Critical Slicing for Software Fault Localization*

Richard A. DeMillo  
Belcore  
445 South Street  
Morristown NJ 07960  
rad@bellcore.com

Hsin Pan  
Department of Computer and Information Science  
National Chiao-Tung University  
HsinChu, Taiwan, ROC  
hpan@cis.nctu.edu.tw

Eugene H. Spafford  
Software Engineering Research Center  
1398 Computer Sciences Building  
Purdue University  
West Lafayette, IN 47907-1398  
spaf@cs.purdue.edu

Keywords: testing, debugging, failures, faults, mutation analysis, critical slicing, dynamic program slicing, static program slicing, fault localization.

Abstract

Developing effective debugging strategies to guarantee the reliability of software is important. By analyzing the debugging process used by experienced programmers, we have found that four distinct tasks are consistently performed: (1) determining statements involved in program failures, (2) selecting suspicious statements that might contain faults, (3) making hypotheses about suspicious faults (variables and locations), and (4) restoring program state to a specific statement for verification. This research focuses support for the second task, reducing the search domain for faults, which we refer to as fault localization.

We explored a new approach to enhancing the process of fault localization based on dynamic program slicing and mutation-based testing. In this new approach, we have developed the technique of Critical Slicing to enable debuggers to highlight suspicious statements and thus to confine the search domain to a small region. The Critical Slicing technique is partly based on "statement deletion" mutant operator of the mutation-based testing methodology. We have explored properties of Critical Slicing, such as the relationship among Critical Slicing, Dynamic Program Slicing, and Executable Static Program Slicing; the cost to construct critical slices; and the effectiveness of Critical Slicing. Results of experiments support our conjecture as to the effectiveness and feasibility of using Critical Slicing for fault localization.

This paper explains our technique and summarizes some of our findings. From these, we conclude that a debugger equipped with our proposed fault localization method can reduce human interaction time significantly and aid in the debugging of complex software.

1 Introduction

In the software life cycle, there is considerable time and effort expended in the testing and debugging phases: some older studies have labeled that cost as more than 50% of the entire software development effort.[8, 33] Thus, developing effective and efficient testing and debugging strategies to guarantee the reliability of software is important.

In IEEE standards[7] errors are defined as inappropriate actions committed by a programmer or designer. Faults or bugs are the manifestations and results of errors during the coding of a program. A program failure occurs when errors or faults cause an unexpected result to be obtained while the program is executing on a certain input.

Many fault localization techniques used in current debugging tools (e.g., setting breakpoints) were developed in the 1960s and have changed little[5]. Users have to discover by themselves useful information for debugging. Two major steps involved in the debugging process are locating and then correcting faults. Previous studies[28, 38] have found that locating faults is the most difficult and important task in debugging.

The major goal of our research was to find an efficient way to accomplish the task of locating faults. By analyzing the debugging process used by experienced programmers, four distinct tasks were found to be consistently performed: 1) determining which statements are involved in program failures, 2) selecting suspicious statements that might contain faults, 3) making hypotheses about suspicious faults (variables and locations), and 4) restoring program state to a specific statement for verification. If all four tasks are performed with direct assistance from a debugging tool, the debugging effort becomes much easier. Our research focused on the second task, reducing the search domain for faults, referred to as fault localization. We believe that an improved approach to this task can enhance the debugging process and reduce human interaction time.

Our approach in this regard has been to combine two well-known techniques, testing and program slicing, in a unique manner. We began with the observation that testing explores the input space of a program such that it causes that program to exhibit a failure. Debugging tries to locate and fix faults (bugs) after failures are detected during testing.
or use. Although testing and debugging are thus closely related, none of the existing debugging techniques attempt to interface with the results of the testing phase. Our method has established one such relationship.

Program slicing is a powerful and useful method to aid in the understanding of actual program behavior. Dynamic program slicing techniques\(^1\) can determine which statements are actually affecting program failures so that the search domain for faults will be reduced. Although it is not guaranteed that dynamic slices always contain the faults (e.g., missing statement or specification faults), to investigate statements actually affecting program failures is a reasonable strategy in debugging. By analyzing semantics and values of variables in suspicious statements in dynamic slices, we might discover valuable information for debugging. However, that is not the only possible source of useful information. There is information we can derive from the testing phase (e.g., test cases and fault analysis from fault-based testing methodology) that should be helpful in the debugging process.

Our approach to enhancing the process of fault localization is based on dynamic instrumentation and mutation-based testing\(^2\).[32] In this paper, we describe the method, Critical Slicing, derived from the simplest mutant operator — “statement deletion.” Results of an experiment indicate that Critical Slicing may provide a cost-effective method of significantly reducing the search domain for faults. For programs in our study, the average reduction from the whole program was around 64%. In addition, around 80% of the obtained critical slices still contain the faulty statements that cause the failure, and thus do not mislead users in further debugging.

We have developed a family of heuristics for reducing the search domain based on a classification of test cases and a dynamic program slicing technique without the assistance of further testing information.[30] Results of our preliminary study for this proposed family of heuristics shows that the reduced search domains average at least a 43% reduction from the whole program.

The form of dynamic program slicing we use in debugging is formally defined by Agrawal et al. in [4, 2, 6]. We present here an informal definition, as presented in [3]:

> There are two major components to constructing a dynamic program slice: the dynamic data slice and the dynamic control slice. A dynamic data slice with respect to a given expression, location, and test case is a set of all assignments whose computations have propagated into the current value of the given expression at the given location. This is done by taking the transitive closure of the dynamic reaching definitions of the variables used in the expression at the given location. The set of all assignments that belong to this closure forms the dynamic data slice. On the other hand, a dynamic control slice with respect to a given location and test case is a set of all predicates that enclose the given location after executing the test case. This is done by taking the transitive closure of the enclosing predicates starting with the given location. The set of all predicates that belong to this closure forms the dynamic control slice.

In summary, a dynamic program slice, \(\text{DPS}(P, v, l, t)\), consists of all statements in the given program \(P\) that actually affect the current value of a variable \(v\) at location \(l\) when \(P\) is executed against a given test case \(t\). From now on, we use DPS to represent this approach of Dynamic Program Slicing and refer to it as exact dynamic program slicing. Also, the static program slicing proposed by Weiser [39, 40] is represented as SPS. A static program slice, \(\text{SPS}(P, v, l)\), contains executable statements of \(P\) that might affect the value of a selected variable-location pair \((v, l)\).

While using program slicing for fault localization, we wish to work with a small number of relevant statements in selected program slices that contain the faulty statements. Static slices (SPS) contain statements that may have nothing to do with the failure. Dynamic slices in the DPS approach consist of statements actually affecting (modifying) the value of a variable occurrence for a given input. However, it could be expensive to calculate the DPS, in some cases. A simpler method is to lightly instrument the code and derive the Executed Static Program Slicing (ESP). We define this as the set of statements in a selected static slice \(\text{SPS}(P, v, l)\) executed when we exercise \(P\) against a selected test case \(t\).

An executed static program slice is represented as \(\text{ESP}(P, v, l, t)\). The cost to calculate executed static slices is usually much less than those for dynamic slices. We then use the ESP Slices as the search space for fault localization.

The rest of this paper is organized as follows. In Section 2, we define Critical Slicing and analyze properties of Critical Slicing such as the relationship among Critical Slicing, Dynamic Program Slicing, and Executed Static Program Slicing; the cost to construct critical slices; and the effectiveness of Critical Slicing. We conducted a small experiment to explore the effectiveness and feasibility of using Critical Slicing for debugging. The result of the experiment is described in Section 3. Finally, we describe a new debugging paradigm with the support of our fault localization techniques in Section 4. Future directions of this research are also discussed.

The major contribution of this research is a new debugging paradigm for enhancing the process of fault localization by reducing the search domain. Our approach is based on information obtained from dynamic program slicing and mutation-based testing. Although it does not guarantee that faults will always be precisely located in the reduced search domain, the reduced search space containing the information leading to fault discovery is still helpful for users in debugging.
A simple mutant program of \( P, M \), is generated by mutating a statement of \( P \) according to one mutant operator. The only difference between \( P \) and \( M \) is the original statement at line \( S \) of \( P \) (i.e., statement \( S_P \)) and the mutated statement on \( S \) (referred to as a mutation, \( S_M \)) in \( M \). Test data are generated and executed against both \( P \) and \( M \). If the results (e.g., behavior or output) of \( P \) and \( M \) are different, mutant \( M \) is killed. The greater the number of mutants killed by a test set, the better the adequacy implied for that test set. Users try to kill all simple mutants by finding different test data. An adequate test set is thus constructed. If \( P \) is correct, the test set is evidence to assure the correctness of \( P \). On the other hand, if \( P \) is not correct, the faults will be manifested by test data generated for killing some simple mutants.

Figure 1 illustrates four mutants that are generated by applying the “statement deletion” mutant operator on an ANSI C do-while iteration-statement block. Mutant operators for C have been reported in [15, 14, 41]. Note from the description that follows that a specialized tool is not required to derive statement mutants. Rather, a simple script to remove a line of code at a time, execute, and compare results will provide the same results, albeit with less efficiency than a dedicated program.

A description of the “statement deletion” mutant operator, as presented in [1], is summarized as follows.

The statement deletion mutant operator (SSDL) is designed to show that each statement in a given program has an effect on the output. SSDL encourages the tester to design a test set that causes all statements to be executed and generates outputs that are different from the program under test.

To maintain the syntactic validity of the mutant, SSDL ensures that the semicolons are retained when a statement is deleted. In accordance with the syntax of C, the semicolon appears only at the end of (i) expression-statement and (ii) do-while iteration-statement. Thus, while mutating an expression-statement, SSDL deletes the optional expression from the statement, retaining the semi-colon. Similarly, while mutating a do-while iteration-statement, the semicolon that terminates this statement is retained. In other cases, such as the selection-statement, the semicolon automatically gets retained as it is not a part of the syntactic entity being mutated.

While exploring the statement analysis mutant operators (e.g., ssdl), we are interested in the actual effect made by each statement, especially when programs are executed by failure-revealing test cases. Faulty statements are likely in the set of those statements that directly contribute to program failures. The dead ssdl mutants identify a set of statements actually making a difference in program results and being critical to the program failures when the ssdl mutants are killed by failure-revealing test cases. This set of

---

2 Critical Slicing

During the testing phase, software testers create test cases to satisfy criteria of selected testing methodologies (e.g., statement coverage). These test cases are then executed against a given program. If a program failure occurs, the test case (referred to as the failure-revealing test case) manifests the existence of faults and will be used for debugging later. However, we are interested in not only the relevant failure-revealing and non-failure-revealing test cases, but also the testing criteria satisfied by the test cases (especially the failure-revealing test cases). The way to satisfy the criteria and features of the criteria being created could help us understand the behavior of program failure. As discussed in [31, 32], we chose program mutation as a testing methodology for debugging purposes and dynamic program slicing as our instrument for detailed analysis.

A simple and effective approach, referred to as Critical Slicing, is derived from studying the “statement deletion” mutant operator of the mutation-based testing methodology. A brief description of mutation-based testing is introduced here.

The principal goal of program mutation\(^3\) is to help users construct a mutation adequate test set that will differentiate a tested program \( P \) from incorrect programs. The adequacy of a test set is measured by executing that test set against a collection of simple mutant programs. A mutant program is made by introducing one simple change to program \( P \). These simple changes are considered as simple fault-inducing transformations on \( P \). They are derived empirically from both studying common faults made by programmers and abstracting the syntactic structure of faults. A set of mutant operators is formed based on those changes.

\(^3\)Readers are referred to [13, 12, 11, 16] for details of program mutation.
statements is the basis for Critical Slicing.

2.1 Definitions

Assume a set of statements \( S = \{S_1, S_2, \ldots, S_i, \ldots, S_F\} \) is an execution path when a faulty program \( P \) is executed against a given failure-revealing test case \( t \) with a failure type \( F_i \), where \( S_i \) represents an executable statement. \( F \) is the set of different types of failures, and \( S_F \) is the statement where the failure \( F_i \) occurs. For example, \( S_F \) could be the last statement being executed, and the output variables are wrong; or \( S_F \) is the statement where an exception failure (e.g., dividing by zero) occurs. For the failure types with incorrect-valued output variables, the incorrect values are considered as features of the related failure type.

Definition 1 A statement \( S_i \) in \( S \) of \( P \) is critical to a selected variable \( v \) in the failure \( F_i \) at location \( S_F \) for test case \( t \) if and only if the execution of \( P \) without \( S_i \) (i.e., an ssdl mutant \( M \) of \( P \) by deleting \( S_i \) from \( P \)) against the test case \( t \) reaches \( S_F \) with a different value of \( v \) from the one in \( S_F \).

This means that not only is \( M \) killed because the execution against \( t \) has a different result from the original one in \( F_i \), but also the execution reaches the same failure point \( S_F \). For the failure types with wrong output variables, the incorrect values of the erroneous output variables are used to decide whether \( M \) has the same result (values) as the original one in \( F_i \). The requirement of reaching \( S_F \) guarantees that the effect of executing \( S_i \) propagates to the same failure point. Meanwhile, killing \( M \) indicates that the effect of executing \( S_i \) actually makes a difference in the result. \( S_i \) is therefore critical to the failure. A set of statements with the same feature of \( S_i \) forms a critical slice with the following formal definition.

Definition 2 A Critical Slice is based on a 5-tuples \((P, F_i, v, S_F, t)\) with definition \( CS(P, F_i, v, S_F, t) = \left\{ S_i \mid S_i \text{ is critical to the variable } v \text{ of the program } P \text{ in the failure } F_i \text{ at location } S_F \text{ for test case } t \right\} \).

2.2 Properties of Critical Slicing

Critical Slicing (CS) provides another view for examining statements directly related to program failures that is different from the program dependency analysis of dynamic slicing. Important properties of CS such as relationships among critical slices, exact dynamic slices, and executed static slices, cost to obtain it, and its effectiveness will be examined in this section.

To compare Critical Slicing with other dynamic slicing techniques (Exact Dynamic Program Slicing, DPS, and Executed Static Program Slicing, ESPS), we have to ensure parameters involved in both approaches are in the same domain. A dynamic slice, \( DPS(P, v, l, t) \), and an executed static slice, \( ESPS(P, v, l, t) \), have four parameters — program \( P \), variable \( v \), location \( l \), and test case \( t \), where \( P \) will not be varied. Meanwhile, a critical slice, \( CS(P, F_i, v, S_F, t) \) has five parameters — program \( P \) which is the same as the \( P \) in DPS and ESPS, failure type \( F_i \), variable \( v \), location \( S_F \), and test case \( t \), where the failure type may have more than one variable involved to decide whether the execution has the same result as the original one in \( F_i \). If there are multiple variables (e.g., in a set \( v_j \)) involved in \( F_i \), then we consider the union of slices with respect to all involved variables at location \( S_F \) for the test case \( t \), i.e., \( \bigcup_{v \in v_j} DPS(P, v, S_F, t) \) vs. \( \bigcup_{v \in v_j} CS(P, F_i, v, S_F, t) \).

2.2.1 Relationships among CS, DPS, and ESPS

As the following two examples demonstrate, there is no superset or subset relationship between Critical Slicing (CS) and Exact Dynamic Program Slicing (DPS).

Example 1: If an assignment statement has no effect on the defined variable (i.e., value of the defined variable on the left hand side of the statement is the same before and after the statement is executed), then the value of the defined variable will be propagated and have the same effect to the result no matter whether the statement is executed or not. The result is thus unchanged if the statement is removed. Therefore, the statement is not in a corresponding critical slice. At the same time, it is possible that the defined variable actually affects the result and is thus in a corresponding exact dynamic slice. In this case, the statement is not in the critical slice but in the corresponding exact dynamic program slice.

For instance, the defined variable \( x \) (on left hand side) of Statement 2 in Figure 2 is assigned zero, which is the same as the value of \( x \) before Statement 2 is executed, if the memory initialization is zero for all variables. However, Statement 2 has data dependency on Statement 8 that assigns the output variable \( x \) at Statement 9. Thus, Statement 2 is not in the critical slice (indicated by $ signs) but in the corresponding exact dynamic slice (indicated by # signs).

Example 2: For the given program \( P \), variable \( v \), location \( l \), and test case \( t \), if a block of statements enclosed by
an executed predicate statement contains assignment statements in the corresponding static slice (with respect to v and t) but not in the corresponding exact dynamic slice (i.e., DPS(P, v, l, t)), then the predicate statement potentially affects v at l for t. In other words, the predicate statement keeps the value of v at l for t “intact”.

Although the predicate statement is not in the corresponding exact dynamic program slice, the predicate statement still contributes to the result by keeping the value of the corresponding variable intact within its scope. Therefore, the predicate statement is critical to the result if it is not executed and is thus in the corresponding critical slice. In this case, the statement is in the critical slice but not in the corresponding exact dynamic program slice.

For instance, Statement 5 in Figure 2 is in the critical slice but not in the exact dynamic slice.

The relationship between Critical Slicing (CS) and Executed Static Program Slicing (ESPS) is described in the following theorem.

**Theorem 1** Statements in a critical slice are a subset of statements in the corresponding executed static program slice, i.e., CS ⊆ ESPS.

**Proof:** By definition, all statements in a critical slice make a difference in the results at the selected point. These statements in the critical slice either actually or potentially affect the results.

For the first case — actually affecting the results — statements are included in the corresponding exact dynamic slice and executed static slice according to the definition of DPS and ESPS, respectively. For the second case — potentially affecting the results — statements are included in the corresponding executed static slice according to the definition of ESPS. Therefore, CS is a subset of ESPS.

On the whole, relationships between Critical Slicing (CS), Exact Dynamic Program Slicing (DPS), and Executed Static Program Slicing (ESPS) are demonstrated in Figure 3.

---

4 This definition is different from the potential influence defined by Korel [24]. Statements covered by their definition include statements with the “potential effect” type as well as in the exact dynamic slices (actual effect).

We suggest starting from statements in a selected executed static slice rather than the whole program as candidates for constructing a corresponding critical slice.

### 2.2.2 Cost

To construct a critical slice, if we verify each statement by executing a new program in the same way as the original one, except for removing the selected statement, then the total number of executions will be the number of all executable statements in the original program for each given test case. In other words, for a given program P and a test case set T, the cost of constructing a critical slice will be O(|P| × |T|), i.e., program size (the number of executable statements) times the test set size (the number of test cases in the given set). The cost is obviously high. As suggested above, we can construct a critical slice by selecting statements in the corresponding executed program slices instead of all executable statements. This is one step for reducing the cost, and corresponds with calculation of simple statement coverage for particular test cases.

Furthermore, if the debugging process is integrated with a mutation-based testing tool, we can get critical slices for debugging purpose during the testing phase without additional cost because the concept of building critical slices is derived from killing ssdl mutants. To eliminate ssdl mutants is usually one of the criteria first satisfied. Thus, we can obtain critical slices automatically while killing ssdl mutants in the testing phase with minor instrumentation.

For instance, the mutation-based testing tool will ask users to identify the failure type \( F_i \), variable v, failure location \( S_F \), and test case t, after t manifests the failure. While executing t to kill ssdl, an enhanced testing tool would compare the execution results to decide whether the original statement of the ssdl mutation statement is in the corresponding critical slice. In this case, the cost of constructing critical slices is not a concern.

### 2.2.3 Effectiveness

To study the effectiveness of Critical Slicing (CS), we are interested in the reduction in size (i.e., number of executable statements) of critical slices, corresponding executed static slices, corresponding exact dynamic slices, and the original program. Also, features of the statements not in the selected critical slices (i.e., not critical to the results of the selected critical slices) but in the corresponding exact dynamic slices and executed static slices (i.e., actually or potentially affecting the results) should be examined when studying the limitation of CS.

From the relationships among CS, DPS, and ESPS presented in Section 2.2.1, two categories can be identified for the statements not in a critical slice but in the corresponding executed program slice. The first one is the assignment statement having no effect on the defined variable but actually or potentially affecting the results. Example 1 illustrates this case (i.e., Statement 2 in Figure 2 which is not in a critical slice but in the corresponding exact dynamic slice).

The other category is that the propagation of the effect of an executed statement (including path selections and
3 Experimental Evidence about Critical Slicing

We have implemented a prototype debugging tool, SPYDER [2, 3, 6, 32], to perform the slicing (e.g., Static Program Slicing (SPS), Exact Dynamic Program Slicing (DPS), and Executed Static Program Slicing (ESPS)) and backtracking functions. Readers are referred to [6, 32] for details of the implementation and functions of SPYDER.

Figure 4 provides a snapshot of the SPYDER interface during a debugging session with the heuristics window. The development environment of SPYDER was on a Sun SPARC-station 1 running SunOS-4.1.1. SPYDER was built into the GNU C compiler "gcc" [36] and the GNU source-level debugger "gdb" [35].

As mentioned before, the purpose of fault localization is to provide a reduced search domain for locating faults. Thus, the effectiveness of fault localization techniques may be evaluated by the accuracy and the size of the reduced search domain. The information provided by the first factor demonstrates whether the reduced domain still contains faulty statements without misleading users. At the same time, the second factor demonstrates the effort that could be saved in terms of the number of suspicious statements to be examined. We therefore develop two comparison methods to evaluate the effectiveness of the proposed Critical Slicing technique.

We first describe the experimental methods as well as our criteria for comparison. Then, a set of tested programs that are faulty and have been previously referred to in the software testing community is illustrated. Features of these programs vary in fault types and locations that match previous studies of fault categories and frequencies in major projects [26] that indicate 26% logic faults, 18% data handling faults, 9% computational faults, etc. Although the sample space of tested programs is not large, we attempted to balance the features of our samples based on previous studies.

Finally, experimental results are presented in tables and figures with a detailed discussion. The results based on the tested programs are positive and show that the proposed fault localization techniques are a promising approach to debug faulty programs with (at least) similar features to our test programs.

3.1 Evaluation Methods

To evaluate the approaches of fault localization, two major criteria associated with the reduced search domain and the given faulty programs should be explored: 1) whether the reduced domain contains faulty statements that caused the failure, referred to as inclusion analysis; and 2) the size of the reduced domain is compared with the size of the tested program as well as the basis of dynamic slices, referred to as effectiveness comparison, which implies the possible effort of users for locating faults.

Inclusion Analysis

This analysis assures that the search domains suggested by Critical Slicing still contain faulty statements and will not mislead users in further debugging. If the reduced search domain contains faulty statements, then the coverage analysis result is positive; otherwise it is negative. Results of analysis are presented in tables regarding tested programs and test cases. Moreover, the percentage of total tested programs as well as test cases with positive inclusion analysis results is calculated. The higher the positive coverage percentage critical slices has, the more the affirmative effect of fault localization critical slices can promise.

Effectiveness Comparison

This comparison reflects the degree of improvement from the whole program to the reduced search domain and also indicates the possible effort to be saved for locating faults after employing approaches in fault localization.

The size of a critical slice with respect to a test case is compared with the size of the original program, the size of the corresponding executed static program slice, and the size of the corresponding exact dynamic program slice. The first and second values indicate the degree of improvement, and the last one shows the difference between critical slices and exact dynamic program slices in terms of size. The following ratios are computed for every critical slice, i.e., every failure-revealing test case of tested programs. Because the way to build critical slices uses a statement as the basic unit, the ratios associated with critical slices are based on statements instead of vertices in a program dependency graph:

\[ R'_{cs} = \frac{\# \text{ of statements in a CS}}{\# \text{ of executable statements of a tested program}} \]  \hspace{1cm} (1)

\[ R'_e = \frac{\# \text{ of statements in a CS}}{\# \text{ of statements in the corresponding ESPS}} \]  \hspace{1cm} (2)

\[ R'_d = \frac{\# \text{ of statements in a CS}}{\# \text{ of statements in the corresponding DPS}} \]  \hspace{1cm} (3)
If the dates are in the same month, we can compute the number of days between them immediately.

```c
if (nonth2 == nonth1)
```

Are we in a leap year?

```c
else
```

```c
/* Add the days in the remaining months */
```

```c
for (i = month1 + 1; i < j (+ 1); ++i)
```

```c
return retval + (day2 + (day1 + month1 - 1) - daysin[mnth1]);
```

Figure 4: X Window screen dump from SPYDER during a software debugging session

While considering the degree of improvement presented in \( R_{cs} \), \( R_e \), and \( R_d \), we are actually interested in the reduction rate between our approaches and the selected domain, e.g., the percentage of reduction from the size of a tested program to the size of region suggested by a critical slice. Therefore, these ratios are redefined as follows to be easily interpreted:

\[
R_{cs} = \begin{cases} 
1 - R_{cs}' & \text{if } R_{cs}' > 0 \\
0 & \text{if } R_{cs} = 0 
\end{cases}
\]  

(4)

\[
R_e = \begin{cases} 
1 - R_e' & \text{if } R_e' \neq 0 \\
0 & \text{if } R_e = 0 
\end{cases}
\]  

(5)

\[
R_d = \begin{cases} 
1 - R_d' & \text{if } R_d' \neq 0 \\
0 & \text{if } R_d = 0 
\end{cases}
\]  

(6)

\( R_{cs} \), \( R_e \), and \( R_d \) indicate the effectiveness of corresponding critical slices, i.e., the reduction rate for the size of the search domains. The larger the value of a reduction rate, the more effective the corresponding approach is.

### 3.2 Tested Programs

Eleven test programs were selected and constructed from seven original programs. Most of these programs were collected from previous studies and are well-known with previously studied faults. They have different characteristics of program size, number of functions, the type of program application (e.g., matrix calculation, text processing), and fault types as well as locations.

Table 1 gives the size, complexity, and characteristics (fault types) of each tested program. The second and third columns show the size of a tested program by listing the number of executable statements and the number of vertices in the program dependency graph of the program. Columns 4 to 7 are obtained from a data flow coverage

---

\(^*\) The programs are described in Appendix A.
testing tool — ATAC (Automatic Test Analysis for C programs) [23], developed at Bellcore.

Column blocks (#Bl) represents the number of code fragments not containing control flow branching.

Column decisions (#De) shows the number of pairs of blocks for which the first block ends at a control flow branch and the second block is a target of one of these branches.

Column p-uses (predicate uses, #P-u) indicates the number of triples of blocks for which the first block contains an assignment to a variable, the second block ends at a control flow branch based on a predicate containing that variable, and the third block is a target of one of these branches.

Column all-uses (#A-u) is the sum of p-uses and pairs of blocks for which the first block contains an assignment to a variable and the second block contains a use of that variable that is not contained in a predicate.

<table>
<thead>
<tr>
<th>Program</th>
<th>#ES</th>
<th>#V</th>
<th>#Bl</th>
<th>#De</th>
<th>#P-u</th>
<th>#A-u</th>
<th>fault types</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: aveg</td>
<td>35</td>
<td>57</td>
<td>36</td>
<td>18</td>
<td>40</td>
<td>79</td>
<td>wrong logical expression</td>
</tr>
<tr>
<td>P2: calend</td>
<td>29</td>
<td>51</td>
<td>32</td>
<td>12</td>
<td>16</td>
<td>31</td>
<td>wrong logical operator</td>
</tr>
<tr>
<td>P3: find1</td>
<td>33</td>
<td>53</td>
<td>32</td>
<td>18</td>
<td>80</td>
<td>124</td>
<td>wrong variable reference</td>
</tr>
<tr>
<td>P4: find2</td>
<td>33</td>
<td>53</td>
<td>32</td>
<td>18</td>
<td>80</td>
<td>124</td>
<td>wrong variable reference</td>
</tr>
<tr>
<td>P5: find3</td>
<td>32</td>
<td>52</td>
<td>32</td>
<td>18</td>
<td>80</td>
<td>122</td>
<td>missing a statement &amp; faults of find1 and find2</td>
</tr>
<tr>
<td>P6: gcd</td>
<td>57</td>
<td>76</td>
<td>57</td>
<td>36</td>
<td>124</td>
<td>230</td>
<td>wrong initialization (value)</td>
</tr>
<tr>
<td>P7: naur1</td>
<td>37</td>
<td>60</td>
<td>28</td>
<td>18</td>
<td>48</td>
<td>80</td>
<td>missing simple logical expression</td>
</tr>
<tr>
<td>P8: naur2</td>
<td>37</td>
<td>60</td>
<td>28</td>
<td>18</td>
<td>50</td>
<td>82</td>
<td>missing simple logical expression</td>
</tr>
<tr>
<td>P9: naur3</td>
<td>36</td>
<td>58</td>
<td>28</td>
<td>18</td>
<td>46</td>
<td>78</td>
<td>missing predicate statement</td>
</tr>
<tr>
<td>P10: transp</td>
<td>155</td>
<td>319</td>
<td>156</td>
<td>73</td>
<td>135</td>
<td>361</td>
<td>wrong initialization (value)</td>
</tr>
<tr>
<td>P11: trityp</td>
<td>37</td>
<td>55</td>
<td>47</td>
<td>39</td>
<td>99</td>
<td>113</td>
<td>wrong logical operators</td>
</tr>
</tbody>
</table>

The data flow coverage criteria (columns 4 to 7) help us understand the complexity of a tested program. Fault types in each tested program are described in Column 8.

Most of the tested programs have only one fault, except P5 and P11, so that we can easily examine the effectiveness of our proposed approaches for fault localization. Although P5 and P11 have two and three faulty statements, respectively, the multiple faulty statements in each program are related to each other. To evaluate the experimental results of P5 and P11, our analysis is based on the circumstance in which any one of the multiple faulty statements is highlighted by our approaches. For P5 and P11, we believe that as long as one faulty statement is discovered, the others can be easily identified.

The tested program “P9: naur3” has a special fault type — missing statements. The missing statement will not be highlighted by any of our proposed approaches. The purpose of having this sample in our experiment is to observe the effectiveness comparison of the suggested search domains, although the domains always have negative inclusion analysis. The ratio of effectiveness comparison in this case will be compared with others to perceive any significant difference.

3.3 Test Cases

In the testing phase, multiple test cases executed against P present different kinds of information. Our goal is to extract as much of that information as possible for debugging. The more related test cases we get, the better results we can have by investigating the information obtained from testing. Therefore, we prefer a thorough test — finishing the testing process to satisfy as many criteria of a selected testing method as possible. After a thorough test, if the existence of faults in program P is detected, then at least one test case will cause P to fail. Such a test case is called an failure-revealing (failure) test case, and T_f is the set of all such test cases. Likewise, the test cases on which P generates correct results are called non-failure-revealing (success) test cases, and T_s is the set of them. A set of failure-revealing test cases is an indispensable resource for debugging. On the other hand, not every non-failure-revealing test case is useful.[17]

ATAC is used to conduct the thorough test by obtaining two test case sets, T_s and T_f. A set of data-flow criteria for a selected program is provided after the tested program is analyzed by ATAC (e.g., blocks, decisions, p-uses, and all-uses). We then create test cases based on these criteria. Each test case will satisfy the criteria to a certain degree when executed against the tested program. A summary of the degree of satisfaction presented in Table 2 consists of both percentage and counts of all four criteria to show the adequacy of selected test cases. ATAC is employed to satisfy the coverage of criteria as much as possible and to guarantee the adequacy. In our experiment, we add test cases into T_s to improve the degree of satisfaction without causing program failures. Test cases in T_f are added to improve the degree of satisfaction under program failures. Information presented in Table 2 contains the highest percentage that was reached by the selected test cases. The number of test cases in T_s and T_f for each program is presented in Column 2 of Table 2.

According to the analysis in [17], only test cases associated with input domains causing failure are useful for de-
negative coverage. Thus, all test cases in $T_f$ and some test cases in $T_s$ are employed for debugging purposes after the thorough test. For instance, in the entry $T_f$ (Column 2) of P1, the total number of non-failure-revealing test cases obtained from ATAC is nine (9), which is enclosed in a pair of parentheses, but only eight (8) of them are related to program failures and useful for debugging. Other non-failure-revealing test cases are the "noise" in the debugging process. So are programs P6, P10, and P11.

3.4 Results of Critical Slicing

According to the definition and features of Critical Slicing (CS) in Section 2.1, every failure-revealing test case will construct one critical slice, and all non-failure-revealing test cases are not used in this experiment.

3.4.1 Inclusion Analysis

Table 3 presents results of inclusion analysis. All critical slices of "P9:naur3" have negative coverage results because of the missing statement fault. For program "P6: gcd" that has a wrong initialization fault, the memory initialization of the system environment will decide whether the faulty statement is included in a corresponding critical slice. If the memory initialization for all variables is zero, which is not the case on our platform, then all critical slices of P6 will have positive coverage. For test case $t_f$ of program "P3:find1", the execution of P3 without the faulty statement against $t_f$ generates a different result but does not reach the same point of original failure. Therefore, the corresponding critical slice does not contain the faulty statement and has a negative coverage. Strictly speaking, in this experiment, only the critical slice with respect to $t_f$ and "P3:find1" has negative coverage.

The high percentage of positive coverage results presented in the table shows the promise of employing Critical Slicing for fault localization. Although there is a negative coverage for P3, we still can find other critical slices of P3 with positive coverage to contain the faulty statement. In addition, the faulty statements that are not included in critical slices (e.g., P6) are always covered by the corresponding executed static program slices, except for P9 with a missing statement. This supports our claim in Section 2.2.3 that executed static slices are used for fault localization when the corresponding critical slices fail to include the faulty statements.

3.4.2 Effectiveness Comparison

Figure 5 presents the degree of reduction for the size of a critical slice (CS) being refined from the corresponding exact dynamic slice (DPS) and executed static slice (ESPS).

The $R_{es}$ of all critical slices range from 0.38 to 0.99 with median 0.64, mean 0.642, and standard deviation 0.170. In this experiment, we claim that the scope of critical slices is at least 38% reduction from the whole program and the average reduction rate is around 64%.

It is expected that the reduction rate from exact dynamic slices is smaller than the one from executed static slices because DPS is a subset of ESPS as indicated in Section 2.2. Thus, the value of $R_{es}$, which represents the reduction rate from the corresponding exact dynamic slice to a selected critical slice, could be negative. In this case, the size of the critical slice is larger than the size of the corresponding exact dynamic slice. $R_{es}$ of the program "P10: transp" in the figure demonstrates this special case (i.e., all four $R_{es}$ with negative value) which often happens for programs with heavy array and pointer refer-

<table>
<thead>
<tr>
<th>Prog.</th>
<th>TC</th>
<th>% blocks</th>
<th>% decisions</th>
<th>% p-uses</th>
<th>% all-uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: aveg</td>
<td>$T_f$</td>
<td>8 (9)</td>
<td>100 (36/36)</td>
<td>100 (18/18)</td>
<td>73 (29/40)</td>
</tr>
<tr>
<td>P1: aveg</td>
<td>$T_s$</td>
<td>6</td>
<td>100 (36/36)</td>
<td>94 (17/18)</td>
<td>70 (28/40)</td>
</tr>
<tr>
<td>P2: calend</td>
<td>$T_f$</td>
<td>5</td>
<td>100 (22/22)</td>
<td>90 (9/10)</td>
<td>94 (15/16)</td>
</tr>
<tr>
<td>P2: calend</td>
<td>$T_s$</td>
<td>3</td>
<td>90 (20/22)</td>
<td>60 (6/10)</td>
<td>69 (11/16)</td>
</tr>
<tr>
<td>P3: find1</td>
<td>$T_f$</td>
<td>6</td>
<td>100 (32/32)</td>
<td>100 (18/18)</td>
<td>79 (63/80)</td>
</tr>
<tr>
<td>P3: find1</td>
<td>$T_s$</td>
<td>3</td>
<td>97 (31/32)</td>
<td>94 (17/18)</td>
<td>76 (61/80)</td>
</tr>
<tr>
<td>P4: find2</td>
<td>$T_f$</td>
<td>5</td>
<td>100 (32/32)</td>
<td>100 (18/18)</td>
<td>79 (63/80)</td>
</tr>
<tr>
<td>P4: find2</td>
<td>$T_s$</td>
<td>3</td>
<td>97 (31/32)</td>
<td>94 (17/18)</td>
<td>75 (60/80)</td>
</tr>
<tr>
<td>P5: find3</td>
<td>$T_f$</td>
<td>6</td>
<td>100 (32/32)</td>
<td>100 (18/18)</td>
<td>80 (64/80)</td>
</tr>
<tr>
<td>P5: find3</td>
<td>$T_s$</td>
<td>4</td>
<td>97 (31/32)</td>
<td>94 (17/18)</td>
<td>75 (60/80)</td>
</tr>
<tr>
<td>P6: gcd</td>
<td>$T_f$</td>
<td>8 (10)</td>
<td>100 (57/57)</td>
<td>89 (32/36)</td>
<td>69 (85/124)</td>
</tr>
<tr>
<td>P6: gcd</td>
<td>$T_s$</td>
<td>9</td>
<td>95 (54/57)</td>
<td>81 (29/36)</td>
<td>64 (79/124)</td>
</tr>
<tr>
<td>P7: naur1</td>
<td>$T_f$</td>
<td>12</td>
<td>100 (28/28)</td>
<td>100 (18/18)</td>
<td>65 (31/48)</td>
</tr>
<tr>
<td>P7: naur1</td>
<td>$T_s$</td>
<td>2</td>
<td>96 (27/28)</td>
<td>89 (16/18)</td>
<td>56 (27/48)</td>
</tr>
<tr>
<td>P8: naur2</td>
<td>$T_f$</td>
<td>2</td>
<td>100 (28/28)</td>
<td>100 (18/18)</td>
<td>66 (31/50)</td>
</tr>
<tr>
<td>P8: naur2</td>
<td>$T_s$</td>
<td>3</td>
<td>100 (28/28)</td>
<td>100 (18/18)</td>
<td>62 (31/50)</td>
</tr>
<tr>
<td>P9: naur3</td>
<td>$T_f$</td>
<td>6</td>
<td>100 (28/28)</td>
<td>100 (18/18)</td>
<td>70 (32/48)</td>
</tr>
<tr>
<td>P9: naur3</td>
<td>$T_s$</td>
<td>7</td>
<td>100 (28/28)</td>
<td>100 (18/18)</td>
<td>78 (36/48)</td>
</tr>
<tr>
<td>P10: transp</td>
<td>$T_f$</td>
<td>5 (7)</td>
<td>94 (146/156)</td>
<td>89 (65/73)</td>
<td>81 (110/135)</td>
</tr>
<tr>
<td>P10: transp</td>
<td>$T_s$</td>
<td>4</td>
<td>96 (150/156)</td>
<td>90 (66/73)</td>
<td>79 (107/135)</td>
</tr>
<tr>
<td>P11: trityp</td>
<td>$T_f$</td>
<td>6 (14)</td>
<td>98 (46/47)</td>
<td>97 (38/39)</td>
<td>76 (75/99)</td>
</tr>
<tr>
<td>P11: trityp</td>
<td>$T_s$</td>
<td>3</td>
<td>66 (31/47)</td>
<td>51 (20/39)</td>
<td>37 (37/99)</td>
</tr>
</tbody>
</table>
Figure 5: Effectiveness comparison of critical slices.
Table 3: Inclusion Analysis for Critical Slicing. \(-T_f\) and \(#FV\) represent the number of failure-revealing (failure) test cases and variables involved in the failure, respectively. In each entry, \(\checkmark\) indicates a positive result and \(\times\) indicates a negative one.

<table>
<thead>
<tr>
<th>Program</th>
<th>(-T_f)</th>
<th>(#FV)</th>
<th>(t_1)</th>
<th>(t_2)</th>
<th>(t_3)</th>
<th>(t_4)</th>
<th>(t_5)</th>
<th>(t_6)</th>
<th>(t_7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: aveg</td>
<td>6</td>
<td>2</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: calend</td>
<td>3</td>
<td>1</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3: find1</td>
<td>3</td>
<td>1</td>
<td>(\checkmark)</td>
<td>(\times)</td>
<td>(\checkmark)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P4: find2</td>
<td>3</td>
<td>1</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P5: find3</td>
<td>4</td>
<td>1</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P6: gcd</td>
<td>9</td>
<td>2</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
</tr>
<tr>
<td>P7: naurl</td>
<td>2</td>
<td>1</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P8: naurl2</td>
<td>3</td>
<td>1</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P9: naurl3</td>
<td>7</td>
<td>1</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</td>
<td></td>
</tr>
<tr>
<td>P10: transp</td>
<td>4</td>
<td>2</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P11: trityp</td>
<td>3</td>
<td>1</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

% of the positive results based on all test cases in \(T_f = 30/47 = 64\%\)
% of the positive results based on all tested programs = \(9/11 = 82\%\)

Without considering P9 that has the special fault type — missing statements,
% of the positive results based on all test cases in \(T_f = 30/40 = 75\%\)
% of the positive results based on all tested programs = \(9/10 = 90\%\)

ences. In this circumstance, excluding a statement with array/pointer references will have serious side-effects and let the statement be easily included into the corresponding critical slice.

In our experimental results, \(R_e\) ranges from 0.0 to 0.76 with median 0.37, mean 0.366, and standard deviation 0.247. Meanwhile, \(R_d\) ranges from -0.25 to 0.76 with median 0.23, mean 0.289, and standard deviation 0.273. This implies that the scope of a critical slice has around 35% and 25% average reduction rate from the corresponding ESPS and DPS, respectively.

Our experiment indicates that Critical Slicing (CS) not only has the power of containing faulty statements as ESPS but also an effectively reduced scope for the search domain. Moreover, CS provides another view for examining statements directly related to program failures other than program dependency analysis for dynamic slicing. For programs without significant references to arrays or pointers, the size of a critical slice is smaller than the size of the corresponding exact dynamic slice, and most of the \(R_d\) in Figure 3 will be greater than 0.2.

4 Concluding Comments and Future work

Debugging is a complex and time-consuming activity. According to previous studies, locating faults is the most difficult task in the debugging process. In this paper, we have recognized that designating suspicious statements related to program failures is an important step done by experienced programmers to locate faults and therefore proposed a new approach to identify suspicious statements. Instead of looking at the whole program without effective clues, a reasonably small subset of the program with suspicious statements that directly contribute to program failures is suggested by our proposed fault localization techniques. We believe the task of locating faults will be improved in an efficient way so that users perform analysis at the right place — a reduced search domain containing faults. Further analysis for locating faults such as predicting fault types and locations, verifying the fault prediction, and fixing the faults is suggested as future work.

Despite the effort required to conduct program dependency analysis and mutation-based testing, the proposed approach based on dynamic program slicing and information from program mutation is an effective means for fault localization and fault identification. A new debugging paradigm was thus proposed. Then, a prototype debugging tool was implemented to demonstrate the feasibility of the approach whose effectiveness was confirmed by experiments. In this section, we summarize the experimental results of our approach, review the new debugging paradigm, and suggest future work.

From our research, Critical Slicing is an approach with a high percentage of positive inclusion analysis as well as significant reduction from the whole program when reducing the search domain. The average reduction rate from the whole program for software in our study is around 64%. Furthermore, the scope of a critical slice has around 35% and 25% average reduction rate from the corresponding exact dynamic slice (ESPS) and exact dynamic slice (DPS), respectively. With minor enhancement of a mutation-based testing tool, we can get critical slices for debugging purposes at no additional cost during the testing phase.

With the support of dynamic instrumentation (e.g., dynamic program slicing and backtracking) and fault localization techniques, an integrated testing and debugging tool can perform the following new debugging paradigm:

1. Users find program failures after a thorough test and analyze the failures before switching to the debugging
mode.

2. The debugging tool interactively helps users reduce the search domain for localizing faults by employing the proposed approaches based on dynamic instrumentation and information from testing. At this stage, users can easily switch between testing and debugging.

3. Further analysis based on the reduced search domain and information from testing is performed to locate faults.

4. After the faults are located and fixed, users can retest the program to assure program failures have been eliminated.

Developing fault prediction strategies to be used in Step 3 is a future direction of this research. In this paradigm, if users are not satisfied with the help provided by the debugging tool in Steps 2 and 3, they still can use their own debugging strategies by employing dynamic instruments (e.g., dynamic program slicing and backtracking) and information from testing. In this case, users are performing the traditional debugging cycle (hypothesize-set-examine) with powerful facilities and valuable information that are not supported by traditional debugging tools. We believe this paradigm will enhance the debugging process and save human interaction time.

The experiment in this paper demonstrates the positive results of the proposed approach for faulty programs with similar features of the tested programs. We need exhaustive experiments involving faulty programs with various program sizes, program applications, fault types, and fault locations. This exhaustive experiment will help to confirm the effectiveness of the approach. With enough samples, we can make a strong claim about the average reduction rate of the confined search domain as well as the threshold requirements for different kinds of faulty programs.

This work makes contributions to both testing and debugging. From the debugging point of view, the proposed approach for fault localization provides a reduced search domain for faults and improves human interaction in debugging. The debugging field benefits from this new direction in developing powerful tools. From the testing point of view, knowledge obtained from testing has been shown beneficial for debugging. This gives the testing field a new way to view the utility of testing methodologies.

Acknowledgment

The authors would like to thank Richard Lipton for discussing the concept of Critical Slicing and to the anonymous reviewers for their valuable comments. Bellcore's agreement to make ATAC available for research use at Purdue University is acknowledged.

References


133
Appendix A: Program Tested

Source code of all tested programs (written in the C programming language) may be obtained from H. Pan (see author information).

Program `aveg` first calculates the mean of a set of input integers. Then, percentages of the inputs above, below, and equal to the mean (i.e., the number of inputs above, below, and equal to the mean divided by the total number of inputs) are reported. Our version is directly translated from a Pascal version, and a fault was accidentally introduced during the transformation. The fault is an incorrect logical expression in an if-statement. If the temporary variable for the mean of the given integers has the value zero during calculation, then the result is incorrect.

Geller's calendar program `calend` [19], which was analyzed by Budd [11], is intended to calculate the number of days between two given days in the same year. An incorrect logical operator (== instead of !=) is placed in a compound logical expression of an if-statement. This fault causes errors in leap years.

The `find` program of Hoare [22] deals with an input integer array `a` with size `n > 1` and an input array index `f`, `1 < f < n`. After its execution, all elements to the left of `a[f]` are less than or equal to `a[f]`, and all elements to the right of `a[f]` are greater than or equal to `a[f]`. The faulty version of `find`, called `buggyfind`, has been extensively analyzed by SELECT [9], DeMillo-Lipton-Sayward [16], and Frankl-Weiss [18]. In our experiment, `find3` is the C version of `buggyfind`, which includes one missing statement fault and two wrong variable references (in logical expressions). The two wrong variable references were placed in `find1` and `find2`, respectively.

Bradley's `gcd` program [10], which was also analyzed by Budd [11], calculates the greatest common divisor for elements in an input integer array `a`. In our experiment, a missing initialization fault of `gcd` was changed to a wrong initialization with an erroneous constant.

Gerhart and Goodenough [20] analyzed an erroneous text formatting program (originally by Naur [29]). Minor modification of this program was made for our experiment. The specification of the program is as follows:

```
Given a text consisting of words separated by BLANKs or by NL (New Line) characters, convert it to a line-by-line form in accordance with the following rules: (1) line breaks must be made only when the given text has a BLANK or NL; (2) each line is filled as far as possible, as long as (3) no lines contain more than MAX-POS characters.
```

Program `naur1` has a missing path fault (e.g., a simple logical expression in a compound logical expression is missing). With this fault, a blank will appear before the first word on the first line except when the first word has the exact length of MAXPOS characters. The form of the first line is, thus, incorrect as judged by rule (2). Program `naur2` also has a missing path fault (e.g., a simple logical expression in a compound logical expression is missing). This fault causes the last word of an input text to be ignored unless the last word is followed by a BLANK or NL. Program `naur3` contains a missing predicate statement fault (e.g., an if-statement is missing). In this case, no provision is made to process successive line breaks (e.g., two BLANKs, three NLs).

Program `transp` [27], which was adopted for experiment by Frankl and Weiss [18], generates the transpose of a sparse matrix whose density does not exceed 66%. Two faults were identified in the original FORTRAN program. [21] We translated the correct version to C and reintroduced one of the faults. The other fault happens because of features of the FORTRAN language and cannot be reproduced in C. The fault present is a wrong initialization with an erroneous constant.

The last tested program, `trityp`, is a well-known experimental program.[34] It takes three input integers as the length of three sides of a triangle, and decides the type of the triangle (scalene, isosceles, equilateral, or illegal). The program contains three faulty statements with the same fault type, wrong logical operator (\( \geq \) instead of \( > \)).