Heuristics for Automatic Localization of Software Faults*

Technical Report SERC-TR-116-P

Hsin Pan *Eugene H. Spafford*

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

{pan, spaf}@cs.purdue.edu

July 29, 1992

Keywords: software testing, debugging, dynamic 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, four distinct tasks are found to be 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. If all four tasks could be performed with direct assistance from a debugging tool, the debugging effort would become much easier.

We have built a prototype debugging tool, *Spyder*, to assist users in conducting the first and last tasks. *Spyder* executes the first task by using *dynamic program slicing* and the fourth task by *backward execution*. This research focuses on the second task, reducing the search domain containing faults, referred to as *fault localization*.

Several heuristics are presented here based on dynamic program slices and information obtained from testing. A family tree of the heuristics is constructed to study effective application of the heuristics. The relationships among the heuristics and the potential order of using them are also explored. A preliminary study was conducted to examine the effectiveness of the heuristics proposed. Results of our study show the promise of fault localization based on these heuristics as well as suggest criteria for precise application of the heuristics (e.g., the standard of selecting thresholds). A new debugging paradigm equipped with these heuristics is expected to reduce human interaction time significantly and aid in the debugging of complex software.

*This research was supported, in part, by a grant from the Software Engineering Research Center at Purdue University, a National Science Foundation Industry/University Cooperative Research Center (NSF Grant ECD–8913133), and by National Science Foundation Grant CCR–8910306. A condensed version of this paper appears as [33].

1 Introduction

In the software life cycle, more than 50% of the total cost may be expended in the testing and debugging phases to ensure the quality of the software [30, 35]. Developing effective and efficient testing and debugging strategies is thus important.

In standards terminology [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 an unexpected result is obtained while executing the program on a certain input because of the existence of *errors* and *faults*.

Testing explores the input space of a program that causes the program to fail, while debugging tries to locate and fix faults or bugs after failures are detected during test or use. Although testing and debugging are closely related, none of the existing debugging tools attempt to interface with testing tools. Conventional debugging tools (e.g., ADB and DBX [15]) are command–driven and tend to be stand–alone. 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 correcting faults. Previous studies [30, 41] found that locating faults is the most difficult and important task in debugging. The focus of our work is to develop methods that automatically localize faults and thus enhance the debugging process as well as reduce human interaction time. The result of this study will support a new debugging paradigm proposed in [34].

By surveying how experienced programmers debug software, we found that four distinct tasks are consistently performed: (1) determining statements involved in program failures (e.g., execution paths) so that the exact behavior of program failures and the influence among statements are understood, (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. The first task can be achieved by executing a program step–by–step using some debugging tools. However, if a debugging tool can automatically highlight the execution path of a program for a given input, the first task will be accomplished more efficiently. The second and third tasks are currently performed by manually examining the code and program failures without the assistance of debugging tools. As to the fourth task, some debugging tools (e.g., DBX and GDB [40]) support facilities allowing users to set breakpoints, to reexecute the code, and to verify the values of variables. Obviously, the debugging effort would become much easier if all four tasks could be performed with direct assistance from a debugging tool.

We have built a prototype debugging tool, *Spyder* [1, 2, 4, 6], to assist users in conducting the first and last tasks. *Spyder* performs the first task by using *Dynamic Program Slicing* [1, 3], and can automatically find the dynamic slice of a program for any given variables, locations, and test cases in terms of data and control dependency analysis. A dynamic slice is denoted as $Dyn(P, v, l, t)$, where P is the target program, v is a given variable, l is the location of v, and t is a given test case [6]. $Dyn(P, v, l, t)$ contains statements of P actually affecting the value of v at location l when P is executed against test case t. Execution paths of the program with given inputs are special cases of Dynamic Program Slicing.

In conducting the fourth debugging task, *Spyder* can restore the program state to a desired location by *backtracking* the program execution to that location and need not reexecute the program from the beginning.[1] The work described in this paper focuses on the second task — to reduce the search domain containing faults

[1]Readers are referred to [1, 3] and [2, 4] for more details on dynamic program slicing and backtracking, respectively.

— referred to as *fault localization*. Several heuristics based on both dynamic program slices provided by *Spyder* and information obtained from testing are proposed.

The rest of this paper is organized as follows. The heuristics proposed by us are illustrated in the second section. In Section 3, a family tree of the proposed heuristics is constructed to study effective application. Examples and results of preliminary studies of these heuristics are presented in Section 4. A brief survey of related work is given in Section 5. Finally, the contributions and extensions to this research are suggested.

2 Our Approach

From the history of the development of fault localization, we find that techniques in some prototype systems work only for programs with restricted structure and solve only limited problems. An efficient debugging paradigm that deals with a broader scope of faults is needed.

2.1 Background

The tasks performed in the process of debugging are to collect valuable information for locating faults. The information may come from program specification, program behavior during execution, results of program execution, test cases after testing, etc. Because deterministic decisions (the general approach that systematically analyzes complicated information to benefit debugging) do not yet exist, we adopt a heuristic approach to gather useful information for different cases. We believe that heuristics can cover varied situations and help us localize faults.

Dynamic Program Slicing can determine statements 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 of dynamic slices, we might discover valuable information for debugging. Therefore, we choose dynamic slices as the search domain to locate faults.

We have developed a family of heuristics to further reduce the search domain based on the dynamic slices provided by *Spyder* and on information obtained from testing. Each heuristic will suggest a set of suspicious statements whose size is usually smaller than the size of the dynamic slices. Although each of our proposed heuristics is only suitable for some specific kinds of faults, the overall debugging power from uniting these heuristics is expected to surpass that of currently used debugging tools.

In the testing phase, multiple test cases running against program P present different kinds of information. Our goal is to extract that information as much as possible for debugging. The more test cases we get, the better results we might have by investigating the information obtained from testing. Therefore, we prefer a *thorough test* — to finish the whole 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 can cause P to fail. These test cases are called *error–revealing test cases*, T_d. Likewise, the test cases on which P generates correct results are called *non–error–revealing test cases*, T_u. Analyzing the results of program failures will help identify suspicious variables (e.g., output variables) that contain unexpected values. Dynamic slices with respect to these suspicious variables and corresponding test cases are then constructed for the heuristics.

A dynamic slice $Dyn(P, v, l, t)$ contains four parameters: the target program P, a given variable v, the location of v (l), and a given test case t. We define two metrics, *inclusion frequency* and *influence frequency*. *Inclusion frequency* of a statement is the number of distinct dynamic slices containing the statement; *influence frequency* of a statement in a $Dyn(P, v, l, t)$ is the number of times the statement was visited in $Dyn(P, v, l, t)$. By varying either or both of the test case (t) and the variable (v) parameters, we can get different kinds of dynamic slices that are used to determine corresponding metrics. Then, heuristics are applied based on the dynamic slices and metrics obtained. At this moment, the effectiveness of heuristics for automatic fault localization under different program (P) or location (l) parameters is not clear to us, but is the subject of ongoing research. [34]

Notations and terminology used in our heuristics are listed in Appendix A.

2.2 Heuristics

Several heuristics for fault localization based on dynamic slices are proposed here. Heuristics constructed using different test case parameters are similar to those constructed using different variable parameters. However, to vary the variable parameter we must be able to verify the value of a given variable with regard to the given location and test case. Because the error–revealing and non–error–revealing test case sets are obtained directly from a thorough test, it is preferred to first employ heuristics based on different test case parameters. In order to explain the heuristics clearly, we first illustrate heuristics using the notations of different test case parameters (Heuristics 1 to 16). These heuristics can be applied to cases of different variable parameters, and another sixteen similar heuristics will be obtained (Heuristic 17). Then, three more heuristics (Heuristics 18 to 20) are proposed based on varying both test case and variable parameters. For clarity and simplicity, the proposed heuristics are primarily developed by assuming only one fault in a failed program. However, many heuristics are still suitable for the case of multiple faults.

Heuristic 1 Indicate statements in $\{ \bigcup_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i}) \} \cup \{ \bigcup_{i=1}^{|T_d|} Dyn(P, v, l, T_{d_i}) \}$. □

Statements in dynamic program slices with respect to all test cases (both non–error–revealing and error–revealing test cases) are highlighted. This heuristic covers all statements in all available dynamic slices.

Heuristic 2 Indicate statements in $\bigcup_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i})$. □

This heuristic makes users focus on statements in dynamic program slices with respect to the non–error–revealing test cases (T_u). Faulty statements could be in these statements, if the fault is not triggered or not propagated to the result. These statements will be used by other heuristics.

Heuristic 3 Indicate statements with low inclusion frequency in $\bigcup_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i})$. □

A statement frequently involved with correct results has less chance to be faulty. When there exist many non–error–revealing test cases (T_u) and very few error–revealing test cases (T_d), this heuristic can be employed to find faulty statements in $\bigcup_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i})$. The faulty statements, if any exist[2], would not lead P to wrong results when executing test cases in T_u.

Heuristic 4 Indicate statements in $\{ \bigcup_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i}) \} \cup \{ \bigcup_{i=1}^{|T_d|} Dyn(P, v, l, T_{d_i}) \}$ with low (including zero) inclusion frequency that is counted according to the non–error–revealing test cases (T_u). □

[2]Failures may also be caused by missing statements. This issue is briefly discussed in Section 4.3.

This is a modification of Heuristic 3. Statements not in $\bigcup_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i})$ but in $\bigcup_{i=1}^{|T_d|} Dyn(P, v, l, T_{d_i})$ will also be considered, and the inclusion frequency of these statements counted based on T_u is simply zero. The idea behind this heuristic is that faulty statements might never be executed when P is executed against test cases in T_u. Thus, this approach is more flexible than Heuristic 3 and would highlight more suspicious statements than Heuristic 3 does.

Heuristic 5 Indicate statements in $\bigcap_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i})$, if the intersection is not an empty set. Then, study the necessity of the statements for correct results. $\square$

This heuristic will indicate statements with the highest inclusion frequency in $\bigcup_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i})$ of Heuristic 2, if the intersection is not an empty set. Studying the necessity of the statements that are inevitable in getting correct results might help us understand the nature of faults. This heuristic highlights exactly these statements. On the other hand, we could ignore these statements because they always lead to correct results, and focus on other statements also suggested by Heuristic 1.

Heuristic 6 Indicate statements in $\bigcup_{i=1}^{|T_d|} Dyn(P, v, l, T_{d_i})$. $\square$

This heuristic makes users focus on all statements in dynamic program slices with respect to the error–revealing test cases (T_d).

Heuristic 7 Indicate statements with high inclusion frequency in $\bigcup_{i=1}^{|T_d|} Dyn(P, v, l, T_{d_i})$. $\square$

A statement often leading to incorrect results has more chance to be faulty, and errors are confined to the statements executed. When there exist many error–revealing test cases (T_d) and very few non–error–revealing test cases (T_u), this heuristic can be employed to find the faulty statements in $\bigcup_{i=1}^{|T_d|} Dyn(P, v, l, T_{d_i})$.

Heuristic 8 Indicate statements in $\bigcap_{i=1}^{|T_d|} Dyn(P, v, l, T_{d_i})$, if the intersection is not an empty set. $\square$

This heuristic will indicate statements with the highest inclusion frequency in $\bigcup_{i=1}^{|T_d|} Dyn(P, v, l, T_{d_i})$ of Heuristic 7, if the intersection is not an empty set. If P has a few faults that commonly cause program failures, then this heuristic could locate the faulty statements quickly, especially when P has only one faulty statement (single fault).

Heuristic 9 Indicate statements in the difference set of $\bigcup_{i=1}^{|T_d|} Dyn(P, v, l, T_{d_i})$ and $\bigcup_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i})$: $\{\, \bigcup_{i=1}^{|T_d|} Dyn(P, v, l, T_{d_i}) \,\} \;-\; \{\, \bigcup_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i}) \,\}$. $\square$

Statements involved in the execution of program failures but never tested by cases in T_u (i.e., statements only appearing in the dynamic slices with respect to error–revealing test cases T_d) are highly likely to contain faults. If there are many test cases in both T_d and T_u, this method is worth trying. Statements indicated by this heuristic are in the difference set between the results of Heuristic 6 and Heuristic 2 (H6 $-$ H2).

Heuristic 10 Indicate statements in the difference set of $\bigcap_{i=1}^{|T_d|} Dyn(P, v, l, T_{d_i})$ and $\bigcup_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i})$: $\{\, \bigcap_{i=1}^{|T_d|} Dyn(P, v, l, T_{d_i}) \,\} - \{\, \bigcup_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i}) \,\}$. $\square$

This heuristic is similar to, but more rigorous than, Heuristic 9 because statements only executed by *all* test cases in T_d are considered. This heuristic indicates statements appearing in every dynamic slices of error–revealing test cases (failures) but not in any dynamic slices of non–error–revealing test cases (correct results). Nevertheless, this difference set (H8 $-$ H2) might be empty.

Heuristic 11 Indicate statements in the difference set of $\bigcup_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i})$ and $\bigcup_{i=1}^{|T_d|} Dyn(P, v, l, T_{d_i})$: $\{\ \bigcup_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i})\ \}\ -\ \{\ \bigcup_{i=1}^{|T_d|} Dyn(P, v, l, T_{d_i})\ \}$. $\square$

When P is executed against the non–error–revealing test cases (T_u), some statements might always be included to get correct results. Moreover, these statements never lead to the incorrect result. Further dependency analysis on these statements may provide useful information for locating faults. This heuristic can indicate these statements by finding the difference set between results of Heuristic 2 and Heuristic 6 (H2 $-$ H6). However, we might ignore these statements because they often contribute to correct results, and focus on other statements also suggested by Heuristic 1.

Heuristic 12 Indicate statements in the difference set of $\bigcap_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i})$ and $\bigcup_{i=1}^{|T_d|} Dyn(P, v, l, T_{d_i})$: $\{\ \bigcap_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i})\ \}-\ \{\ \bigcup_{i=1}^{|T_d|} Dyn(P, v, l, T_{d_i})\ \}$. $\square$

This heuristic is similar to, but more rigorous than, Heuristic 11 because statements only executed by *all* test cases in T_u are considered. This heuristic indicates statements appearing in every dynamic slices of non–error–revealing test cases (correct results) but not in any dynamic slices of error–revealing test cases (failures). Nevertheless, the difference set (H5 $-$ H6) might be empty.

Heuristic 13 Indicate statements in $\{\ \bigcup_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i})\ \} \bigcup \{\ \bigcup_{i=1}^{|T_d|} Dyn(P, v, l, T_{d_i})\ \}$ with high inclusion frequency in $\bigcup_{i=1}^{|T_d|} Dyn(P, v, l, T_{d_i})$ and low inclusion frequency in $\bigcup_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i})$. $\square$

This heuristic is a combination of Heuristics 3 and 7. It is suitable when many statements are involved in $\bigcup_{i=1}^{|T_d|} Dyn(P, v, l, T_{d_i})$ as well as $\bigcup_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i})$, and many elements are in T_d and T_u after a thorough test. Statements leading to incorrect results and less involved in the correct program execution are highly likely to contain bugs. For such statements, the ratio of the corresponding inclusion frequency in $\bigcup_{i=1}^{|T_d|} Dyn(P, v, l, T_{d_i})$ to the corresponding inclusion frequency in $\bigcup_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i})$ is a useful indicator. The higher ratio a statement has, the higher chance the statement contains faults.

Heuristic 14 If a set of statements B_1 located by the above heuristics, especially by those indicating statements with low inclusion frequency in $\bigcup_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i})$ (e.g., Heuristics 3, 4, and 13), does not contain faults and belongs to a branch of a decision block such as `if` (exp) `then` $\{\ldots B_1 \ldots\}$ `else` B_2, or `while` (exp) $\{\ldots B_1 \ldots\}$, then the logical expression exp should be examined. $\square$

Because the logical expression exp is always executed to decide whether B_1 should be executed, inclusion frequency of the predicate statement (`if`–statement or `while`–statement) is always equal to or greater than that of $B1$. The indication of the predicate statement based on inclusion frequency is thus not as effective as B_1. This heuristic reminds users to examine logical expression exp in the predicate statement.

Heuristic 15 Indicate statements with high influence frequency in a $Dyn(P, v, l, T_d)$. $\square$

The influence frequency measures the effect of statements being executed more than once (e.g., in a loop) but not counted in the inclusion frequency. The logic behind this heuristic is the same as that behind Heuristic 7 — statements often contributing to incorrect results are more likely to be faulty.

Heuristic 16 Indicate statements with low influence frequency in a $Dyn(P, v, l, T_u)$. $\square$

This heuristic, in other words, excludes statements with high influence frequency in the $Dyn(P, v, l, T_u)$ because statements often leading to correct results are less likely to be faulty.

Heuristic 17 Based on the above heuristics, another sixteen similar heuristics can be obtained by only varying the variable parameter (instead of different test case parameters) with respect to the given P, l, and t. $\square$

In these cases, we must be able to verify the value of suspicious variables with respect to the given location (l) and test case (t) in order to construct V_d (having incorrect value) and V_u (having correct value). These sixteen new heuristics indicate potential faulty statements based on another feature of dynamic slices (the variable parameter).

Heuristic 18 Indicate statements in $\bigcap_{j=1}^{|V_d|} Stmt(P, V_{d_j}, l, T_X)$, where T_X represents a selected test case in T_d or T_u, $Stmt(P, v, l, T_X)$ is the set of statements suggested by one of Heuristics 1 to 16, and V_d is mentioned in Heuristic 17. $\square$

The idea behind this heuristic is similar to that behind Heuristic 8. If wrong variable values are caused by the same fault, statements indicated by this heuristic are suspicious.

Heuristic 19 Indicate statements in $\{ \bigcup_{j=1}^{|V_d|} Stmt(P, V_{d_j}, l, T_X) \} - \{ \bigcup_{j=1}^{|V_u|} Stmt(P, V_{u_j}, l, T_X) \}$. $\square$

This heuristic is derived from Heuristic 9 with information obtained from varying both variable and test case parameters.

Heuristic 20 Indicate statements in $\{ \bigcap_{j=1}^{|V_d|} Stmt(P, V_{d_j}, l, T_X) \} - \{ \bigcup_{j=1}^{|V_u|} Stmt(P, V_{u_j}, l, T_X) \}$. $\square$

This heuristic is derived from Heuristics 10 and 19.

More specific criteria of applying these heuristics have yet to be studied. For example, the threshold for statements with high inclusion frequency in $\bigcup_{i=1}^{|T_d|} Dyn(P, v, l, T_{d_i})$ could be suggested as the highest twenty percent statements in the union sorted by inclusion frequency. Users are allowed to set their own threshold for different purposes. Our preliminary study discovers a better way to decide the threshold. The suggestion is discussed in Section 4.3.

Heuristics and experiments according to relational (decision–to–decision) path analysis on execution paths were studied by Collofello and Cousins [12]. A few of their approaches are similar to ours, such as the concept behind Heuristic 13, the most useful one among theirs. As our approach, at this moment, allows users to vary two out of four parameters (variable and test case) of dynamic slices that contain statements actually affecting program failures, possible faulty statements suggested by our heuristics should be more

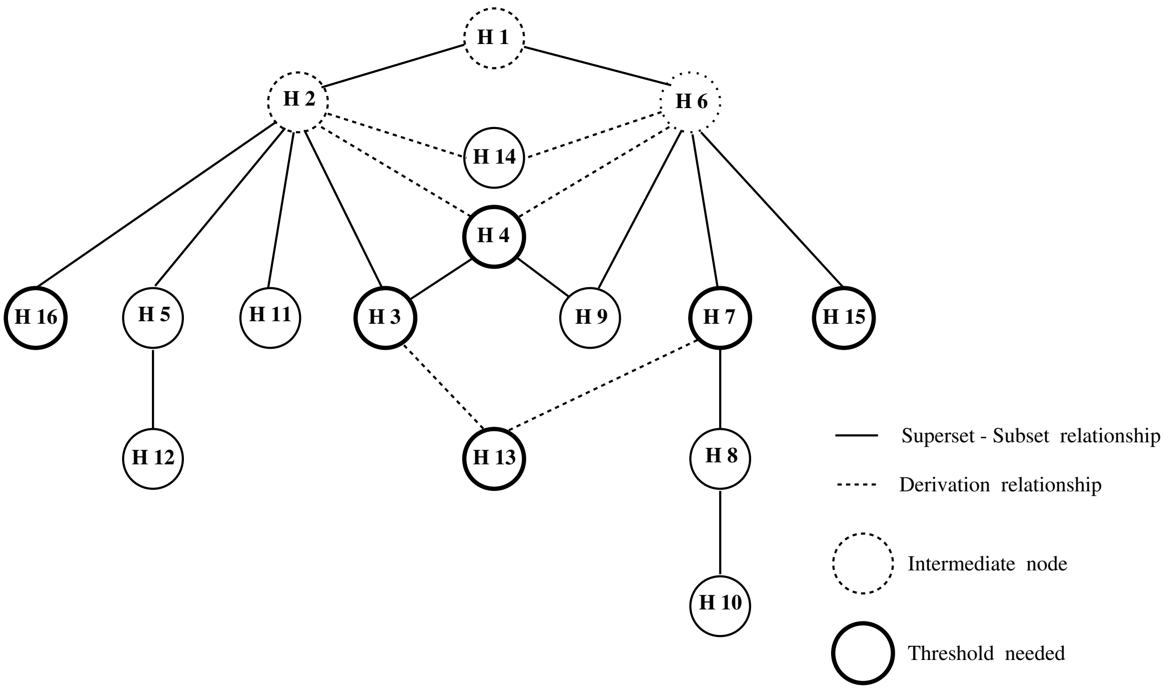

Figure 1: The family tree of the proposed Heuristics 1 to 16

precise than those suggested by theirs and other approaches. Also, more information is provided by our heuristics.

With the aid of our heuristics, a reduced search domain for faults (e.g., a smaller set of suspicious statements) is anticipated. Other functions of dynamic instrumentation provided by *Spyder* will then help us manage further examination. For instance, the reaching definition, which shows the latest definition of a variable and is a subset of the def–use chain in program dependency analysis, enables us to trace back to find the place where a suspicious variable went wrong. Then, the backtrack function effectively "executes" a program in *reverse* until a preset breakpoint is reached, just like forward program execution being suspended at breakpoints. In short, an efficient debugging session is conducted by locating faults from a reduced domain via our heuristics as well as by using effective functions of dynamic instrumentation.

3 Further Analysis

We have constructed a family of heuristics to study effective algorithms for applying the heuristics. In this family, relationships among the heuristics and the potential order of using them were explored.

We first examined heuristics with different test case parameters (Heuristics 1 to 16) by constructing the family tree in Figure 1. The same argument based on Figure 1 can then be applied to heuristics with different variable parameters (indicated in Heuristic 17). In Figure 1, each node represents one heuristic. A solid line connects an upper (parent) and a lower (child) nodes with a superset–subset relationship that statements indicated by the parent (upper) node contain those of the child (lower) node. A dotted line links an upper node and a lower node that is derived from the upper node but without a superset–subset relationship. Bold nodes are heuristics that require thresholds, and dot (intermediate) nodes are heuristics derived from others to construct a complete family tree. Statements highlighted by heuristics within the intermediate nodes only give us basic information for debugging. By contrast, their descendant heuristics will provide more helpful information. Among the intermediate nodes, Heuristic 1 (H1) is the root of the family tree, Heuristic 2 (H2) is the root of a subtree with respect to non–error–revealing test cases, and Heuristic 6 (H6) is the root of a subtree with respect to error–revealing test cases. H14 can be applied after traversing other heuristics in the tree except H15 and H16.

Heuristics of the subtree rooted by H2 are based on dynamic slices with respect to non–error–revealing test cases). These heuristics, especially H5, H12, and H11, are different from others because they highlight statements that often lead to correct results and suggest studying the necessity of the statements for correct results. However, we can ignore these statements and focus on other statements involved in the family tree (i.e., statements suggested by Heuristic 1). In this case, $\overline{\text{H5}}$, $\overline{\text{H12}}$, and $\overline{\text{H11}}$ represent the corresponding compliment heuristics.

As heuristics H1 to H14 are based on all dynamic slices with respect to error–revealing and/or non–error–revealing test cases, their functions are interpreted as *global analysis*. On the other hand, H15 and H16 conduct *local analysis* because they are based on one dynamic slice at a time. We prefer to perform global analysis first. Local analysis is then conducted to reduce the search domain further to locate faulty statements.

According to the family tree in Figure 1, heuristics without threshold requirements (especially for the nonintermediate nodes) are preferred before those with threshold requirements. This is because the former suggest a precise set of statements and the latter indicate a different set of statements for different thresholds. For a pair of nodes with a superset–subset relationship, the parent heuristic can be evaluated first because the child heuristic conducts analysis based on the result of the parent heuristic analysis (e.g., H5/H12 and H8/H10).

From the above discussion, global analysis can be summarized as:

Group 1: applying heuristics in nonintermediate nodes — H8/H10, H9, H5/H12, H11, $\overline{\text{H5}}$/$\overline{\text{H12}}$, and $\overline{\text{H11}}$.

Group 2: applying heuristics with threshold requirements — H4, H3, H7, and H13. Search domains provided by heuristics in this group contain less statements than those provided by heuristics in Group 1. The top–down order of employing these heuristics are preferred because the results of upper nodes will be used by lower nodes. For each heuristic, we should try the strictest threshold first. If faulty statements are not in the suggested region, the threshold will be increased gradually and the search domain will be expanded accordingly.

Group 3: applying H14 to recheck some suspicious statements that are ignored by the above heuristics.

While traversing the family tree, we interpret the top–down steps as refining suspicious statements and the bottom–up steps as extending the search domain. Users can make a guess first to get a set of suspicious statements for examination. Then, our tool will help them refine or extend the search domain by traversing the family tree of heuristics.

The result of global analysis is a set of suspicious statements (a reduced search domain) based on dynamic slices. To further verify faulty statements, heuristics performing local analysis (e.g., H15 and H16) are employed based on one dynamic slice at a time. Statements indicated by both global and local analyses are first examined, then the other highlighted statements.

Program	# of executable statements	blocks	decisions	p-uses	fault types
aveg	35	36	18	40	wrong logical expression
calend	29	22	10	16	wrong logical operator
find1	33	32	18	80	wrong variable reference
find2	33	32	18	80	wrong variable reference
find3	32	32	18	80	missing a statement & faults of find1 and find2
gcd	57	57	36	124	wrong initialization (value)
naur1	37	28	18	48	missing simple logical expression
naur2	37	28	18	50	missing simple logical expression
naur3	36	28	18	46	missing predicate statement
transp	155	156	73	135	wrong initialization (value)
trityp	37	47	39	99	wrong logical operator

Table 1: Tested programs

Techniques and steps discussed above can be applied to heuristics with different variable parameters (indicated in Heuristic 17) as well as to heuristics with different test case and variable paramenters (Heuristics 18 to 20).

After a reduced search domain for faults is presented by our proposed heuristics, further analysis is needed to identify whether faulty statements are in the highlighted suspicious region. Ongoing research will provide automated decision support to do verification. [42]

4 Results of A Preliminary Study

A simple trial was conducted to examine the effectiveness of the heuristics proposed. Because the major goal of fault localization is to reduce the search domain containing faults, the effectiveness was analyzed by comparing the known faulty statements and the suspicious domains suggested by heuristics for each tested program. In this section, we discuss the selection of tested programs, the results obtained from applying the heuristics.

4.1 Tested Programs

Eleven test programs were selected and constructed based on seven programs. Most of these programs were collected from previous studies and are well–known experimental programs with previously studied faults.[3]

Table 1 gives the size and complexity of each tested program. The first column lists the number of executable statements, showing the size of a tested program. Columns 3 to 5 are obtained from a data flow coverage testing tool — *Atac* (Automatic Test Analysis for C programs) [22], developed at Bellcore. The following definitions are quoted from the man pages of *Atac*.

Column *blocks* represents the number of code fragments not containing control flow branching. Column *decisions* shows the number of pairs of blocks for which the first block ends at a control

[3]The programs are described in Appendix B.

Prog.	t.c. types	% blocks	% decisions	% p-uses	% all-uses
aveg	T_u	100 (36)	100 (18)	73 (29/40)	81 (64/79)
	T_d	100 (36)	94 (17/18)	70 (28/40)	77 (61/79)
calend	T_u	100 (22)	90 (9/10)	94 (15/16)	97 (30/31)
	T_d	91 (20/22)	60 (6/10)	69 (11/16)	74 (23/31)
find1	T_u	100 (32)	100 (18)	79 (63/80)	84 (104/124)
	T_d	97 (31/32)	94 (17/18)	76 (61/80)	82 (102/124)
find2	T_u	100 (32)	100 (18)	79 (63/80)	84 (104/124)
	T_d	97 (31/32)	94 (17/18)	74 (59/80)	81 (100/124)
find3	T_u	100 (32)	100 (18)	80 (64/80)	85 (104/122)
	T_d	97 (31/32)	94 (17/18)	75 (60/80)	80 (98/122)
gcd	T_u	100 (57)	89 (32/36)	69 (85/124)	71 (163/230)
	T_d	95 (54/57)	81 (29/36)	64 (79/124)	67 (153/230)
naur1	T_u	100 (28)	100 (18)	65 (31/48)	66 (53/80)
	T_d	96 (27/28)	89 (16/18)	56 (27/48)	61 (49/80)
naur2	T_u	100 (28)	100 (18)	66 (33/50)	67 (55/82)
	T_d	100 (28)	100 (18)	62 (31/50)	65 (53/82)
naur3	T_u	100 (28)	100 (18)	70 (32/46)	69 (54/78)
	T_d	100 (28)	100 (18)	78 (36/46)	79 (62/78)
transp	T_u	94 (146/156)	89 (65/73)	81 (110/135)	85 (307/361)
	T_d	96 (150/156)	90 (66/73)	79 (107/135)	84 (304/361)
trityp	T_u	98 (46/47)	97 (38/39)	76 (75/99)	78 (88/113)
	T_d	66 (31/47)	51 (20/39)	37 (37/99)	39 (44/113)

Table 2: *Atac*'s measurement of test case adequacy

flow branch and the second block is a target of one of these branches. Column *p–uses* (predicate uses) 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* 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.

The data flow coverage criteria (columns 3 to 5) help us understand the complexity of a tested program. Fault types of each tested program are presented in Column 6. Most of the tested programs have only one fault so that we can easily examine the effectiveness of our proposed heuristics for fault localization.

4.2 Results

As mentioned in Section 2.1, a thorough test is preferred before applying our proposed heuristics. *Atac* was used to conduct the thorough test and to construct two test case sets, non–error–revealing test case set T_u and error–revealing test case set T_d. A set of data–flow criteria of a selected program is provided after the tested program is analyzed by *Atac* (e.g., blocks, decisions, p–uses, and all–uses). Each test case satisfies the criteria to a certain degree when executed against the tested program. Summary of the satisfaction degree presented in Table 2 consists of both percentage and counts of all four criteria to show the adequacy

Prog.		H1	H2	H6	H8	H10	H9
aveg	a	80% (28/35)*	80% (28/35)*	80% (28/35)*	66% (23/35)*	$\emptyset$	$\emptyset$
	b	(28)*	100% (28/28)*	100% (28/28)*	83% (23/28)*	$\emptyset$	$\emptyset$
calend	a	80% (23/29)*	80% (23/29)*	76% (22/29)*	25% (7/29)*	$\emptyset$	$\emptyset$
	b	(23)*	100% (23/23)*	96% (22/23)*	31% (7/23)*	$\emptyset$	$\emptyset$
find1	a	76% (25/33)*	76% (25/33)*	76% (25/33)*	73% (24/33)*	$\emptyset$	$\emptyset$
	b	(25)*	100% (25/25)*	100% (25/25)*	96% (24/25)*	$\emptyset$	$\emptyset$
find2	a	76% (25/33)*	76% (25/33)*	76% (25/33)*	67% (22/33)*	$\emptyset$	$\emptyset$
	b	(25)*	100% (25/25)*	100% (25/25)*	88% (22/25)*	$\emptyset$	$\emptyset$
find3	a	75% (24/32)*	75% (24/32)*	75% (24/32)*	66% (21/32)*	$\emptyset$	$\emptyset$
	b	(24)*	100% (24/24)*	100% (24/24)*	88% (21/24)*	$\emptyset$	$\emptyset$
gcd	a	69% (39/57)*	67% (38/57)	64% (36/57)*	20% (11/57)*	2% (1/57)*	2% (1/57)*
	b	(39)*	98% (38/39)	93% (36/39)*	29% (11/39)*	3% (1/39)*	3% (1/39)*
naur1	a	84% (31/37)*	84% (31/37)*	84% (31/37)*	68% (25/37)*	$\emptyset$	$\emptyset$
	b	(31)*	100% (31/31)*	100% (31/31)*	81% (25/31)*	$\emptyset$	$\emptyset$
naur2	a	84% (31/37)*	84% (31/37)*	84% (31/37)*	68% (25/37)*	$\emptyset$	$\emptyset$
	b	(31)*	100% (31/31)*	100% (31/31)*	81% (25/31)*	$\emptyset$	$\emptyset$
naur3	a	84% (30/36)	84% (30/36)	84% (30/36)	75% (27/36)	$\emptyset$	$\emptyset$
	b	(30)	100% (30/30)	100% (30/30)	90% (27/30)	$\emptyset$	$\emptyset$
transp	a	30% (45/155)*	29% (44/155)	20% (31/155)*	19% (29/155)*	1% (1/155)*	1% (1/155)*
	b	(45)*	98% (44/45)	69% (31/45)*	65% (29/45)*	3% (1/45)*	3% (1/45)*
trityp	a	57% (21/37)*	57% (21/37)*	41% (15/37)*	28% (10/37)*	$\emptyset$	$\emptyset$
	b	(21)*	100% (21/21)*	72% (15/21)*	48% (10/21)*	$\emptyset$	$\emptyset$

$*$: the highlighted statements contain faulty statements
a: # of highlighted statements / # of executable statements
b: # of highlighted statements / # of statements highlighted by H1

Table 3: Effectiveness analysis for heuristics without threshold requirements

of selected test cases. *Atac* was employed to satisfy the coverage of criteria as much as possible and to guarantee adequacy. In our experiment, test cases in T_u were added to improve the satisfaction degree without causing program failure. Test cases in T_d were added to improve the satisfaction degree under program failure. Information presented in Table 2 contains the highest percentage that were reached by the selected test cases.

Our heuristics were then applied based on T_u, T_d, and output variables of tested programs using *Spyder*. A set of suspicious statements was suggested by each heuristic for a selected program. To measure the effectiveness of a proposed heuristic, we first examined whether the known faulty statements were contained by the suggested statements. Then, we compared the reduced domain with the size of the original tested program (i.e., the number of executable statements) and with the domain suggested by the root of the heuristic family in Figure 1 (i.e., H1). These results are presented in Rows a and b of Tables 3 and 4. The percentage of each entry in the tables tells the degree of effectiveness for the reduced search domain. The star superscript in an entry indicates the suggested statements containing the known faulty statements. Thus, heuristics with low promising percentage (especially in Row b) are preferred, except for H5, H12, and H11.

Prog.		H5	H12	H11	$\overline{H5}$	$\overline{H12}$	$\overline{H11}$
aveg	a	60% (21/35)*	∅	∅	20% (7/35)	80% (28/35)*	80% (28/35)*
	b	75% (21/28)*	∅	∅	25% (7/28)	100% (28/28)*	100% (28/28)*
calend	a	14% (4/29)	∅	4% (1/29)	66% (19/29)*	80% (23/29)*	76% (22/29)*
	b	18% (4/23)	∅	5% (1/23)	83% (19/23)*	100% (23/23)*	96% (22/23)*
find1	a	46% (15/33)	∅	∅	31% (10/33)*	76% (25/33)*	76% (25/33)*
	b	60% (15/25)	∅	∅	40% (10/25)*	100% (25/25)*	100% (25/25)*
find2	a	49% (16/33)	∅	∅	28% (9/33)*	76% (25/33)*	76% (25/33)*
	b	64% (16/25)	∅	∅	36% (9/25)*	100% (25/25)*	100% (25/25)*
find3	a	60% (19/32)*	∅	∅	16% (5/32)*	75% (24/32)*	75% (24/32)*
	b	80% (19/24)*	∅	∅	21% (5/24)*	100% (24/24)*	100% (24/24)*
gcd	a	25% (14/57)	4% (2/57)	6% (3/57)	44% (25/57)*	65% (37/57)*	64% (36/57)*
	b	36% (14/39)	6% (2/39)	8% (3/39)	65% (25/39)*	95% (37/39)*	93% (36/39)*
naur1	a	9% (3/37)	∅	∅	76% (28/37)*	84% (31/37)*	84% (31/37)*
	b	10% (3/31)	∅	∅	91% (28/31)*	100% (31/31)*	100% (31/31)*
naur2	a	9% (3/37)	∅	∅	76% (28/37)*	84% (31/37)*	84% (31/37)*
	b	10% (3/31)	∅	∅	91% (28/31)*	100% (31/31)*	100% (31/31)*
naur3	a	9% (3/36)	∅	∅	75% (27/36)	84% (30/36)	84% (30/36)
	b	10% (3/30)	∅	∅	90% (27/30)	100% (30/30)	100% (30/30)
transp	a	27% (41/155)	10% (14/155)	10% (14/155)	3% (4/155)*	20% (31/155)*	20% (31/155)*
	b	92% (41/45)	32% (14/45)	32% (14/45)	9% (4/45)*	69% (31/45)*	69% (31/45)*
trityp	a	9% (3/37)	∅	17% (6/37)	49% (18/37)*	57% (21/37)*	41% (15/37)*
	b	15% (3/21)	∅	29% (6/21)	86% (18/21)*	100% (21/21)*	72% (15/21)*

∗: the highlighted statements contain faulty statements
a: # of highlighted statements / # of executable statements
b: # of highlighted statements / # of statements highlighted by H1

Table 3: (continued) Effectiveness analysis for heuristics without threshold requirements

As discussed in Section 3, heuristics without threshold requirements should be applied before heuristics with the requirements. Results of applying heuristics without threshold requirements are presented in Table 3. Heuristics 1, 2, and 6 are considered separately because they are intermediate nodes in Figure 1, and search domains highlighted by H2 and H6 are not significantly reduced from the domain provided by H1. We are more interested in other heuristics in Table 3.

For those entries with an empty set (∅), the related heuristics do not indicate any suspicious statements for fault localization and can be ignored. If an empty set is provided by H5, H12, or H11, the corresponding compliment heuristic ($\overline{\text{H5}}$, $\overline{\text{H12}}$, or $\overline{\text{H11}}$) will also be ignored because the search domain indicated by the compliment heuristic has the same size as the one suggested by H1. Because of the features of H5, H12, and H11 as mentioned in Section 3, these heuristics and their complement heuristics will be considered to find effective methods of applying them. For instance, the high percentage in Row b of H5 for program transp indicates the suggested domain is not reduced enough. Thus the domain suggested by $\overline{\text{H5}}$ should be examined first because of its small size. The low percentage in Row b of H12 and H11 for program gcd precisely indicates very few statements for correct results, and the semantics of these statements help us realize faults in the tested program. Thus $\overline{\text{H12}}$ and $\overline{\text{H11}}$ will be ignored because of their corresponding

Prog.		H3	H4	H7	H13	H14
aveg	a	80% (28/35)*	80% (28/35)*	66% (23/35)*	72% (25/35)*	26% (9/35)*
	b	100% (28/28)*	100% (28/28)*	83% (23/28)*	90% (25/28)*	33% (9/28)*
	c	100% (3/3)*	100% (3/3)*	34% (1/3)*	67% (2/3)*	–
calend	a	59% (17/29)*	59% (17/29)*	25% (7/29)*	25% (7/29)*	11% (3/29)*
	b	74% (17/23)*	74% (17/23)*	31% (7/23)*	31% (7/23)*	14% (3/23)*
	c	40% (2/5)*	40% (2/5)*	34% (1/3)*	29% (2/7)*	$*$
find1	a	31% (10/33)*	31% (10/33)*	73% (24/33)*	28% (9/33)*	28% (9/33)*
	b	40% (10/25)*	40% (10/25)*	96% (24/25)*	36% (9/25)*	36% (9/25)*
	c	67% (2/3)*	67% (2/3)*	50% (1/2)*	50% (2/4)*	–
find2	a	13% (4/33)*	13% (4/33)*	67% (22/33)*	13% (4/33)*	28% (9/33)*
	b	16% (4/25)*	16% (4/25)*	88% (22/25)*	16% (4/25)*	36% (9/25)*
	c	50% (2/4)*	50% (2/4)*	34% (1/3)*	25% (1/4)*	–
find3	a	13% (4/32)*	13% (4/32)*	66% (21/32)*	13% (4/32)*	29% (9/32)*
	b	17% (4/24)*	17% (4/24)*	88% (21/24)*	17% (4/24)*	38% (9/24)*
	c	50% (2/4)*	50% (2/4)*	34% (1/3)*	40% (2/5)*	–
gcd	a	67% (38/57)	2% (1/57)*	20% (11/57)*	2% (1/57)*	30% (17/57)
	b	98% (38/39)	3% (1/39)*	29% (11/39)*	3% (1/39)*	44% (17/39)
	c	100% (5/5)	17% (1/6)*	15% (1/7)*	12% (1/9)*	N/A
naur1	a	76% (28/37)*	76% (28/37)*	68% (25/37)*	63% (23/37)*	17% (6/37)*
	b	91% (28/31)*	91% (28/31)*	81% (25/31)*	75% (23/31)*	20% (6/31)*
	c	75% (3/4)*	75% (3/4)*	50% (1/2)*	60% (3/5)*	$*$
naur2	a	76% (28/37)*	76% (28/37)*	68% (25/37)*	60% (22/37)*	17% (6/37)*
	b	91% (28/31)*	91% (28/31)*	81% (25/31)*	71% (22/31)*	20% (6/31)*
	c	75% (3/4)*	75% (3/4)*	50% (1/2)*	20% (1/5)*	$*$
naur3	a	9% (3/36)	9% (3/36)	84% (30/36)	9% (3/36)	14% (5/36)
	b	10% (3/30)	10% (3/30)	100% (30/30)	10% (3/30)	17% (5/30)
	c	50% (2/4)	50% (2/4)	100% (4/4)	50% (2/4)	$*$
transp	a	30% (45/155)	1% (1/155)*	19% (29/155)*	1% (1/155)*	24% (36/155)
	b	100% (45/45)	3% (1/45)*	65% (29/45)*	3% (1/45)*	80% (36/45)
	c	100% (3/3)	25% (1/4)*	50% (1/2)*	20% (1/5)*	N/A
trityp	a	28% (10/37)*	28% (10/37)*	28% (10/37)*	11% (4/37)*	30% (11/37)*
	b	48% (10/21)*	48% (10/21)*	48% (10/21)*	20% (4/21)*	53% (11/21)*
	c	38% (3/8)*	38% (3/8)*	34% (1/3)*	25% (2/8)*	$*$

$*$: the highlighted statements contain faulty statements
a: # of highlighted statements / # of executable statements
b: # of highlighted statements / # of statements highlighted by H1
c: rank of the critical level / # of ranked levels
– : not effective enough

Table 4: Effectiveness analysis for heuristics with critical threshold requirements

high percentages.

Heuristics with threshold requirements are presented in Table 4. Statements involved in a heuristic with threshold requirements are first ranked according to the metric used by the heuristic (e.g., inclusion frequency) and are then grouped based on the ranks (i.e., statements with the same rank are in one group). Among different groups, the rank associated with a group that contains the fault is referred to as the *critical level*. The threshold of the heuristic should at least be set at the critical level to assure the suggested domain containing faults. This minimal threshold is referred to as the *critical threshold*. After the critical level is decided, suspicious statements are thus highlighted by the heuristic.

In Table 4, Rows c and b are the critical thresholds: the ratio of the rank of the critical level to the number of ranked levels and the ratio of suspicious statements within and below the critical level to statements involved in the heuristic (i.e., statements highlighted by H1), respectively. If faulty statements do not belong to any statements involved in the heuristic, the critical threshold will be 100% (e.g., the entry of H3 for Rows b and c of program transp in Table 4).

We use the entry of H3 (indicating statements with low inclusion frequency in $\bigcup_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i})$) on program calend as an example for further illustration. Five groups with different inclusion frequency are obtained from $\bigcup_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i})$) on calend: the first group (ranked 1) has the lowest inclusion frequency 1 and contains eight statements; the second group (ranked 2) contains nine statements with inclusion frequency 2; the third group (ranked 3) contains one statement with inclusion frequency 3; the fourth group (ranked 4) contains one statement with inclusion frequency 4; and the fifth group (ranked 5) contains four statement with inclusion frequency 5. The faulty statement (Statement 30) is in the second group. Thus, the critical level of H3 on calend is 2 (i.e., associated with the second group), and those seventeen statements (including the faulty one) in the first (having eight statements) and second (having nine statements) groups are highlighted. We then decide the critical threshold in Row c based on the ranks (the ratio of the rank of the critical level to the total number of ranked levels — 2/5) and the the critical threshold in Row b based on the statements (the ratio of suspicious statements within and below the critical level to statements suggested by H1 — 17/23).

Heuristic 14 is designed to enhance the location of faults in the predicate expression when other heuristics are not effective enough. Because H14 would consider results of other heuristics, a precise critical threshold is hard to define for it. In Table 4, Rows a and b for H14 indicate the number of predicate statements in a tested program divided by the number of the executable statements and the number of the statements highlighted by H1, respectively. Row c tells the effectiveness of using H14 to locate faulty predicate statements. "N/A" means that this heuristic is not applicable because faults of the corresponding tested program are not in predicate statements. "$*$" means that the faulty predicate statements can be located by using this heuristic combined with other heuristics. If this heuristic does not provide more effective information than other heuristics do for locating faulty predicate statements, "$-$" is marked in the entry.

4.3 Discussion

The determination of thresholds for heuristics with this requirement (e.g., H3, H4, H7, and H13) will affect the effectiveness and size of suggested domains. A unique threshold, which makes the suggested domain reasonably small and consistently contain faults, is highly desirable. In Table 4, critical thresholds for various heuristics in Row b range from 1% to 91%. A standard threshold cannot be decided within this wide scope. On the other hand, we find that most critical thresholds in Row c are below 50%. Moreover, thresholds based on ranked levels are easy to use (e.g., from the first to the last ranked level, gradually). Therefore, we conclude to choose thresholds based on ranked levels and using the first 50% of ranked levels as a standard threshold for the first–time criterion when employing these heuristics.

By comparing the size and percentage of search domains suggested by heuristics for each tested program, we found that for most cases the domains provided by heuristics with threshold requirement are more precise than those provided by heuristics without this requirement. Therefore, we prefer to examine the search domains suggested by heuristics with threshold requirements first (i.e., H4, H3, H7, and H13), although extra effort is needed to decide the critical thresholds. From Tables 3 and 4 it is hard to conclude which heuristic is the most effective one. Also it is not appropriate to make this conclusion because these heuristics are proposed to handle different situations. Program behavior and type of faults would affect the effectiveness of these heuristics. We can only provide a general approach to apply these heuristics.

For some types of faults (e.g., missing assignment), our heuristics cannot directly cover the faults in suggested domains. However, analyzing the statements highlighted by our heuristics (e.g., semantics of suspicious variables in the statements, the analysis approaches mentioned in [6]) can lead to the identification of the faults. For instance, the missing predicate statement in program naur3 can be located by analyzing the semantics of statements suggested by Heuristics in Table 4. If the faults of wrong initialization in program gcd and transp are changed to missing initialization statements, we still can indirectly locate the faults of missing initialization statements by analyzing the semantic of variables in suspicious statements highlighted by our heuristics. Other approaches to fault localization based on test knowledge (e.g., error–revealing mutations derived from mutation–based testing [34]) and dynamic program slices are currently under development by the authors. These approaches further will help fault localization.

5 Related Work

In this section, a brief survey of typical fault localization techniques is presented.

Traditional debugging techniques such as dumping memory, scattering print statements, setting breakpoints by users, and tracing program execution only provide utilities to examine a *snapshot* of program execution. Users have to use their own strategies to do fault localization.

Shapiro [39] proposed an interactive fault diagnosis algorithm, the *Divide–and–Query* algorithm, for debugging programs represented well by a computation tree (e.g., the logic programs written in Prolog). The computation tree (the target program) is recursively searched until bugs are located and fixed. Renner [37] applied this approach to locating faults in programs written in Pascal. With this method, users can only point out procedures that contain bugs; other debugging tools are needed to debug the faulty procedures. The similar result is obtained in [17].

The knowledge–based approach attempts to automate the debugging process by using techniques of artificial intelligence and knowledge engineering. Knowledge about both the classified faults and the nature of program behavior is usually required in this approach. Many prototype debugging systems have been developed based on this approach since the early 1980s. [14, 38] However, knowledge about programs in the real world is complicated. These prototype systems can only handle restricted fault classes and very simple programs.

Program slicing proposed by Weiser [43, 44] is another approach to debugging. This method decomposes a program by statically analyzing the data–flow and control–flow of the program — referred to as *static program slicing*. Program dicing, proposed by Lyle and Weiser [27, 45], attempts to collect debugging information according to the correctness of suspicious variables involved in static program slices. *Focus* [28] is a debugging tool based on program dicing to find the likely location of a fault. Because static program slices contain many irrelevant statements that make fault localization inefficient, studying program slicing based on dynamic cases to get the exact execution path is warranted. Dynamic Program Slicing [1, 3, 6, 25] is a powerful facility for debugging and dependency analysis. Nevertheless, it has not been systematically applied to fault localization. In Agrawal's dissertation [6], he briefly alluded to the idea of combining dynamic program slices and data slices for fault localization. Our heuristics are based on dynamic slices that are collected by varying test cases, variables, and location of variables.

Current testing and debugging tools are separate. Even if they are presented in one tool, the two functions are not well integrated to benefit each other. Osterweil [32] tried to integrate testing, analysis, and debugging, but gave no solid conclusion about how to transform information between testing and debugging to benefit each other. Clark and Richardson [11] were the first to suggest that certain test strategies (based on the symbolic evaluation) and classified failure types could be used for debugging purposes. However, only one example is given to describe their idea, and no further research has been conducted.

STAD (System for Testing and Debugging) [24] is the first tool to successfully integrate debugging with testing. As mentioned above, its testing and debugging parts do not share much information except for implementation purposes (e.g., they share the results of data flow analysis). The debugging part of STAD will be invoked once a fault is detected during a testing session, and leads users to focus on the possible erroneous part of the program rather than locate the fault precisely. PELAS (Program Error–Locating Assistant System) [23] is an implementation of the debugging tool in STAD. Korel and Laski proposed an algorithm based on hypothesis–and–test cycles and knowledge obtained from STAD to localize faults interactively [26]. However, STAD and PELAS only supported a subset of Pascal, and limited program errors are considered.

Collofello and Cousins [12] proposed many heuristics to locate suspicious statement blocks after testing. A program is first partitioned into many decision–to–decision paths (DD–paths), which are straight–line codes existing between two consecutive predicates of the program. Two test data sets are obtained after testing: one detects the existence of faults and the other does not. Then, heuristics are employed to predict possible DD–paths containing bugs based on the number of times that DD-paths are involved in those two test data sets. Their ideas are related to some of the heuristic fault localization strategies proposed by us. The deficiency of their method is that only execution paths (DD-paths), a special case of dynamic program slicing, are examined. After reducing the search domain to a few statement blocks (DD-paths), no further suggestion is provided for locating bugs.

Unlike the approaches proposed by Lyle–Weiser and Collofello–Cousins, which are only based on suspicious variables and test cases respectively, our heuristics are developed by considering test cases, variables, and location of variables together. We believe our methods will obtain more helpful information and reduce the search domain effectively.

6 Concluding Remarks

With the support of *Spyder* and the proposed heuristics, a new debugging scenario can be described as follows: (1) users find program failures by using the testing methodology provided by an integrated testing tool; (2) the debugging tool interactively helps users reduce the search domain for faults by dynamic instrumentation (e.g., dynamic program slicing and backtracking) and the information obtained from the testing phase; (3) the tool supports fault prediction strategies based on the reduced domain and test–based information; and (4) users can retest the program to assure that the program failure has been prevented after faults are located and fixed. The tool to support this new debugging scenario should be integrated with a testing environment and can conduct program dependency analysis, monitor execution history for backtracking, and provide fault prediction strategies based on information obtained from failure analysis and fault classification. [34]

In this paper, a set of heuristics is proposed to confine the search domain for bugs to a small region. The heuristics are based on dynamic program slices that are collected by varying test cases, variables, and location of variables. Preliminary results of our studies indicate the effectiveness and feasibility of the proposed heuristics. Although it is not guaranteed that faults can be found in the domains suggested by the proposed heuristics, a confined small region containing faults or information leading to fault discovery is provided for further analysis.

We continue to study the nature of program failures/faults as well as information obtained from testing methodology to further develop the foundation of our heuristics and to develop other promising approaches. We expect that our new debugging paradigm equipped with these heuristics will significantly reduce human interaction time and aid in the debugging of complex software.

Acknowledgements

The authors would like to thank Richard DeMillo for suggesting the idea of constructing the family tree for proposed heuristics in Section 3, Hiralal Agrawal for building the prototype debugging tool – *Spyder*, and Shi-Miin Liu and R. J. Martin for reviewing early drafts of this article. We also thank Bellcore for making *Atac* available for research use at Purdue. The 10th Pacific Northwest Software Quality Conference reviewers provided valuable comments and suggestions.

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the ACM SIGSOFT '91 Fourth Symposium on Testing, Analysis, and Verification (TAV4)*, pages 60–73, Victoria, British Columbia, Canada, October 8–10 1991.

[2] H. Agrawal, R. A. DeMillo, and E. H. Spafford. An execution backtracking approach to program debugging. *IEEE Software*, 8(3):21–26, May 1991.

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

[4] H. Agrawal and E. H. Spafford. An execution backtracking approach to program debugging. In *Proceedings of the 6th Pacific Northwest Software Quality Conference*, pages 283–299, Portland, Oregon, September 19–20 1988.

[5] H. Agrawal and E. H. Spafford. A bibliography on debugging and backtracking. *ACM Software Engineering Notes*, 14(2):49–56, April 1989.

[6] Hiralal Agrawal. *Towards Automatic Debugging of Computer Programs*. PhD thesis, Purdue University, West Lafayette, Indiana, August 1991. (Also released as Technical Report SERC-TR-103-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana, September 1991).

[7] ANSI/IEEE. *IEEE Standard Glossary of Software Enginnering Terminology*. IEEE Std 729–1983. IEEE, New York, 1983.

[8] R. S. Boyer, E. Elspas, and K. N. Levitt. SELECT — a system for testing and debugging programs by symbolic execution. In *Proceedings of International Conference on Reliable Software*, pages 234–245, 1975. (*ACM SIGPLAN Notices*, 10(6), June 1990).

[9] Gordon H. Bradley. Algorithm and bound for the greatest common divisor of n integers. *Communications of the ACM*, 13(7):433–436, July 1970.

[10] T. A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven, Connecticut, 1980.

[11] Lori A. Clarke and Debra J. Richardson. The application of error–sensitive testing strategies to debugging. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 45–52, Pacific Grove, California, March 1983. (*ACM Software Engineering Notes*, 8(4), August 1983; *ACM SIGPLAN Notices*, 18(8), August 1983).

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

[13] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–43, April 1978.

[14] Mireille Ducasse and Anna-Maria Emde. A review of automated debugging systems: Knowledge, strategies, and techniques. In *Proceedings of the 10th International Conference on Software Engineering*, pages 162–171, Singapore, April 1988.

[15] Kevin J. Dunlap. Debugging with DBX. In *UNIX Programmers Manual, Supplementary Documents 1, 4.3 Berkeley Software Distribution*. Computer Science Division, University of California, Berkeley, California, April 1986.

[16] Phyllis G. Frankl and Stewart N. Weiss. An experimental comparison of the effectiveness of the all–uses and all–edges adequacy criteria. In *Proceedings of the ACM SIGSOFT '91 Fourth Symposium on Testing, Analysis and Verification (TAV4)*, pages 154–164, Victoria, British Columbia, Canada, October 8–10 1991.

[17] Peter Fritzson, Tibor Gyimothy, Mariam Kamkar, and Nahid Shahmehri. Generalized algorithmic debugging and testing. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 317–326, Toronto, Canada, June 26–28 1991.

[18] M. Geller. Test data as an aid in proving program correctness. *Communications of the ACM*, 21(5):368–375, May 1978.

[19] J. B. Goodenough and S. L. Gerhart. Towards a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2):156–173, June 1975.

[20] F. Gustavson. Remark on algorithm 408. *ACM Transactions on Mathematical Software*, 4:295, 1978.

[21] C. Hoare. Algorithm 65: FIND. *Communications of the ACM*, 4(1):321, April 1961.

[22] J. R. Horgan and S. London. Data flow coverage and the C language. In *Proceedings of the ACM SIGSOFT '91 Fourth Symposium on Testing, Analysis, and Verification (TAV4)*, pages 87–97, Victoria, British Columbia, Canada, October 8–10 1991.

[23] Bogdan Korel. PELAS – program error-locating assistant system. *IEEE Transactions on Software Engineering*, SE-14(9):1253–1260, September 1988.

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

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

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

[27] James R. Lyle. *Evaluating Variations on Program Slicing for Debugging*. PhD thesis, University of Maryland, College Park, Maryland, December 1984.

[28] James R. Lyle and Mark Weiser. Automatic program bug location by program slicing. In *Proceedings of the 2nd International Conference on Computers and Applications*, pages 877–883, Beijing, PRC, June 1987.

[29] J. M. McNamee. Algorithm 408: A sparse matrix package (part I) [f4]. *Communications of the ACM*, 14(4):265–273, April 1971.

[30] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc., 1979.

[31] P. Naur. Programming by action clusters. *BIT*, 9:250–258, 1969.

[32] Leon Osterweil. Integrating the testing, analysis, and debugging of programs. In H. L. Hausen, editor, *Software Validation*, pages 73–102. Elsevier Science Publishers B. V., North–Holland, 1984.

[33] H. Pan and E. H. Spafford. Toward automatic localization of software faults. In *Proceedings of the 10th Pacific Northwest Software Quality Conference*, Portland, Oregon, October 19–21 1992.

[34] Hsin Pan. Debugging with dynamic instrumentation and test–based knowledge. Technical Report SERC-TR-105-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana, September 1991.

[35] Roger S. Pressman. *Software Enginnering: A Practitioner's Approach*. McGraw–Hill, Inc., second edition, 1987.

[36] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, SE-2(4):293–300, December 1976.

[37] Scott Renner. Location of logical errors on Pascal programs with an appendix on implementation problems in Waterloo PROLOG/C. Technical Report UIUCDCS-F-82-896, Department of Computer Science, University of Illinois at Urbana–Champaign, Urbana, Illinois, April 1982. (Also with No. UIUC-ENG 82 1710.).

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

[39] Ehud Y. Shapiro. *Algorithmic Program Debugging*. The MIT Press, Cambridge, Massachusetts, 1983. (PhD thesis, Yale University, New Haven, Connecticut, 1982).

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

[41] Iris Vessey. Expertise in debugging computer programs: A process analysis. *International Journal of Man–Machine Studies*, 23(5):459–494, November 1985.

[42] Chonchanok Viravan. Fault investigation and trial. Technical Report SERC-TR-104-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana, September 1991.

[43] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[44] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[45] Mark Weiser and Jim Lyle. Experiments on slicing–based debugging aids. In Elliot Soloway and Sitharama Iyengar, editors, *Empirical Studies of Programmers*, pages 187–197. Ablex Publishing Corp., Norwood, New Jersey, 1986. (Presented at *the First Workshop on Empirical Studies of Programmers*, Washington DC, June 5–6 1986.).

Appendix A: Notations and Terminology

Notation and terminology used in our proposed heuristics are as follows:

T_d : a set of error–revealing test cases that detect the existence of faults in P. $|T_d|$ represents the number of test cases in T_d.

T_u : a set of non–error–revealing test cases that do not detect the existence of faults in P. $|T_u|$ represents the number of test cases in T_u.

V_d (V_u) : a set of variables that have incorrect (correct) value with respect to the given location l, test case t, and P.

L_d (L_u) : a set of locations where the given variable v has incorrect (correct) value with respect to t and P.

$Dyn(P, v, l, t)$: a dynamic slice contains statements of P actually affecting the value of v at location l when P is executed against test case t.

$Dyn(P, v, l, T_{d_i})$: $1 \leq i \leq |T_d|$, $T_{d_i} \in T_d$, a dynamic slice with respect to the given P, v, l, and error–revealing test case T_{d_i}.

$Dyn(P, v, l, T_{u_i})$: $1 \leq i \leq |T_u|$, $T_{u_i} \in T_u$, a dynamic slice with respect to the given P, v, l, and non–error–revealing test case T_{u_i}.

$Dyn(P, V_{d_j}, l, t)$: $1 \leq j \leq |V_d|$, $V_{d_j} \in V_d$, a dynamic slice with respect to the given P, l, t, and variable V_{d_j} with incorrect value.

$Dyn(P, V_{u_j}, l, t)$: $1 \leq j \leq |V_u|$, $V_{u_j} \in V_u$, a dynamic slice with respect to the given P, l, t, and variable V_{u_j} with correct value.

$\bigcup_{i=1}^{|T_d|} Dyn(P, v, l, T_{d_i})$: union of the dynamic slices for different test case parameters with respect to all error–revealing test cases.

$\bigcup_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i})$: union of the dynamic slices for different test case parameters with respect to all non–error–revealing test cases.

$\bigcap_{i=1}^{|T_d|} Dyn(P, v, l, T_{d_i})$: intersection of the dynamic slices for different test case parameters with respect to all error–revealing test cases.

$\bigcap_{i=1}^{|T_u|} Dyn(P, v, l, T_{u_i})$: intersection of the dynamic slices for different test case parameters with respect to all non–error–revealing test cases.

Inclusion frequency of a statement: number of dynamic slices containing the statement.

Influence frequency of a statement in $Dyn(P, v, l, t)$: number of times the statement was visited in $Dyn(P, v, l, t)$.

Some slightly different notations are also used in this study: $Dyn(P, v, L_{d_k}, t)$, $Dyn(P, v, L_{u_k}, t)$, $\bigcup_{j=1}^{|V_d|} Dyn(P, V_{d_j}, l, t)$, $\bigcup_{j=1}^{|V_u|} Dyn(P, V_{u_j}, l, t)$, $\bigcap_{j=1}^{|V_d|} Dyn(P, V_{d_j}, l, t)$, and $\bigcap_{j=1}^{|V_u|} Dyn(P, V_{u_j}, l, t)$. Their interpretations are similar to those presented above.

Appendix B: Tested Programs

Source code of all tested programs (written in the C programming language) may be obtained from the authors.

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 working variable for the mean of the given integers has the value zero during calculation, then the result is incorrect.

Geller's calendar program calend [18], which was analyzed by Budd [10], tries to calculate the number of days between two given days in the same year. A wrong 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 [21] deals with an input integer array a with size $n \geq 1$ and an input array index f, $1 \leq f \leq n$. 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 [8], DeMillo–Liption–Sayward [13], and Frankl–Weiss [16]. 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 [9], which was also analyzed by Budd [10], 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 [19] analyzed an erroneous text formatting program (originally due to Naur [31]). 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 MAXPOS 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 [29], which was adopted for experiment by Frankl and Weiss [16], generates the transpose of a sparse matrix whose density does not exceed 66%. Two faults were identified in the original FORTRAN program. [20] 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, tritytp, is a well–known experimental program.[36] 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 $>$).