliability, according to the strategy involved. When an error is detected, a debugging session begins.

Unlike traditional break-and-examine debuggers, the interactive debugger under STAD makes use of the structure of the program and of the execution trajectory along which an error has occurred. The information extracted from these two is represented by the program dependence network [16]. The network is used by the error-locating algorithms to guide the construction, evaluation, and modification of hypotheses about possible causes of the error.

In this paper, the user perspective of STAD is presented. Therefore, the implementation issues are not discussed here and the underlying theoretical principles are treated only briefly. The discussion is illustrated on an example introduced in the next section. This is followed by a debugging session (Sec. 3), a discussion of the principles of data flow testing (Sec. 4), a structural testing scenario (Sec. 5) and an introduction to the debugging principles in STAD (Sec. 6). Finally, in the Conclusions, future research is outlined.

2. AN EXAMPLE

Consider the program in Fig. 1 and its narrative specification in Fig. 2. The black-box test used in the experiment is shown in Fig. 3. Any black-box technique can be used in our scenario.

Testing the program on the black-box test is successful because an error is immediately detected for tests t1 and t2 from Fig. 3. Therefore, a debugging session can be initiated at this point. Such a session for test t2 is described in the next section.
program grader;
var
n,m: integer; (# of students, # tests)
i,j,k: integer; (aux variables)
sc:array[1..80] of real; (students' scores)
stnum:array[0..20] of integer; (students id's)
w: real; (test weight)
grade: array [1..20] of real; (grades)
max: array [1..20] of real; (max scores)
min: array [1..20] of real; (min scores)
avr: array [1..20] of real; (average scores)
xr,rr2,rr3:integer; rx,rr4:real; (auxiliary)

begin
(* ENTER INPUT DATA *)
1 write ('Enter # of tests:');
2 readln(m);
3,4,5,6 i:=0; w:=1/m; k:=1; stnum[i]:=1;
7 while stnum[i] > 0 do (* ->8,19 *)
8,9 i:=i+1; writeln;
10 write ('Enter student # : I);
11 readln(stnum[i]);
12 if stnum[i]>0 then (* ->13,7 *)
13 j:=1;
14 while j<=m do (* ->15,7 *)
15 write('score of test',j);
16,17 readln(sc[k]); k:=k+1;
18 j:=j+1;
end; (* -> 14 *)
end;
19 n:=i-1;
(* COMPUTE MAX, MIN, AVR *)
20 i:=1;
21 while i <= m do (* -> 22,39 *)
begin
22,23,24 mx:=sc[i]; mn:=sc[i]; av:=sc[i];
25,26 j:=2; k:=i+m;
27 while j <= n do (* -> 28,35 *)
begin
28 if mx < sc[k] (* -> 29,30 *)
begin
29,30 then mx:=sc[k];
31 then mn:=sc[k];
32,33,34 av:=av+sc[k]; k:=k+m; j:=j+1;
end; (* -> 27 *)
end;

The smallest, executable parts of statements identified at the source level are numbered as instructions (assignment statements, read and write statements, and predicates in the if and while statements). Notation (* -> i,j *) at instruction k identifies successors of k.

Fig.1. A listing of the Grader program.

3.DEBUGGING

For the test t2 in Fig.3, the program produces the following two-element list of pairs (Student #, Grade): (115, 25) (212, 15), rather than (212, 50) (115, 40). The other parts of the output, i.e. the total number of students (=2) and the statistics

Test # max min avr
1 50 40 45
2 60 30 45

are correct. In what follows the debugging session involved is described in Steps 1 through 7.
The program reads in a list of n students, identified by their id's (stnum), their scores for m tests and computes (1) max, min and avr score in class for each test, (2) grade for each student. It also sorts the grades in the descending order.

Example.
Input: m=4, stdsc = (115,(80 40 30 50)), (212,(0 90 20 10)), (310,(15 25 70 90)).

(There are 4 tests and 3 students. Student #115 has scores 80,40, 30 and 50 for the tests numbered 1,2,3,4, respectively.)

Expected Output: Sorted list of (Student #, Grade)= (212,30) (115,50) (310,50)

n = 3 (number of students)

m=4, stdsc = (115,(80 40 30 50)), (212,(0 90 20 10)), (310,(15 25 70 90))

Fig.2. Specification of Grader and a sample solution.

Step 1. On invoking the debugger the program is again executed on t2. Its text is displayed, and the statements being executed are highlighted. The execution is suspended in the last line of the program as the initial breakpoint, and the debugger queries the user to

ENTER INCORRECT VARIABLES:
The array grade is obviously incorrect. The array stnum[1..2] is incorrect, too, but only when considered together with grade. Indeed, if considered individually, stnum is acceptable because it is a permutation of the students' id numbers. Therefore, the user enters stnum and grade as the findings of incorrectness.

Step 2. Using the findings of incorrectness, the debugger generates a set of suspect statements. The suspected part of the program is determined to be the sorting loop 51-68 because it contains the last assignment to the variable grade. The next breakpoint is placed in the predicate 51 of the while loop. The program execution is resumed from the beginning. When the execution reaches the new breakpoint, the contents of stnum and grade are displayed

stnum = (115,212) grade=(25,15) (at 51).
The programmer is queried about their correctness. The array stnum appears correct. Because the array grade is incorrect, the user informs the debugger about that fact.

Step 3. On the basis of these findings, the debugger identifies the loop 41-49 as a new suspected part of the program and the next breakpoint is placed in the predicate 41 of that loop. The program execution is resumed from the beginning. When the execution reaches the breakpoint, the first time the debugger informs the user that

? ARRAY grade IS UNINITIALIZED (at 41).

Step 4. Because this is not an error, the execution continues. After one iteration, the value of grade[1] has been modified, and the debugger displays it:

? grade[1] = 25 (at 48)

That value should have been 40; the user informs the debugger about an incorrect value of grade[1]. Because the last assignment to grade[1] was in statement 48, the debugger draws the conclusion that statement 48 is faulty or the input to it, i.e. the variable grd, is incorrect. The second hypothesis is selected. To verify that hypothesis, the debugger displays the message
Step 5. The user confirms the incorrectness of \( \text{grd}=25 \) at 48 (the correct value is 40). On this basis, the debugger determines that the suspected part of the program is the loop 44-47 (variable \( \text{grd} \) receives a value in that loop). A breakpoint is inserted in front of instruction 44. The execution is resumed from the beginning. When the new breakpoint is reached the debugger displays

\[ \text{? \ grd=0 \ at \ 44} \]

Step 6. The user confirms correctness of that value at that point whereupon the execution is resumed. After one iteration, the value of \( \text{grd} \) has been modified. The debugger displays the message \( \text{? \ grd=25} \); because the user asserts its correctness the following message appears:

\[ \text{? EXIT FROM THE LOOP (44-47) AFTER 1 ITERATION.} \]

Step 7. However, this loop should iterate twice. Therefore, the user informs the debugger of incorrect flow of execution after one iteration. The debugger reasons that either predicate 44 is faulty or the input to it, the variable \( \text{grd} \), is incorrect. The second hypothesis is selected by the debugger. To verify this hypothesis, the value of variable \( \text{j} \) is displayed (it is assumed that the value of \( \text{m} \) is correct because it is the program input variable):

\[ \text{? j = 3} \]

However, the value of \( \text{j} \) after the first iteration should have been 2. The user informs the debugger about the incorrectness of \( \text{j} \). There are two suspects in that case: statements 42 and 47. It is now easy for the user to determine that statement 47 is faulty and that its correct version is \( \text{j := j + 1} \).

Having corrected the fault in instruction 47, the program is again run on the black-box test in Fig.3 (regression testing). This time, the testing fails to detect an error and, therefore, there are no grounds on which to reject the hypothesis that the program is correct. As suggested in Sec.1, this fact can be assigned a numerical value, if the concomitant percentage of coverage of a structural strategy is computed. In the next section, three such strategies are discussed.

4. DATA FLOW TESTING.

The rationale behind data flow testing is the assumption that faults in the program lead to incorrect values of some variables in it. If those are used to compute the values of other variables then an error might show up at the output. Therefore, the RES in data flow testing are certain patterns or data exchanges in the program. Those might involve individual program variables or tuples of variables.

Chain testing, introduced as "data environment" in [3], is equivalent to the strategy proposed earlier in [13]; when combined with branch coverage it is equivalent to "all-uses" in [6,7]. It requires that, for every variable in the program, every assignment to it be used in all possible places. In data flow parlance, an instruction that assigns a value to a variable is called a definition of the variable; an instruction in which the variable is used (referenced) is a use of the variable [1]. For example, instructions 3, 8, 20, 38, 40 and 49 in Fig.1 are all definitions of the variable \( i \) while instruction 8 and 22 are two uses of \( i \). If there is a path from a definition \( d \) to a use \( u \) of variable \( v \), along which \( v \) is not redefined, then the definition \( d \) reaches the use \( u \). (A path is a potentially executable sequence of instructions in the program). For example, in the program in Fig.1 the definition of the variable \( i \) in instruction 3 reaches instructions 6, 7, 8 and 19. The same definition does not reach, however, instruction 11 because, on any path to the latter, instruction 8 is always executed providing a new definition of \( i \). If there is a path from a definition \( d \) to a use \( u \) of variable \( v \), along which \( v \) is not redefined, then the definition \( d \) reaches the use \( u \). (A path is a potentially executable sequence of instructions in the program). For example, in the program in Fig.1 the definition of the variable \( i \) in instruction 3 reaches instructions 6, 7, 8 and 19. The same definition does not reach, however, instruction 11 because, on any path to the latter, instruction 8 is always executed providing a new definition of \( i \). A definition-use chain [1], or simply chain, is a triple \( (v,d,u) \) where (1) \( v \) is a variable, (2) \( d \) is a definition of \( v \), and (3) \( u \) is a use of \( v \) such that \( d \) reaches \( u \). Chain testing requires that all chains in the program be activated.

A U(se)-context at instruction \( k \) (originally introduced as "data context" in [3]) is a set of definitions of variables used in \( k \) that reach \( k \) together on a single path (cf [25,26]). For example, instruction 51 in the program in Fig.1 involves a simultaneous use of the variables \( r1 \) and \( n \). There are two U-contexts at 51: (19 50) and (19 68), denoted as tuples of distinct definitions of \( r1 \) and \( n \).

A L(ive)-Context at instruction \( k \) involves not only variables that are used in \( k \) but also those that can be used on a path originating in \( k \) before being assigned a new value [25,26]. For example, a
computation that originates at instruction 51 in Fig.1 is determined not only by the current values of n and r1 but also by the values of the variables stnum and grade, because they are referenced on a path from 51 to the end of the program. The variables n, r1, stnum and grade are said to be live at 51 (cf. [1]). However, live variables at an instruction might, but not have to, be used together. If they are, a tuple of their definitions that reach the instruction is called an L-context. For example, the following are live contexts at 51: (6 19 48 50), (11 19 48 50), (19 64 67 68).

U- and L-context testing respectively require that all U- and L-contexts in the program be exercised. It has been shown elsewhere [24] that, in the absence of certain data flow anomalies (i.e. referencing of uninitialized variables), U-context testing is more demanding than chain testing ("subsumes" it [7,10]). That is, whenever U-context coverage is met, so is chain coverage but not the other way round. Whenever defined, L-context testing is more demanding than U-context testing and, consequently, more demanding than chain testing. However, for some instructions L-contexts might not exist even if chains and U-contexts do. This is because of the fact that on some computations that originate at an instruction, only a subset of live variables might be reached, rather than all of them.

5. STRUCTURAL COVERAGE

STAD supports all of the above three data flow strategies. The cumulative history of their coverage during the testing session is shown in Fig.4. The lists of those RES that are not covered at the end of the session are shown in Fig.5. If the coverage is found incomplete, the user can use those lists to synthesize additional tests that activate RES in the lists.

Each uncovered RE listed in Fig.5 is, in fact, the specification of a sequence of instructions in the program that has to be traversed in order to activate that RE. Such a sequence will be referred to as a constructor of the RE.

Although STAD displays the RES that have not been covered, it does not identify their constructors: This task is currently left to the programmer. It is reasonable to begin the analysis with the least demanding, and simplest to understand, criterion, i.e. with chain coverage. For example, a constructor of the chain (1,3,19) has to traverse instruction 3 (assignment to i) and not go through instruction 8 (another assignment to i) before instruction 19 is reached. That means that the loop body at 7 should not be entered. However, because of the assignment in 6, the loop is always entered. Therefore, that chain cannot be activated at all: it is said to be infeasible. Incidentally, all chains listed in Fig.5 are also infeasible. Therefore, the black-box test in Fig.3 has resulted in 97% coverage of chains but in 100% coverage of feasible chains.

All U-contexts at instructions 19, 48, 53, 65 and the first two contexts at 59 in Fig.5 are infeasible. Apparently, this is due to the fact that they contain simple chains that are infeasible themselves [24]. For example, both U-contexts at 48 involve the chain (43,48), which is infeasible.

Test CHAINS U-CONTEXTS L-CONTEXTS
(total 168) (total 131) (total 136)
t1 88.07 81.51 47.06
t2 89.77 84.93 55.15
t3 89.77 87.67 61.03
t4 92.05 91.10 64.71
t5, t6, t7 no change
t8 97.00 92.47 64.71

Fig.4. Cumulative history of data flow coverage for the black-box test in Fig.3.

CHAINS NOT COVERED:
(1,3,19), (stnum,6,53), (stnum,6,59),
(stnum,6,65), (grd,43,48)

U-CONTEXTS NOT COVERED
(Instruction: Contexts)
19: (3); 48: (40 43) (43 49)
53: (6 50)
57: (52 61 64) (58 61 64) 58: (61 64)
59: (6 54) (6 61) (61 67) 65: (6 50)

SOME L-CONTEXTS NOT COVERED AT INSTRUCTION:
50: (6 19 48) (live variables: stnum grade)
57: (19 52 53 55 61 64 67 68)
(live variables: n k stnum grade r0 r1 r2 r3)

Fig.5. A sample coverage report at the end of the session in Fig.4.
All U-contexts at instructions 57 and 58 and the U-context (61 67) at 59 are feasible, however. Consider, for example, the U-context (52 61 64) at instruction 57 in which the variables grade, r0 and r2 are referenced. A constructor of the context has to (1) traverse the instructions 52, 61 and 64 (in any order), activating assignments to r0, r2 and grade, and (2) reach 57 in such a way that no other assignment to these variables occurs along it. Such a shortest constructor iterates the main loop at 51 twice. During the first iteration of the loop, the assignment to grade in 64 is activated; the way the inner loop is traversed now becomes important. During the second iteration of the main loop, the assignments to r0 in 52 and r2 in 61 are activated but the way the inner loop is traversed now becomes important. Clearly, it has to iterate twice, without entering the then clause of 57 on the first iteration, to avoid the assignment to r0 in 58. This means that, on entering the main loop at 51, there has to be (1) n=4 and (2) the first three entries of array grade are sorted. The following test
t9: m=1, stdsc = (1,(100)), (2 (80)),
   (3,(60)),(4,(70)), (5,(90))

covers all remaining U-contexts at instruction 57 and 58 and the U-context (61 67) at 59. Therefore, the black-box test in Fig.3, enriched by t9, covers all feasible U-contexts in the program.

As it was remarked in in Sec.4., in most cases L-context coverage is more demanding than U-context coverage. For example, the new test, t9, that "completes" U-context coverage also improves L-context coverage but it does not lead to a 100% coverage of it: Some new tests are needed to achieve that goal.

The infeasibility problem is undecidable, i.e. there is no algorithm that solves it for an arbitrary program. However, STAD offers some support that alleviates it: The user can enter a list of chains known to be infeasible and the system prunes the lists of contexts that contain those chains, updating the coverage accordingly.

6. DEBUGGING PRINCIPLES.

The basis for the debugging process is the dependence network that is used by the fault-locating algorithm [12,16,27]. The network models the way instructions in the program influence each other on a trajectory, i.e. a recorded history of a particular execution. Three types of such influence have been identified. Data influence is similar to the notion of chain defined with respect to the trajectory: An instruction assigns a value to an item of data and the other instruction uses that value [2]. Control influence is defined between test instructions and the instructions which can be chosen to execute or not execute by these test instructions. For instance, test instruction 7 of the program of Fig.1 has influence on the execution of instruction 8, but has no influence on the execution of instruction 19. The potential influence is defined between test instructions and instructions whose inputs are modified by an instruction that is within the control influence of the test instruction but which itself does not necessarily have to appear in the trajectory. For instance, instruction 44 controls the execution of instruction 45 which modifies the variable grd that is used in 48; therefore, 44 potentially influences 48.

The dependence network is the union of these dependencies [16]. An example of a part of it for the Grader program is shown in Fig.6.

The network provides a convenient way to represent possible sources of incorrect data (or incorrect control flow) along the trajectory that might have been responsible for the recorded finding of incorrectness. Therefore, it is used by the error-locating algorithm to guide the programmer during the fault localization. For a given finding of incorrectness, a set of hypotheses is generated. A tentative hypothesis is selected for findings already known to be present assuming, perhaps falsely, that no additional findings of incorrectness will subsequently be discovered. The hypothesis is verified; if it is found valid, the localization of the fault is narrowed. Otherwise, a new hypothesis is formulated. If, during the process of verification, new findings of incorrectness have been detected, a new set of hypotheses is generated and the process is repeated. If no more hypotheses can be formulated, the localization process terminates and a faulty "chunk" of code is reported.

This fault localization process generally moves backwards along the trajectory. In that way, the programmer is guided to examine the most likely sources of trouble in the program, rather than wandering around it in an uncontrolled way. Fig.7 shows the localization process for Steps 4-7 in Sec. 3.
Data influence arcs are labeled by the variables involved. Control influence and potential influence arcs are labeled by c and p, respectively.

Fig. 6. The dependence subnetwork for the one iteration of the loop 41-49 on test t2 from Fig. 3.

Suppose we start the error-localization process with the incorrect value of grade\[l\] at statement 48 in Step 4 in Sec. 3. We trace back from statement 48 through the dependence subnetwork of Fig. 5 to enumerate the possible sources of the incorrect value of grade\[l\]. There are two possibilities: either statement 48 is incorrect, or its input variable did not have the expected value and the problem lies further back. To pursue the second possibility, the input (variable grd) to statement 48 is verified. It is incorrect; we formulate hypotheses about the possible sources of the incorrect value of the variable grd at 48, and (H2) predicate 44 has potential influence on grd at 48. Statement 45 is selected as a suspect. To verify H1, the value of grd is verified after one iteration of the loop 44-47. It is correct, H1 is not valid. H2 is then selected, predicate 44 is a suspect. Now either predicate 44 is incorrect or its input is incorrect. To pursue the second possibility, the value of j (first input to 44) is verified. The value of j is incorrect. We trace back from 44 to enumerate the possible sources of incorrect value of j. The only suspect which can be generated from the dependence subnetwork of Fig. 5 is statement 47 which has data influence on 44. In the next step, the input to this statement is verified. Statement 47 is faulty. Observe that when the searching process is "close" to a faulty statement, the programmer can very often find out what is wrong. In this case, there is no need for precise localization of a faulty statement.

Fig. 7. The error localization process at Steps 4-7 in Sec.3.

7. CONCLUSIONS

Our example illustrates the fact that structural coverage of a well-designed black-box test tends to be stronger than that of chain testing, equivalent to feasible U-context coverage, and weaker than L-context testing. Theoretical analysis shows that, in the absence of data flow anomalies, U-context coverage indeed subsumes chain coverage [24]. As mentioned in Sec. 5, a similar relationship cannot, in general, be established for L-context coverage: The latter has to be redefined to take into consideration possible subsets of live variables at instructions. There is little doubt, however, that multidata testing corresponds to a more accurate model of data exchanges in the program than chain testing does.

Also, its error-detecting potential is greater than that of chain testing [3]. It is not only due to the subsumption involved: Apparently, exercising more complex patterns of data exchanges increases the probability of exposing hidden "persistent" errors that escape the initial testing [14]. Our experiments with STAD indicate that their cost, measured in the number of tests required, is still acceptable although no theoretical estimate of it is known.

It is felt, however, that ultimately data testing should involve sequences of interacting tuples of variables and their definitions along the lines suggested in [4] (k-dr interactions) and [5] (definition-tree). Array and procedure handling are still unsolved problems. Treating an assignment to an array element as a definition of the entire array is inadequate. Also, the difficulty of identifying definitions and uses of a procedure call requires a novel approach to interprocedural data flow analysis.

The integration of testing and debugging, both being based on closely related program models, appears to have been the single most fruitful concept of the system. A major experimental finding is that dependence-based debugging significantly speeds up fault localization, particularly for inexperienced programmers. The method has been proved to guarantee the localization of certain types of statement faults [16]. Further research is needed, however, to extend it onto design faults. To this point, a model of faults and their effects (errors) in the program is needed that can be used as a theoretical basis for both testing and debugging.
REFERENCES.


