SOFTWARE DEBUGGING WITH DYNAMIC INSTRUMENTATION AND TEST–BASED KNOWLEDGE

A Thesis
Submitted to the Faculty

of

Purdue University

by

Hsin Pan

In Partial Fulfillment of the
Requirements for the Degree

of

Doctor of Philosophy

August 1993

To My Family.

ACKNOWLEDGMENTS

I would first like to express my grateful acknowledgement to my major professors, Richard DeMillo and Eugene Spafford, for their patience, support, and friendship. Professor DeMillo motivated me to study software testing and debugging, and who provided many helpful ideas in this research as well as the area of software engineering. The encouragement, valuable criticism, and expert advice on technical matters by my co-advisor, Eugene Spafford, have sustained my progress. In particular, I thank Professor Richard Lipton of Princeton University for suggesting the idea of Critical Slicing. I thank my committee members Professors Aditya Mathur and Buster Dunsmore for their suggestions and discussion.

I am also thankful to Stephen Chapin, William Gorman, R. J. Martin, Vernon Rego, Janche Sang, Chonchanok Viravan, Michal Young, and many others with whom I had valuable discussion while working on this research. My ex-officemates, Hiralal Agrawal and Edward Krauser, deserve special recognition for developing the prototype debugging tool, SPYDER. Bellcore's agreement to make ATAC available for research use at Purdue is acknowledged.

Finally, my sincere thanks go to my parents Chyang–Luh Pan and Shu–Shuan Lin, and my wife Shi–Miin Liu, for their love, understanding, and support. I am especially grateful for my wife's efforts in proofreading part of this thesis and her advice on writing.

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 NSF Grant CCR–8910306.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

APPENDICES

LIST OF TABLES

LIST OF FIGURES

Figure Page

Appendix
Figure

ABSTRACT

Pan, Hsin. Ph.D., Purdue University, August 1993. Software Debugging with Dynamic Instrumentation and Test–Based Knowledge. Major Professors: Richard A. DeMillo and Eugene H. Spafford.

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 dissertation focuses on the second task, reducing the search domain for faults, referred to as *fault localization*.

A new approach to enhancing the process of fault localization is explored based on dynamic program slicing and mutation–based testing. In this new scenario, a set of heuristics was developed to enable debuggers to highlight suspicious statements and thus to confine the search domain to a small region. A prototype debugging tool, SPYDER, was previously constructed to support the first task by using dynamic program slicing and the fourth task by backward execution; some of the heuristics were integrated into SPYDER to demonstrate our approach. An experiment confirms the effectiveness and feasibility of the proposed heuristics. Furthermore, a decision algorithm was constructed as a map to convey the idea of applying those heuristics. A new debugging paradigm equipped with our proposed fault localization strategies is expected to reduce human interaction time significantly and aid in the debugging of complex software.

1. INTRODUCTION

The existence of errors in software development is inevitable because of human inability to perform tasks or to communicate perfectly.[Deu79] Schwartz pointed out that in real world software systems, program errors are a fundamental phenomenon, and a bug–free program is an abstract theoretical concept.[Sch71] As the size and the complexity of programs increase, more errors are introduced during software development. Along with the rapid evolution of hardware, software plays an increasingly significant role in the success of many products, systems, and businesses.[Pre87] In the software life cycle, more than 50% of the total cost is expended in the testing and debugging phases.[Boe81, Mye79, Pre87] Developing effective and efficient testing and debugging strategies to guarantee the reliability of software is thus important.

In [ANS83], *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 the program is executed on a certain input because of the existence of *errors* and *faults*. In other words, a failure is a symptom of faults.

Testing explores the input space of a program that causes the program to fail, and debugging tries to locate and fix faults (bugs) after failures are detected during testing 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 [Dun86]) are command–driven symbolic debugging tools and tend to be stand–alone. Many fault localization strategies used in current well-known debugging tools (e.g., setting break-points and tracing) were developed in the 1960s and have changed little. [AS89] These debugging tools do not locate faults efficiently. Users have to discover by themselves information useful for debugging. Debugging processes are still labor–intensive, and these tools are thus little used.[Joh83] To debug software systematically, it is important that a tool use techniques based on the debugging process of experienced programmers, make hidden information available to users, and be directly accessible from the testing environment. However, these criteria have not been fully met by existing debugging tools.

1.1 Problems in Locating Faults

Two major steps involved in the debugging process are locating and then correcting faults. Myers [Mye79] pointed out that the fault–locating aspect represents 95% of the debugging effort. Several studies [GD74, Gou75, Ves85, Mye79, MM83, ST83], including behavioral research, suggest that locating faults is the most difficult and important task in the debugging process. Different strategies for locating faults would therefore affect the performance of debugging. [SCML79]

A representative paragraph from Martin [MM83] states:

> “Traditionally, programmers have spent too much time looking for errors in the wrong places. Myers [Mye78] found that programmers focused their attention on normal processing at the expense of considering special processing situations and invalid inputs. Weinberg [Wei71] found that programmers have difficulty finding errors because their conjectures become prematurely fixed, blinding them to other possibilities.
>
> Knowing what types of errors are likely to occur and where they are likely to occur in the program can avoid these problems and greatly simplify the debugging process.”

All these studies conclude that locating faults is the most human-intensive and expensive task in the debugging process.

1.2 Scope of This Research

The major goal of this research is to find an efficient way to accomplish the task of locating faults. By analyzing the debugging process used by experienced programmers, four

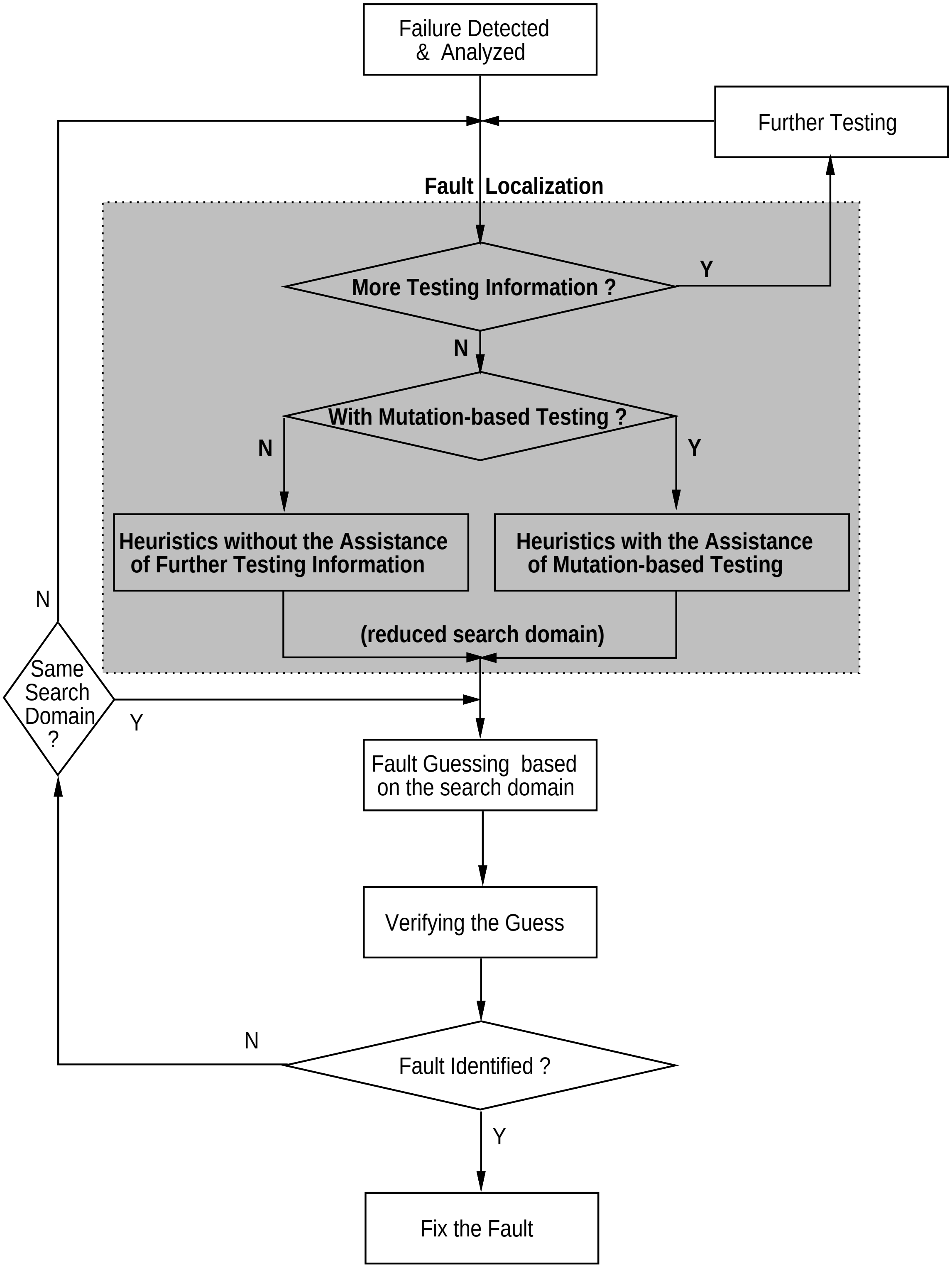

Failure Detected
& Analyzed
Further Testing
Fault Localization
More Testing Information ?
Y
N
With Mutation-based Testing ?
N
Y
Heuristics without the Assistance of Further Testing Information
Heuristics with the Assistance of Mutation-based Testing
(reduced search domain)
N
Same Search Domain ?
Y
Fault Guessing based on the search domain
Verifying the Guess
N
Fault Identified ?
Y
Fix the Fault

Figure 1.1 A new debugging scenario. Focus of this research, fault localization, is in the shaded area.

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 are performed with direct assistance from a debugging tool, the debugging effort becomes much easier. This dissertation focuses on the second task, reducing the search domain for faults, referred to as *fault localization*.

To assist users in conducting the first and last tasks, two techniques —*Dynamic Program Slicing* [ADS91a, AH90] and *Execution Backtracking* [ADS91b, AS88] — were developed by Agrawal, DeMillo, and Spafford. A prototype debugging tool, SPYDER [ADS91a, ADS91b, AS88, Agr91], has been constructed with those techniques. To achieve the first task, SPYDER helps users automatically find the dynamic slice of a program for any given variables, locations, and test cases in terms of data and control dependency analysis, where a dynamic slice of a program is the set of statements that actually affect the value of a selected variable at a specific location after the program is executed against a given test case.[1] For the fourth task, SPYDER provides a facility to restore the program state to a desired location by *backtracking* the program execution to that location and avoids the need to reexecute the program from the beginning.

In this dissertation, a new approach to enhancing the process of fault localization is explored based on dynamic program slicing and mutation–based testing.[2] In the new scenario, a set of heuristics was developed according to different situations and highlights suspicious statements to confine the search domain to a reduced region. Some of the heuristics were integrated into SPYDER to demonstrate the feasibility and usefulness of our approach.

Figure 1.1 depicts steps in the proposed debugging paradigm. A new direction in developing powerful debugging tools is suggested as a result of this study.

[1] An informal definition of Dynamic Program Slicing is described in Chapter 3.3. See [AH90, ADS91a, Agr91] for the formal definition.

[2] Program mutation has been studied for over a decade. It is well documented in the literature [CDK$^+$89, DGK$^+$88, BDLS80, Bud80, DLS78]. and is described in Chapter 5.

1.3 Contributions

The principal contribution of this dissertation 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, and does not guarantee that faults will always be precisely located in the reduced search domain. However, the reduced search space containing faults or the information leading to fault discovery will help users in debugging.

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. The effectiveness of integrating debugging and testing tools to enhancing the capability of locating faults has been demonstrated. From the testing point of view, this new approach can let the context be purposely switched between debugging and testing. Knowledge obtained from testing has been shown beneficial for debugging. This gives the testing field a new way to think of the usage of testing methodologies.

The feasibility of using test–based information to benefit debugging is demonstrated in this dissertation.

1.4 Organization of this Dissertation

The rest of this dissertation is organized as follows. A brief survey of software debugging techniques is given in the next chapter. Chapter 3 describes the analysis of our new debugging approach. We also define Expanded Dynamic Program Slicing, an enhancement of Dynamic Program Slicing, for debugging purposes. In Chapter 4, we propose a set of heuristics without the assistance of further testing information. These heuristics are based on test cases and dynamic program slices. Chapter 5 presents heuristics with the assistance of mutation–based testing. An effective debugging instrument, *Critical Slicing*, derived from the simplest mutant operator — statement deletion — is illustrated. In Chapter 6, we describe the integration of heuristics into a prototype debugging tool, SPYDER, along with results of a preliminary experiment that was conducted to confirm the effectiveness and feasibility of the proposed heuristics. From the experimental results, algorithms for applying the proposed approaches are presented in Appendix E. Finally, the conclusion of this dissertation and future directions of this research are given in Chapter 7.

2. RELATED WORK

We now briefly survey the debugging techniques used in existing software debugging tools.

2.1 Traditional Techniques

There are two “brute force” traditional debugging techniques according to Myer.[Mye79] One is analyzing memory dumps that usually display all storage contents in octal or hexadecimal format. The other is scattering print statements around suspicious places in a faulty program to display the value of variables and the flow of program execution. Users are expected to understand where the program goes wrong and the abnormal behavior of the program by using these techniques. However, these techniques are often inefficient because of a massive amount of data to be analyzed. It is time–consuming to verify the correctness of a prediction by repeating the process of executing and analyzing. While the technique of analyzing memory dumps is not often used now, the technique of scattering print statements is still employed, especially when debugging tools are not used.

One debugging technique — *setting break-points by users* — has been the main facility of many debugging tools for both low-level and high-level languages since the early 1960s (e.g., DDT [SW65] and FLIT [SD60] for assembly languages, and DBX [Dun86] for C). In order to avoid dealing with machine codes as well as scattering print statements in a program, interactive symbolic debugging tools were developed. These tools provide not only the basic facility, setting break-points, but also some of the following capabilities: displaying values of variables, tracing preset trace-points during execution, continuing execution of the program from a break-point, executing single steps from a break-point, modifying program states such as value of variables, and reexecuting a program (e.g., [Kat79, MB79, Bea83, Dun86]). These utilities can be employed by users to investigate a faulty program interactively. First, users can set break-points in the program. Then, the execution of the program will be suspended at these break-points. Users are allowed to examine the current program state (e.g., value of variables) at a suspended point and decide whether to examine the next statement, to continue forward execution, or to set new break-points.

These tools only provide utilities to examine a *snapshot* of program execution. Users have to conduct their own strategies to debug without producing a massive amount of information for analysis. These drawbacks increase the difficulty of debugging.

Another debugging technique keeps track of execution history for backtracking. The concept of execution backtracking has been implemented in database systems (e.g., rollback recovery for system failure [Ver78, HR83]) and in fault–tolerant software systems (e.g., the recovery–block technique [Ran75]). In the above implementations, system states to be rolled back for later analysis must be set at the beginning. However, from the software debugging standpoint, execution backtracking should be able to gradually backtrack program execution from any break-point under the control of users. EXDAMS [Bal69], for example, is an interactive debugging tool equipped with this technique.

The major problem with implementing backtracking is the consumption of large amounts of space for storing the execution history and program states. Agrawal, DeMillo, and Spafford [ADS91b] proposed a structured backtracking approach to implement execution backtracking without storing the whole execution history. Their approach saves the latest value of variables changed in a statement and only allows backtracking to the program state prior to a complete statement.

The idea of a user-friendly interface has been built into some debugging tools such as Dbxtool [AM86]. Windows and a mouse are used to handle the selections in debugging processes instead of the traditional command–driven approach which accepts typed commands only. Being able to display information (e.g., program execution flow, break-points, and program states) simultaneously makes the debugging process more convenient and efficient.

2.2 Algorithmic Approaches

Shapiro [Sha83] proposed an interactive fault diagnosis algorithm, the *Divide–and–Query* algorithm, for debugging. The algorithm will recursively search a computation tree that represents the target program until bugs are located and fixed. At each node of the computation tree, a query of node n will divide the tree rooted at node n into two roughly equal subtrees. If the result of an intermediate procedure call at node n is correct, the algorithm omits the tree rooted at n and iterates; otherwise the algorithm will be recursively applied to the two subtrees of node n. Shapiro proved that if a program is correct, every subprogram of the program is also correct because the computation tree of a subprogram is a subset of the computation tree of the whole program. If a program is not correct, then there exists at least one erroneous subprogram in it. The correctness of each intermediate procedure call at the node of a computation tree will be verified by users. This approach is suitable for debugging programs that can be well–represented as a computation tree. Logic programs, such as those written in Prolog, are the best candidates. A few enhanced debugging systems for Prolog programs have been developed based on this approach.[Per86, Fer87]

Renner tried to apply this approach to locating faults in Pascal programs.[Ren82] In order to verify the results of intermediate procedure calls mentioned above, an oracle is implemented to ask users the correctness of the return values of procedures in Pascal. Afterwards, the debugging system would investigate and execute each procedure in a top–down direction until a procedure fails or generates incorrect results detected by the oracle. A procedure containing bugs is thus located and the goal is achieved.

The primary limitation of applying this approach to programs written in structured languages such as Pascal is that it can only point out the procedure containing bugs. Other debugging tools are needed to debug the faulty procedure. A similar result can be found in [FGKS91].

2.3 Knowledge–Based Approaches

This approach attempts to automate the debugging process by using the techniques of artificial intelligence and knowledge engineering. Many existing automated prototype systems for program debugging have been developed based on this approach since the early 1980s.[DE88, Sev87] Knowledge of both the classified faults and the nature of program behavior is usually required in this approach. However, the knowledge required for real world programs is too complicated. The prototype systems mentioned in [DE88, Sev87] can only handle restricted fault classes and very simple programs. A few representative debugging tools using knowledge–based approaches are reviewed in this section.

PUDSY (Program Understanding and Debugging SYstem) [Luk80] analyzes a program before starting the process of debugging. Inputs to PUDSY are a program and a specification of the program. A knowledge base is maintained for the first phase — analyzing and understanding a program. The knowledge base keeps a set of programming schemas, which describe the simple typical behavior of each statement block (e.g., schema to find the maximum element in an array, and schema to exchange values of two elements). For each schema, there exists a set of assertions to formalize the function of that schema.

After the target program is automatically decomposed into chunks by heuristic methods, PUDSY will search the knowledge base to match programming schemas with those chunks. If a matched pair is found, the related assertion of the schema is constructed for the matched chunk of the given program. Then the debugging phase starts by comparing these assertions with the given specification. If any of these assertions violates the given specification, bugs are located in a corresponding chunk of code. Types of the bugs can be found by comparing the assertion with the given specification.

Because PUDSY needs the specification of the faulty program, only one subprogram (e.g., a procedure or a function) can be handled at a time. Otherwise, the corresponding specification for a complete program is too complicated to be described. The major limitation of this system is the variety of the programming schemas in the knowledge base. Only a few typical schemas can be representeded well.

Laura [AL80] uses a straightforward approach to debug students' programs. The specification of a program given to the system by teachers is the correct program model. The system compares programs written by students with the correct model. Programs for comparison are transformed into internal representation forms — flow graphs. Then, the flow graphs are systematically compared by *Laura* to automatically locate bugs. Because of the limitation of flow graphs to representing complicated programs, this system is designed for tutorial purposes in classrooms.

PROUST [JS84, JS85] is an "intention–based diagnosis system" to help novice programmers learn how to write correct programs. It does online analysis and understanding of Pascal programs in order to identify nonsyntactic bugs. PROUST takes a program and a description of the program's intentions as input. The description is not an algorithm; it is just a list of goals (intentions). The goals are then "synthesized" and implemented by "programming plans" retrieved from a knowledge base. A programming plan contains statements and subgoals to implement a typical function. Then, the synthesized goals are compared with the code of the given program. If the matching results are negative, all parties involved are analyzed. Diagnosis of the problem and suggestions for correcting the faults are reported to novices. However, if there are no proper programming plans to synthesize the given goal (intention), bugs are reported even if the code of the given program is correct. This problem becomes serious with complicated programs.

TALUS [Mur85, Mur86b, Mur86a] employs an approach similar to PROUST's, but uses a *theorem prover* to do the transformation from programming plans to codes. This method partially solves the problem of restricted programming plans in PROUST. However, TALUS can only handle small LISP programs.

Generally speaking, the above systems, classified as *program–analysis* techniques [Sev87], can only handle relatively small programs because they have to fully understand and *statically* analyze the programs to be debugged. The necessity of detailed understanding of programs prevents these techniques from being applied to practical programs.

Instead of statically analyzing the entire program, FALOSY (FAult LOcalization SYstem) [ST83] emphasizes fault localization by comparing the actual output with the expected output. Automatic fault location is performed by analyzing output discrepancies with the help of prestored fault cause–effect knowledge. Two major resources are maintained in the knowledge base. One is a set of heuristics describing the mapping between output discrepancies (fault symptoms) and possible causes. The other is a set of functional prototypes of certain standard program tasks (e.g., sorting). Based on output discrepancies, FALOSY searches the possible causes of the fault symptoms and compares the "functional prototypes" retrieved from its knowledge base with the given program. Then, the faults are reported in terms of the "prototype schema" (similar to the programming plans in PROUST) rather than the buggy code. Unfortunately, only a limited class of programs and faults are implemented in this system.

Harandi [Har83] presented a heuristic model for knowledge–based program debugging. The system aims to debug compile–time errors and certain run–time errors. It is assumed that most of the debugging knowledge of experienced programmers can be encoded as heuristic rules in the form of "situation–action" pairs. The "situation" part summarizes the possible symptoms of errors. The "action" part includes possible causes of errors and suggestions for fixing the errors. The system matches the present error symptoms and the information, which is provided by users or obtained through analysis of the program, with the situation part of the rules. Then, the corresponding actions of the rules are invoked. Because compilation errors can easily be fixed by experienced programmers using the error messages provided by the compiler, this system is mainly used for tutoring purposes.

2.4 Program Slicing Approach

Program slicing was proposed by Weiser [Wei82, Wei84] as another approach for debugging. Weiser's program slicing (also called *static* slicing) decomposes a program by statically analyzing data–flow and control–flow of the program. A static program slice for a given variable at a given statement contains all the executable statements that could influence the value of that variable at the given statement. The slice is a subset of the original program. The exact execution path for a given input is a subset of the static program slice with respect to the output variables at the given checkpoint. If a program fails on a test case and a variable is found incorrect at a statement, we can hypothesize that bugs are highly likely in the program slice regarding that variable–statement pair because only statements having influence on that variable–statement pair could cause the incorrect result. Thus, the search domain for faults is reduced from the entire program to the program slice.

Focus [LW87] is an automatic debugging tool based on static program slicing to locate bugs. It attempts to combine program slices with test cases to find the suspicious region containing faults, which is assumed to be a subset of a program slice. Two groups of slices are formed based on the program outputs after executing selected test cases — referred to as *program dicing* by Lyle and Weiser.[Lyl84, WL86] One group contains slices with respect to erroneous variables. The other group contains slices with respect to variables having correct values. *Focus* tries to confine the suspicious region for faults by choosing statements in the former slice group but not in the latter one for a given suspicious variable–statement pair. However, the slice generated by *Focus* contains many statements with no influence on the suspicious variable–statement pair because of the feature of static program slicing.

The static program slicing approach cannot resolve runtime ambiguities, thus highlights many spurious statements with no influence on the incorrect results. In this case, the faulty statements cannot be effectively identified.

Korel and Laski [KL88a, KL90] extended Weiser's static program slicing to dynamic program slicing (K-L slicing). K-L slicing defines an *executable* subset of an original program that computes the same function for given variables and inputs. In this case, their approach does not show the exact influence on a particular value of the given variable, location, and test case. For example, for a given variable V at the end of a loop and a given test data d, statement S_1 in the loop affects the value of V during the first iteration, and statement S_2 in the loop, not S_1, affects the value of V during the second iteration. According to the definition of K-L slicing, both statements S_1 and S_2 will be included in the K-L dynamic program slice for variable V and test data d in order to form an executable subprogram. However, if we want to know the exact influence on the particular value of V at the end of the second iteration, K–L slicing cannot give the information because of the existence of S_1 in this situation.

Agrawal and Horgan [AH90] claimed “the usefulness of a dynamic slice lies not [only] in the fact that one can execute it, but in the fact that it isolates only those statements that [actually] affected a particular value observed at a particular location.” They use the program dependency graph and an extended static slicing algorithm to construct dynamic program slices. Dynamic program slices obtained in this way are similar to those defined by Korel and Laski. Thereafter, in order to compute accurate dynamic slices such as in the example mentioned above, a dynamic dependency graph and specialized algorithms are employed. Accurate dynamic program slicing can isolate statements that actually affect a particular value of a suspicious variable at a given location for a given input. Thus, in the above example, the exact influence on the particular value of V at the end of the second iteration, which is statement S_2 not statement S_1, will be shown in the accurate dynamic program slice.

Dynamic program slicing for pointers based on the same approach has been implemented in the prototype debugging tool SPYDER [ADS91a, ADS91b, Agr91]. Prior the work reported here, dynamic program slicing had not been systematically applied to fault localization, although in Agrawal’s dissertation [Agr91] he briefly alluded to the idea of combining dynamic program slices and data slices for fault localization. Part of our heuristics are based on dynamic slices that are collected by varying test cases, variables, and location of variables.

2.5 Test–Integrated Support

The relationship between testing and debugging has never been clearly defined. Current testing and debugging tools are independent of each other. Even if they are integrated in one tool, strengthening the capability of this tool to detect and to locate faults needs to be seriously studied. This section will focus on the possibility of using the information derived from existing testing methodologies for debugging purposes.

Osterweil [Ost84] attempted to integrate testing, analysis, and debugging, but gave no solid conclusion about how to transform information between testing and debugging to benefit each other. An interesting result is suggested by Osterweil’s research: “debugging is probably best supported by a mix of static and dynamic capabilities, just as is the case for testing and verification." This points out a valuable direction for building new debugging tools.

Clark and Richardson [CR83] were the first to suggest certain testing strategies and classified failure types can be used for debugging purposes. Only one example was given to describe their idea. No further detailed study was conducted. They described how certain test data selection strategies based on symbolic evaluation [How77, CHT79, CR81] can help the debugging process. Typical error types classified by them include: erroneous reference to an input value; erroneous processing of special input values; erroneous processing of typical/atypical values; erroneous loop processing (e.g., never terminated, never executed); and erroneous production of special/typical/atypical output values. For each error type, test data are selected according to the result of symbolic evaluation. If an error in a program is detected after selected test data are applied to the program, we can know the potential type of the error and then locate the bugs through the attributes of the test data and the erroneous results.

STAD (System for Testing And Debugging) [KL88b, Las90, Kor86] is the first tool to integrate debugging with testing. Nevertheless, its testing and debugging parts do not share much information together except for implementation purposes (e.g., they share the results of data flow analysis). Information obtained from the testing phase for debugging purposes consists of the execution history of program failures and the test cases causing program failures. In this situation, the capability of STAD for locating bugs is no better than that of independent and unintegrated tools. The data flow testing methodology is the testing technique of STAD. The debugging part will be invoked once a fault is detected during a testing session.

STAD uses the structure of the program and the "execution trajectory" of a failure as its knowledge. The knowledge is reflected by a *program dependence network*, which is derived from dependency analysis (data flow and control flow dependency). A set of hypotheses indicating potential locations of faults is generated by using the program dependence network. Then, all hypotheses in the set are verified by users at preset break-points while reexecuting the program.

The main goal of fault localization in the debugging session of STAD is to help users focus on the possible erroneous region, rather than precisely locating faults. PELAS (Program Error-Locating Assistant System) [Kor88] is an implementation of the debugging part of STAD. Korel and Laski proposed an algorithm based on the hypothesis–and–test cycle and the above knowledge to localize faults interactively.[KL91] Only a subset of Pascal is supported, and limited program errors are considered in STAD and PELAS.

Collofello and Cousins [CC87] proposed a set of heuristics to locate suspicious statement blocks after a thorough test. A program is first partitioned into many decision–to–decision paths (DD–paths), which are composite statements existing between predicates. After testing, two test data sets are obtained: 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. The idea of these heuristics helped us develop our proposed fault localization strategies. The main restriction of their heuristics is that only execution paths that are a special case of dynamic program slicing [AH90] are examined. After the search domain is reduced to a few statement blocks (DD-paths), no further suggestion is provided for locating bugs.

2.6 Summary

Araki, Furukawa and Cheng summarized the debugging steps conducted by experienced programmers and proposed a debugging process model as a general framework. [AFC91] They also pointed out that “debugging tools must support each stage in the debugging process: hypothesis verification, hypothesis–set modification, and hypothesis selection.” However, they did not describe what kinds of facilities and functions will be used to support each stage.

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.

The evolution of software debugging has not made much progress during the last three decades. The most popular debugging techniques employed by commonly used debugging tools (e.g., DBX), setting break-points by users and tracing, were introduced around the early 1960s. [Sto67] 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.

3. ANALYSIS OF A NEW DEBUGGING APPROACH

Most studies of fault analysis [Knu89, BP84, OW84, Bow80, Lip79, AC73, JSCD83, SPL$^+$85] classify faults collected from long–term projects. However, no further studies have been conducted based on the categorized information. We believe that there is value to debugging research in such analysis.

In order to observe and analyze program failures, dynamic instrumentation capable of doing (dynamic) program dependency analysis is chosen as a basic facility of the new debugging approach. [Pan91] 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. At the same time, the knowledge derived from the testing phase (e.g., test cases and fault analysis from fault–based testing methodology) would be helpful to the debugging process.

In this chapter, the general background of our analysis is first described. Then, we enhance the existing dynamic program slicing technique by developing Expanded Dynamic Program Slicing (EDPS) that has better ability to include faulty statements for debugging. In addition, we conduct a generic analysis for test–based knowledge to help us construct debugging strategies based on available information from the testing phase.

3.1 Background

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 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 *error–revealing 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–error–revealing test cases*, and T_s is the set of them.

A set of error–revealing test cases is an indispensable resource for debugging. On the other hand, not every non–error–revealing test case is useful. Partition analysis on the input domain helps us identify the subdomains associated with program failures. [WO80, RC85, HT90, WJ91] After a thorough test, users can partition the input domain of P based on the specification of P and the results from testing. If an input subdomain of specification contains both error–revealing and non–revealing test cases after testing, then those non–revealing test cases are considered for debugging. Test cases in the input subdomain indicate the divergence between the expected and abnormal behavior of P after being executed against P. The divergence of program behavior with respect to the test cases provide valuable clues for fault localization.

If all test cases, which are constructed for the thorough test, in an input subdomain of specification are non–error–revealing ones, they are merely evidence to support the correctness of P for the certain input domain. It is likely that they do not provide direct help for locating faults. We thus ignore those non–error–revealing test cases at the beginning of debugging process to save effort. In order to reduce the size of suspicious input subdomains to a minimum, we prefer a well–defined input domain and specification of a given faulty program.

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 new debugging approach. Based on the selected test cases obtained from the partition analysis as mentioned above, we analyze the divergence among dynamic slices to understand program failures. This analysis helps us construct heuristics in the next chapter.

In brief, the general analysis can be summarized as follows.

- Employ a thorough test to collect as many relevant test cases as possible for an effective debugging process.

- Construct a well-defined input domain and specification of a given faulty program to reduce the suspicious input domain to a minimum. We can then deal with minimum dynamic program slices with respect to related test cases, and the task of fault localization will be more efficient.

- Focus on the smallest set of related test cases one at a time, especially for the non–error–revealing test cases.

- Conduct logical operations, such as intersection, union, and difference, on dynamic program slices with respect to selected test cases to study similarities or differences among these dynamic slices.

3.2 Heuristics vs. Deterministic Steps

Unlike testing, debugging is an unstructured, spontaneous, and "mystical" process.[Tra79, Lau79] 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, available test cases, etc. Deterministic decisions (the approach that systematically analyzes complicated information to benefit software debugging) may not be possible for the general case, and those for specific applications do not yet exist. We thus adopt a heuristic approach as a feasible solution to enhance the debugging process. We believe that heuristics, which gather useful information from different cases, can cover varied situations and help us localize faults.

The heuristic approach is like a forward reasoning technique to locate faults based on existing information. On the other hand, to construct deterministic steps, we need first examine all actual behavior and understand information flow made by faults, then induce systematical solutions for locating faults. This approach is like a backward reasoning technique. However, understanding all actual behavior and information flow made by faults is a difficult task. The exact mapping between results and causes for localizing faults is another obstacle to overcome. Thus the heuristic approach is a feasible way for debugging at the current stage.

Although each heuristic does not provide a general solution and is only suitable for some specific situations, the overall debugging power from uniting these heuristics is expected to surpass that of currently used debugging tools.

3.3 Expanded Dynamic Program Slicing

In this section, we enhance Dynamic Program Slicing [AH90, ADS91a, Agr91], which was chosen as our basic instrument, to handle certain types of faulty statements. An informal definition of Dynamic Program Slicing mentioned in [ADS93] is quoted and summarized as follows.[1]

> 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. Thus

[1] Readers are referred to [AH90, ADS91a, Agr91] for the formal definition.

> a dynamic program slice with respect to a given variable, location, and test case is the closure of all the data and control slices with regard to all expressions and locations in its constituent dynamic data and control slices.

In summary, a dynamic program slice, 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 the approach of Dynamic Program Slicing and refer to it as *exact dynamic program slicing.* Also, the static program slice proposed by Weiser [Wei82, Wei84] is represented as SPS, which is associated with a variable–location pair (v, l).

During the process of debugging, all statements making contribution to the erroneous variables that contain unexpected values are candidates for investigation. For the given P, v, l, t, we are interested not only in statements actually affecting/modifying the current value of v (i.e., statements in DPS(P, v, l, t)), but also statements contributing to keeping the defined variables in the dynamic slice with respect to v at l *intact*. That property is described in Definitions 3.1 and 3.2. Statements in the latter case may not be covered by DPS.

Definition 3.1 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 l) 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 and has *potential effect type 1* (PE1).[2] □

In other words, the predicate statement with type PE1 in the above definition keeps the value of v at l for t “intact”.

The example in Figure 3.1 indicates that the if–statement at Statement 5 *potentially affects* the output variable x of Statement 9 as defined in Definition 3.1. We say the if–statement at Statement 5 has type PE1. Statement 6 is in the scope of the predicate

[2] This definition is different from the *potential influence* defined by Korel [Kor88]. Statements covered by their definition include statements with type PE1 (potential effect) as well as in the exact dynamic slices (actual effect).

```
1: * !  input a;     /* input for a is  7 */
2: *#   x = 0;
3: *#!  j = 5;
4: * !  a = a - 10;  /* correct version  a = a + 10; */
5: * !  if (a > j) {
6:         x = x + 1;
        }
        else {
7: *       z = 0;
        }
8: *#   x = x + j;
9: *#   print x;     /* x is 5,  but should be 6 */
```

Figure 3.1 An example for the type of potential effect (PE). Statement labels with a star mark (*) are executed; those with a hash mark (#) are in the exact dynamic slice with respect to the output variable x at Statement 9; and those with an exclamation point (!) are in the potential effect slice with respect to the output variable x at Statement 9.

statement at Statement 5 (i.e., control dependent on the if–statement) and would affect the output variable x at Statement 9 if executed. Consequently, the unexecuted Statement 6 does not actually affect the output, and Statement 5 is thus not included in the exact dynamic slice (DPS($P, \mathtt{x}, 9, t$)) as marked by hash signs (#). However, Statement 5 still has a contribution to the output that involves keeping the value of the output variable *intact*. If we ignore this kind of contribution, then the faulty statement (Statement 4) which has the reaching definition on variable x at Statement 5 cannot be directly highlighted as a result. Therefore, the following definitions are warranted.

Definition 3.2 A *potential effect slice* (PES) with respect to the given program P, variable v, location l, and test case t contains all predicate statements with potential effect type 1 (PE1) as defined in Definition 3.1 as well as statements actually or potentially affecting the predicate statements. □

Definition 3.3 In a potential effect slice, statements other than the predicate statements with PE1 type are called *potential effect type 2* (PE2). All statements in a potential effect slice are called *potential effect type* (PE). □

In the example of Figure 3.1, the corresponding potential effect slice contains Statements 5, 4, 3, and 1. Statements 1, 3, and 4 have type PE2. This example shows that a faulty statement with PE type may not be covered by an exact dynamic slice (e.g., Statements 1, 4 and 5).

By using dynamic slices for debugging, we are concerned that faulty statements are either highlighted by a dynamic slice with respect to an output variable having an incorrect result for a given error–revealing test case or indirectly located by analyzing the dynamic slice. Unfortunately, it is not guaranteed that the associated exact dynamic slice always contains the faulty statements. We thus characterize the fault types that cannot be covered in an exact dynamic slice by analyzing program dependency and effect propagation between statements, and intend to develop an alternative approach to enhance the ability of including faulty statements.

The common feature of these fault types is that the effect of a faulty statement does not propagate to the point of failure occurrence or does not actually affect the incorrect result. In consequence, the faulty statement is not included in the corresponding exact dynamic slice. We already know statements with type PE1 in Definition 3.1 having this feature. Other types with this feature are defined as follows.

Definition 3.4 The type for statements of the missing statement fault type is called *omitted execution type 1* (OE1). □

Because the missing statements do not exist in the program, none of the program slicing techniques can directly highlight “faulty statements” with OE1 type.

Definition 3.5 A statement has *omitted execution type 2* (OE2) if and only if 1) the statement has an incorrect assignment variable (e.g., wrong left hand side variable — l-expression); 2) the missing definition of the correct assignment variable is the reason P

failed on a given error–revealing test case; and 3) the extra definition of the incorrect assignment variable does not *actually affect* the incorrect results. □

For instance, in a faulty statement with a wrong left hand side variable "$y = f(\ldots);$" whose correct version should be "$x = f(\ldots);$", the missing definition of the variable x causes P to fail on a given error–revealing test case. If the extra definition of the wrong variable y in the faulty program does not *actually affect* the incorrect result, then the faulty statement "$y = f(\ldots);$" will not be in the corresponding exact dynamic slice.

From the above examination of OE1, OE2, and PE types, the following theorems depict the ability of DPS to contain faulty statements.

Theorem 3.1 If a statement has type PE1 (i.e., predicate statement) with respect to the value of a given variable, location, and test case, then the statement is not contained by the exact dynamic program slice with respect to the variable, location, and test case.

Proof: According to Definition 3.1 for potential effect type 1 (PE1), executed statements in the scope of the predicate statement do not actually affect the current value of the given variable, location, and test case. Therefore, the predicate statement does not appear in the corresponding exact dynamic program slice (DPS). □

However, statements with type PE2 could be included in the corresponding exact dynamic program slice, e.g., Statement 3 in Figure 3.1 is in both the exact dynamic slice and the potential effect slice.

Theorem 3.2 If a given program has only one faulty statement that is not with type OE1, OE2, or PE, then the statement is contained by at least one exact dynamic program slice with respect to a selected erroneous variable at the location where the fault is manifested after executing an error–revealing test case.

Proof: Assume the faulty statement, which is the only fault in the given program and not with type OE1, OE2, or PE, is *not* contained by any exact dynamic program slices with respect to a selected erroneous variable at the location where the fault is manifested after executing an error–revealing test case.

According to the definition of DPS, statements in the corresponding exact dynamic slice actually affect the value of the selected erroneous variable. The reasons the faulty statement are not included by any corresponding exact dynamic slices are thus classified into two cases: 1) the faulty statement does not have any effect on the program failure; and 2) the faulty statement causes the program failure but does not appears in DPS.

The first case contradicts the single fault assumption which implies that the fault must have effect on the program failure. For the second case, the faulty statement either has the missing statement fault type or does not actually affect the program failure. The former one is type OE1 that contradicts the assumption in this proof which indicates the faulty statement not of type OE1. For the latter one, if the faulty statement potentially affects the program failure, then the fault statement has type PE that contradicts the assumption of this proof. Otherwise, the appearance of the faulty statement does not affect the program failure, but the missing definition of the defined (left hand side) variable which is replaced by the faulty statement causes the program failure. This is exactly the type OE2 that also contradicts the assumption of this proof. Thus, this theorem is proved. □

If the program is terminated with an incorrect output value, then the exact dynamic slice in the above theorem is associated with the erroneous output variables at the end of program execution. For the case of multiple faulty statements without OE1, OE2, or PE type, we slightly modify the above theorem as “the faulty statements are guaranteed to be in the *union* of exact dynamic program slices with respect to all erroneous output variables at the location where failures occurred.”

In order to enhance the ability of DPS to contain faulty statements, we examine a new dynamic slicing technique to cover the three types — PE1, OE1, and OE2. Unfortunately, faulty statements with OE1 type cannot be directly highlighted by any program slicing techniques. If the wrong left hand side variable of faulty statements with type OE2 has static dependency with output variables, then the faulty statements can be highlighted by Static Program Slicing (SPS), which is inefficient for debugging purposes. Otherwise, even

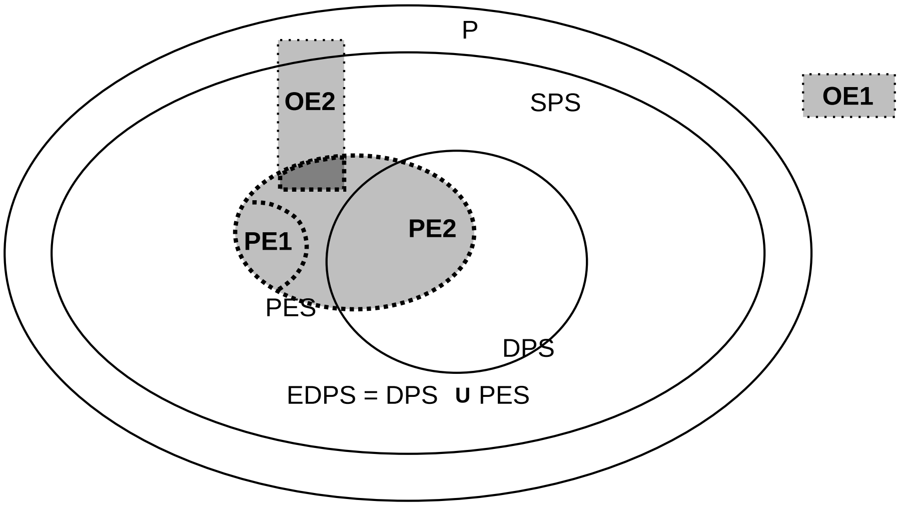


Figure 3.2 Relationships between DPS, EDPS, and SPS. Statements with special types are in grey area.

SPS cannot highlight the faulty statements with OE2 type. For statements with type PE, we develop Expanded Dynamic Program Slicing to include those statements that are in a potential effect slice but not in the corresponding exact dynamic program slice.

Definition 3.6 An expanded dynamic program slice (EDPS) is the union of an exact dynamic program slice (DPS) and the corresponding potential effect slice. □

A faulty statement with PE type and not in an exact dynamic slice will be included in a corresponding expanded dynamic slice. In Figure 3.1, the expanded dynamic slice contains Statements 1, 2, 3, 4, 5, 8, and 9 in which Statement 5 is the faulty one.

To use the EDPS approach effectively, we must know the relationships and features of DPS, EDPS, and SPS. Figure 3.2 presents the relationships,

$$\text{DPS} \subseteq \text{EDPS} \subseteq \text{SPS} \subseteq \text{P},$$

as well as those three types that cannot be highlighted by DPS. It is obvious that any program slice is a subset of the original program P, and any dynamic slice is a subset of the corresponding static slice. From the definition of EDPS (Definition 3.6), an exact dynamic slice is always a subset of the corresponding expanded dynamic slice. The above relationships are thus obtained.

Faulty statements with OE1 type do not exist in the given faulty program P at all. Also, it is not guaranteed that SPS can always contain faulty statements with OE2 type. If the wrong left hand side variable of faulty statements with OE2 type potentially affects output variables (i.e., with PE2 type), then the faulty statements can be included in a corresponding potential effect slice that is a subset of a corresponding expanded dynamic program slice. Otherwise, the faulty statements cannot be highlighted by the expanded dynamic program slice (EDPS). Thus, faulty statements with OE2 type may be highlighted by SPS or EDPS without guarantee as indicated in Figure 3.2.

Moreover, we claim that statements in an exact dynamic program slice (DPS) will not have the types of OE1, OE2, and PE1 regarding the same parameters of the exact dynamic program slice. Statements with PE type, which could not always be covered by DPS, can be included by EDPS.

3.4 Generic Analysis of Testing–Based Information

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 error–revealing test case) manifests the existence of faults and will be used for debugging later. However, we are interested not only in relevant error–revealing and non–error–revealing test cases, but also the testing criteria satisfied by the test cases (especially error–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. We thus define the following features of testing criteria from the standpoint of debugging.

Definition 3.7 If the execution of *every* test case that satisfies a selected testing criterion always causes program failure, then the testing criterion is called an *error–revealing criterion.* [WO80] □

In other words, it is guaranteed that any test case satisfying an error–revealing testing criterion causes program failure. Only error–revealing test cases can satisfy this criterion. This is the strongest condition for finding criteria revealing faults. Unless we already know the faults, an error–revealing criterion cannot be identified. Therefore, it is not feasible to use this criterion for debugging. A weaker and more feasible condition is evolved.

Definition 3.8 If the execution of a test case that is designed to satisfy a selected testing criterion causes program failure, then the criterion is called an *error–indicating criterion.* □

There is no guarantee that a test case satisfying an error–indicating testing criterion will definitely cause program failure. If we assume the failed program has only one faulty statement (i.e., single fault assumption), then the error–indicating criteria are the *necessary* conditions to reveal the fault after the criteria are satisfied. And yet, the error–revealing criteria are the *if and only if* condition to reveal the fault. Therefore, the error–indicating–criteria can be identified during the testing phase and used as indicators for debugging.

Three white box testing methodologies were examined in our study. They are structural coverage, data flow testing [PC90, HL91, CPRZ89, Nta88, FW88, RW85, LK83], and fault–based testing [DLS78, Bud80, BDLS80, How82, Mor90]. For the last two methods, we choose c–use and p–use criteria in data flow testing as well as program mutation[3] in fault–based testing. These two methods are superior to the first one as indicated in [HL91].

To examine the way error–indicating criteria are satisfied and the corresponding error–revealing test cases cause program failure, we employ three characteristics addressed in Constraint Based Testing [DO91, Off88], interpreted as follows for our analysis:

Reachability: The code specified by the criteria must be included in the program execution flow after applying a given error–revealing test case.

For data flow testing, this condition only ensures that the define part of a selected def–use pair criteria is reached.

[3] Program mutation has been studied for over a decade and is well documented in the literature [CDK$^+$89, DGK$^+$88, BDLS80, Bud80, DLS78]. It will not be discussed in detail here. A brief introduction will be addressed in Chapter 5.

Necessity: For a faulty statement, the local program state that is described by the value of variables becomes erroneous after the execution flow passes through the statement. However, the above claim about changing to erroneous local state after a criteria being satisfied is not always true for error–revealing and error–indicating criteria. This condition is modified for each testing method.

For statement coverage, this condition does not provide further requirement to be matched. We thus claim that this condition is automatically satisfied when reachability is achieved.

For data flow testing, the necessity condition indicates that the execution flow must pass through the corresponding use part after a selected define part is executed.

For mutation–based testing, local program states between a mutant and the original program right after the mutation statement is executed should be different in order to kill the mutant.

Sufficiency: The local effect generated by satisfying the above two conditions of selected criteria propagates to the end.

If the necessity condition is implied by the reachability condition (e.g., for coverage), then the sufficiency condition cannot be evaluated because of the unchanged local effect.

Analysis of reachability lets us focus on the program behavior while trying to reach the error–indicating criteria (i.e., checkpoints). Analysis of necessity helps us examine the cause of local effects. Then the sufficiency feature demonstrates how the local effect affects the program results. Information flow transfer among statements as mentioned in [TRC93] provides a similar approach to analyze the propagation of local effects. At each step, we explore the reasons causing program failures and try to discover relationships between the three characteristics and possible faults. As mentioned in Chapter 3.2, the debugging process tries to collect valuable information for locating faults. We are interested in those testing methods able to provide information of all three characteristics.

To satisfy a criterion in data flow testing, the execution of a selected test case must first reach the define part of the criterion, then follow the path to reach the use part. According to our interpretation of the necessity condition for data flow testing, both reachability (reaching the define part) and necessity (executing the use part) are required to claim that a criterion of data flow testing is satisfied. All three features — reachability, necessity, and sufficiency — must be matched to satisfy a criterion in mutation–based testing (i.e., killing mutants).

Information provided by reachability and necessity in data flow testing is also supported by the dynamic program slicing approach. The error–indicating criteria of data flow testing merely provide check-points in dynamic program slices for further examination. The error–indicating criteria of mutation–based testing introduce a mutant program that could be beyond the scope of the original program in terms of program dependency analysis. A combination of the three features plus the error–indicating criteria (mutants) supplies a valuable resource for further dynamic analysis. Therefore, we decided to explore information of mutation–based testing for debugging purposes and address the details in Chapter 5.

3.5 Summary

Analysis of the behavior of program failures and the type of faults provides valuable information for constructing effective heuristics for fault localization, especially when the deterministic steps mentioned in Section 3.2 do not exist. Static slices (SPS) contain statements that may have nothing to do with the wrong results. Dynamic slices generated from the DPS approach consist of statements actually affecting (modifying) the value of a variable occurrence for a given input. For statements potentially affecting the wrong result during execution, they are not always covered by DPS. The expanded dynamic program slicing technique (EDPS) includes such statements. By using dynamic instrumentation to observe program execution, DPS provides the smallest set of statements actually affecting the value of results, and EDPS provides a set of all statements having a contribution to the value of results. We thus prefer dynamic slices of EDPS as the basis instrumentation for debugging. The limitation of EDPS is that it is unable to include faulty statements with OE1 or OE2 type. The OE1 type (missing statements) cannot always be directly pointed out by any debugging approach. Yet, analyzing statements suggested by other heuristics, especially those statements in the statement block that has missing statement fault, could lead to identifying the nature of the missing statement. To handle the OE2 type, the error–indicating criteria of the test–based information could be a valuable resource to investigate.

For clarity and simplicity, we make the single fault assumption while developing the heuristics. This assumption helps us simplify the situation for thorough examination. However, all heuristics are still suitable for the case of multiple faults.

4. HEURISTICS WITHOUT THE ASSISTANCE OF FURTHER TESTING INFORMATION

Dynamic Program Slicing (DPS) determines the smallest set of statements actually affecting the value of results, and Expanded Dynamic Program Slicing (EDPS) determines statements actually contributing to the value of results. As mentioned in Chapter 3, we choose dynamic slices as the search domain to locate faults. The set of statements provided by EDPS will be used as a basis for the search domain.

We develop a family of heuristics to reduce the search domain according to dynamic slices (both DPS and EDPS) with respect to relevant test cases 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 none of our proposed heuristics is a general solution and each of them might be effective for certain specific situations only, the overall debugging power from uniting these heuristics is expected to surpass that of currently used debugging tools.

A dynamic slice $Dyn(P, v, l, t)$ that is either an exact dynamic program slice (DPS) or an expanded dynamic program slice (EDPS) contains four parameters: the target program P, a given variable v, the location of v (l), and a given test case 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 available dynamic slices and metrics. The issue of varying the program parameter (P) in $Dyn(P, v, l, t)$ is interpreted as getting dynamic slices from mutant programs, and will not be addressed here but in Chapter 5.

We define two metrics, *inclusion frequency* ($\mathcal{F}_c$) and *influence frequency* ($\mathcal{F}_f$). *Inclusion frequency* of a statement is the number of distinct dynamic slices containing the statement; and *influence frequency* of a statement in a $Dyn(P, v, l, t)$ is the number of times the

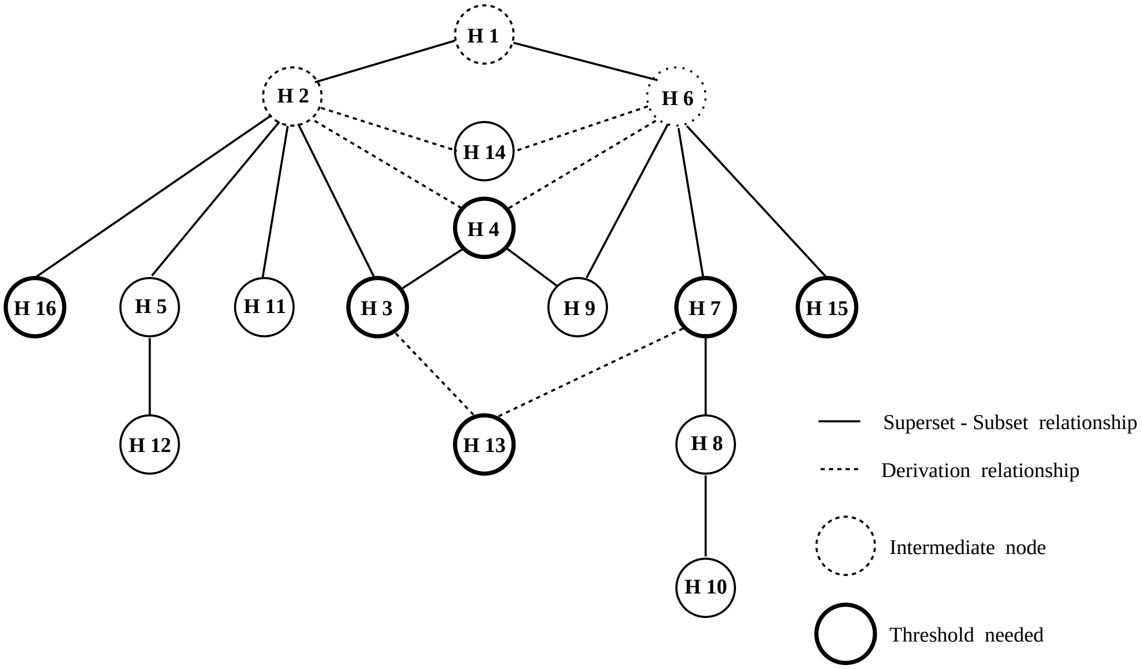


Figure 4.1 The family tree of proposed heuristics in Chapter 4.

statement was referred to in terms of data and control dependency in $Dyn(P, v, l, t)$. If a statement is executed many times (e.g., in a loop), then it is likely that the influence frequency of the statement in a dynamic slice is high.

Notation and terminology used in our heuristics are listed in Appendix A. In the next section, we examine the proposed heuristics. These are presented in Figure 4.1 as a family tree. Relationships among the heuristics in the family tree and the potential order of using them are explored in Section 4.2. A table of these heuristics can be found in Appendix B for reference purpose.

4.1 Heuristics based on Relevant Test Cases

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 easily, we first illustrate heuristics using the notations of different test case parameters (Heuristics 1 to 16). Within these heuristics, the set of dynamic slices with respect to error–revealing test cases (T_f) is called the *failure set* (i.e., $\bigcup_{i=1}^{|T_f|} Dyn(P, v, l, T_{f_i})$) and the one with respect to non–error–revealing test cases (T_s) is called the *success set* (i.e., $\bigcup_{i=1}^{|T_s|} Dyn(P, v, l, T_{s_i})$).

These heuristics can be applied to cases of different variable parameters, and another sixteen similar heuristics will be obtained (Heuristics 17 to 32). Then, three more heuristics (Heuristics 33 to 35) are proposed based on varying both test case and variable parameters together.

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.

Because many heuristics are constructed based on the two metrics (inclusion and influence frequency), a threshold to evaluate statements with low or high inclusion frequency in the heuristics is needed. For example, the threshold for statements with high inclusion frequency in the failure set ($\bigcup_{i=1}^{|T_f|} Dyn(P, v, l, T_{f_i})$) could be suggested as the highest twenty percent of statements in the union sorted by inclusion frequency. Users are allowed to set their own threshold for different purposes. 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. An efficient threshold, which makes the suggested domain reasonably small and consistently contain faults, is highly desirable for the first guess.

Statements involved in a heuristic with threshold requirements are first ranked according to the metric used by the heuristic (e.g., inclusion or influence frequency) and are then grouped based on the ranks (i.e., statements with the same rank are in one group/level). Among different groups (levels), the rank associated with a group (level) that contains the fault is referred to as the *critical level*. An example to describe ranks, groups, and the critical level is given in Chapter 6.2.3.2. For the first guess, the threshold of a heuristic is ideally set at the critical level to assure the suggested domain contains faults and is reasonably small. This minimal threshold is referred to as the *critical point* or *critical threshold.*

Heuristic 1 H1(t) $= \{ \bigcup_{i=1}^{|T_s|} Dyn(P, v, l, T_{s_i}) \} \cup \{ \bigcup_{i=1}^{|T_f|} Dyn(P, v, l, T_{f_i}) \}$ □

H1 indicates statements present in all available dynamic slices (with respect to all available non–error–revealing and error–revealing test cases) and is the basis of search domain for other heuristics.

Heuristic 2 H2(t) $= \{ \bigcup_{i=1}^{|T_s|} Dyn(P, v, l, T_{s_i}) \}$ □

H2 indicates statements present in all dynamic slices of the success set and makes users focus on statements in dynamic slices with respect to the non–error–revealing test cases (T_s). Faulty statements might be in these statements, if the fault is not triggered or not propagated to the result. Statements highlighted by this heuristics will be used by other heuristics.

Heuristic 3 H3($t, \mathcal{F}_c$) $=$ { statements in $\bigcup_{i=1}^{|T_s|} Dyn(P, v, l, T_{s_i})$ with inclusion frequency $\leq \mathcal{F}_c$ }, where $\mathcal{F}_c$ is an inclusion frequency number decided by the threshold given by users to select low inclusion frequency. □

H3 indicates statements with low inclusion frequency in all dynamic slices of the success set. The hypothesis for this heuristic is that a statement frequently involved with correct results may have less chance to be faulty. When there exist many non–error–revealing test cases (T_s) and very few error–revealing test cases (T_f), this heuristic can be employed to find faulty statements in the success set. The faulty statements, if any exist, would not lead P to wrong results when executing test cases in T_s.

Heuristic 4 H4($t, \mathcal{F}_c$) $= \{$ H3($t, \mathcal{F}_c$) $\cup$ H9(t) $\}$ □

H4 indicates statements in the failure set but not in the success set, plus statements with low inclusion frequency in dynamic slices of the success set. This is a modification of Heuristic 3. The inclusion frequency, based on the non–error–revealing test cases (T_s), for statements in the failure set but not in the success set is simply zero. The idea behind this heuristic is that faulty statements might never be executed when P is executed against non–error–revealing test cases (T_s). Thus, this approach is more flexible than H3 and would highlight more suspicious statements than H3 does. However, if all statements in the failure set are also in the success set, this heuristic is the same as H3.

Heuristic 5 H5(t) $=$ $\{ \bigcap_{i=1}^{|T_s|} Dyn(P, v, l, T_{s_i}) \}$ □

H5 indicates statements appearing in every success slice, i.e., statements with the highest inclusion frequency in the success set of H2, if the intersection is not an empty set. Examining 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 (i.e., $\overline{\text{H5}}(t)$ $=$ $\{$ H1(t) $-$ H5(t) $\}$).

Heuristic 6 H6(t) $=$ $\{ \bigcup_{i=1}^{|T_f|} Dyn(P, v, l, T_{f_i}) \}$ □

H6 indicates all statements in dynamic slices of the failure set and makes users focus on statements in dynamic slices with respect to the error–revealing test cases (T_f) — failure slices.

Heuristic 7 H7($t, \mathcal{F}_c$) $=$ $\{$ statements in $\bigcup_{i=1}^{|T_f|} Dyn(P, v, l, T_{f_i})$ with inclusion frequency $\geq$ $\mathcal{F}_c$ $\}$, where $\mathcal{F}_c$ is an inclusion frequency number decided by the threshold given by users to select high inclusion frequency. □

H7 indicates statements with high inclusion frequency in dynamic slices of the failure set. The hypothesis for this heuristic is that 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 (failure slices) and very few non–error–revealing test cases (success slices), this heuristic can be employed to find the faulty statements in the failure set.

Heuristic 8 H8(t) $=$ $\{ \bigcap_{i=1}^{|T_f|} Dyn(P, v, l, T_{f_i}) \}$ □

H8 indicates statements appearing in every dynamic slice of the failure set, i.e., statements with the highest inclusion frequency in the failure set of H7, 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 H9(t) $=$ $\{ \bigcup_{i=1}^{|T_f|} Dyn(P, v, l, T_{f_i}) \}$ $-$ $\{ \bigcup_{i=1}^{|T_s|} Dyn(P, v, l, T_{s_i}) \}$ (i.e., { H6 – H2 }) □

H9 indicates statements in the difference set of the failure set and the success set. The hypothesis for this heuristic is that statements involved in the execution of program failures but never tested by cases in T_s (i.e., statements only appearing in failure slices) are highly likely to contain faults. If there are many test cases in both T_f and T_s, this method is worth trying.

Heuristic 10 H10(t) $=$ $\{ \bigcap_{i=1}^{|T_f|} Dyn(P, v, l, T_{f_i}) \}$ $-$ $\{ \bigcup_{i=1}^{|T_s|} Dyn(P, v, l, T_{s_i}) \}$ (i.e., { H8 – H2 }) □

H10 indicates statements appearing in every failure slice but not in any success slice. This heuristic is similar to, but more rigorous than, H9 because only statements executed by *all* test cases in T_f are considered. Nevertheless, this difference set might be empty.

Heuristic 11 H11(t) $=$ $\{ \bigcup_{i=1}^{|T_s|} Dyn(P, v, l, T_{s_i}) \}$ $-$ $\{ \bigcup_{i=1}^{|T_f|} Dyn(P, v, l, T_{f_i}) \}$ (i.e., { H2 – H6 }) □

H11 indicates statements in the success set but not in the failure set. When P is executed against the non–error–revealing test cases (T_s), some statements might always be included to get correct results. Moreover, these statements never lead to the incorrect result. Further dependency analysis of these statements to understand how they contribute to correct results may provide useful information for locating faults. However, we could ignore these statements because they often contribute to correct results, and focus on other statements also suggested by Heuristic 1 (i.e., $\overline{\text{H11}}(t)$ $=$ { H1(t) – H11(t) }).

Heuristic 12 H12(t) $= \{ \bigcap_{i=1}^{|T_s|} Dyn(P, v, l, T_{s_i}) \} - \{ \bigcup_{i=1}^{|T_f|} Dyn(P, v, l, T_{f_i}) \}$ (i.e., { H5 − H6 }). □

H12 indicates statements appearing in every success slice but not in any failure slice. This heuristic is similar to, but more rigorous than, H11 because only statements executed by *all* test cases in T_s are considered. Examining the necessity of these statements for correct results might help us understand the nature of faults. Similar to H11, we can focus on other statements highlighted by $\overline{\text{H12}} = \{$ H1(t) − H12(t) $\}$.

Heuristic 13 Indicate statements in all dynamic slices with high inclusion frequency in the failure set and low inclusion frequency in the success set. □

This heuristic is a combination of Heuristics 3 and 7. It is suitable when many statements are involved in both failure ($\bigcup_{i=1}^{|T_f|} Dyn(P, v, l, T_{f_i})$) and success ($\bigcup_{i=1}^{|T_s|} Dyn(P, v, l, T_{s_i})$) sets, and many elements are in error–revealing (T_f) and non–error–revealing (T_s) test sets after a thorough test. The hypothesis for this heuristic is that 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 the failure set to the corresponding inclusion frequency in the success set 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 $\bigcup_{i=1}^{|T_s|} Dyn(P, v, l, T_{s_i})$ (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. □

Because the logical expression exp is always executed to decide whether B_1 should be executed, the 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 supplemental heuristic reminds users to examine the logical expression exp in the predicate statement.

Heuristic 15 H15($T_{f_i}, \mathcal{F}_f$) $=$ { statements in a $Dyn(P, v, l, T_{f_i})$ with influence frequency $\geq \mathcal{F}_f$ }, where $\mathcal{F}_f$ is an influence frequency number decided by the threshold given by users to select high influence frequency. □

H15 indicates statements with high influence frequency in a selected failure slice, $Dyn(P, v, l, T_{f_i})$. The influence frequency measures the effect of statements being referred to more than once (e.g., in a loop) but not counted in the inclusion frequency. The hypothesis for this heuristic is the same as that for H7 — statements often contributing to incorrect results are more likely to be faulty.

Heuristic 16 H16($T_{s_i}, \mathcal{F}_f$) $=$ { statements in a $Dyn(P, v, l, T_{s_i})$ with influence frequency $\leq \mathcal{F}_f$ }, where $\mathcal{F}_f$ is an influence frequency number decided by the threshold given by users to select low influence frequency. □

H16 indicates statements with low influence frequency in a selected success slice, $Dyn(P, v, l, T_{s_i})$. This heuristic, in other words, excludes statements with high influence frequency in the $Dyn(P, v, l, T_{s_i})$ because of the hypothesis that statements often leading to correct results are less likely to be faulty.

Heuristic 17 Based on the above heuristics, another sixteen similar heuristics (Heuristics 17 to 32) can be obtained by varying only the variable parameter v (instead of different test case parameters) with respect to the given P, l, and t, e.g., H#(v). □

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_f (having incorrect value) and V_s (having correct value). These sixteen new heuristics indicate potential faulty statements based on another feature of dynamic slices (the variable parameter).

Heuristic 33 H33(t, V_{f_j}) $=$ { $\bigcap_{j=1}^{|V_f|} Stmt(P, V_{f_j}, l, t)$ }, where t represents a selected test case in T_f or T_s, $Stmt(P, v, l, t)$ is the set of statements suggested by one of the Heuristics 1 to 16, and V_f is mentioned in Heuristic 17. □

The hypothesis for this heuristic is similar to that for H8. If wrong variable values are caused by the same fault, statements indicated by this heuristic are suspicious.

Heuristic 34 H34(v, t) $=$ $\{ \bigcup_{j=1}^{|V_f|} Stmt(P, V_{f_j}, l, t) \} - \{ \bigcup_{j=1}^{|V_s|} Stmt(P, V_{s_j}, l, t) \}$, where v is one of V_{f_j} or V_{s_j}. □

This heuristic is derived from H9 with information obtained from varying both variable and test case parameters.

Heuristic 35 H35(v, t) $=$ $\{ \bigcap_{j=1}^{|V_f|} Stmt(P, V_{f_j}, l, T_X) \} - \{ \bigcup_{j=1}^{|V_s|} Stmt(P, V_{s_j}, l, T_X) \}$, where v is one of V_{f_j} or V_{s_j}. □

This heuristic is derived from Heuristics 10 and 34.

From the above discussion, we notice that appropriate thresholds for some heuristics cannot be decided without observing real examples. In our experiment presented in Chapter 6.2, we try to find the average critical point that will be recommended for the first guess. Two kind of thresholds are measured: *rank threshold* — ratio of the rank of the critical level to the number of ranked levels, and *general threshold* — ratio of suspicious statements within and below the critical level to statements involved in the selected heuristic. From the results of our experiment, we discover that the rank threshold (based on ranked levels) is more reasonable than the general threshold (based on number of statements) and is easy to use (e.g., from the first to the last ranked level, gradually). Therefore, we suggest using the rank threshold with the first 75% (or 50%) of ranked levels as a standard threshold for the first–time criterion when employing the heuristics with inclusion frequency requirements. Detailed results of our experiment are presented in Chapter 6.2.3.

Heuristics and experiments according to relational (decision–to–decision) path analysis on execution paths were studied by Collofello and Cousins [CC87]. 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, possible faulty statements suggested by our heuristics should be more 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 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.

4.2 Further Analysis

In this section, an overall analysis to study effective methods for applying heuristics in Figure 4.1 is examined. We first examined heuristics with different test case parameters t (Heuristics 1 to 16). The same argument based on Figure 4.1 can then be applied to heuristics with different variable parameters (indicated in Heuristic 17). In Figure 4.1, 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, H1 is the root of the family tree, H2 is the root of a subtree with respect to non–error–revealing test cases, and 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 as a supplement 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 H1). In this case, $\overline{\text{H5}}$, $\overline{\text{H12}}$, and $\overline{\text{H11}}$ represent the corresponding complement 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 4.1, 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 fewer statements than those provided by heuristics in Group 1. The top–down order of employing these heuristics is 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.

There is another way to apply the heuristics before starting analysis. We can first examine the intersection of statements suggested by Heuristics 1 to 13, except H5, H12, and H11. The intersection is a minimum region indicated by the heuristics, if it is not empty. If the faulty statements are not in the region, then we start the global analysis. This method is interpreted as a *meta–heuristic* of the heuristic family tree.

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 parameters (Heuristics 33 to 35).

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. [Vir91]

4.3 Summary

We have presented a set of heuristics based on dynamic slices with respect to relevant test cases that are obtained from a thorough test. Analysis of relationships between the proposed heuristics was also examined, and a family tree was obtained as a result. The possible methods of applying the heuristics by traversing the family tree as well as grouping features of the heuristics were suggested. An experiment is described in Chapter 6.2 to show the effectiveness of the heuristics.

5. HEURISTICS WITH THE ASSISTANCE OF MUTATION–BASED TESTING

As discussed in Chapter 3, we chose program mutation as a testing methodology for debugging purposes and dynamic program slicing as our instrument for detailed analysis. A brief description of mutation–based testing is introduced here.

The principal goal of program mutation[1] 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.

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 is implied for that test set. Users would 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.

In Table 5.1, mutant operators of the MOTHRA [CDK$^+$89, KO91] testing tool for FORTRAN 77 are listed. Mutant operators for C have been reported in [ADH$^+$89], and are a

[1] Readers are referred to [CDK$^+$89, DGK$^+$88, BDLS80, Bud80, DLS78] for details of program mutation.

Table 5.1 Mutant operators used by MOTHRA for FORTRAN 77.

Op.‡	Description	Example†
aar	array reference for array reference replacement	A(I)=B(J) → A(I)=A(I)

abs	absolute value insertion (special value coverage)	X=Y+Z → X=abs(Y)+Z
		X=Y+Z → X=negabs(Y)+Z
		X=Y+Z → X=zpush(Y)+Z

acr	array reference for constant replacement	X=1 → X=A(I)

aor	arithmetic operator replacement	X=Y+Z → X=Y−Z
		X=Y+Z → X=Y
		X=Y+Z → X=Z

asr	array reference for scalar variable replacement	X=Y+Z → X=A(I)+Z
car	constant for array reference replacement	A(I)=B(J) → A(I)=0
cnr	comparable array name replacement	A(I)=B(J) → A(I)=A(J)

crp	constant replacement (twiddle)	X=3 → X=2
		X=3.0 → X=3.3

csr	constant for scalar replacement	A(I)=B(J) → A(3)=B(J)
der	DO statement end replacement	do 10 I=1, N → onetrip 10 I=1, N
dsa	data statement alterations	data X/1/ → data X/2/
glr	goto label replacement	goto 10 → goto 20
lcr	logical connector replacement	X.and.Y → X.or.Y
ror	relational operator replacement	X.eq.Y → X.gt.Y
rsr	return statement replacement	X=Y+Z → return
san	statement analysis (replacement by trap)	X=Y+Z → trap
sar	scalar variable for array reference replacement	A(I)=B(J) → A(I)=X
scr	scalar for constant replacement	X=1 → X=Y
sdl	statement deletion	X=Y+Z → continue
src	source constant replacement	X=1 → X=3
svr	scalar variable replacement	X=Y+Z → X=X+Z
uoi	unary operator insertion	X=Y+Z → X=++(X+Z)

† $S_1 \rightarrow S_2$ means S_2 is the mutation statement of the original statement S_1 in a given program after the corresponding mutant operator in the first column is applied. Abs(Y) and negabs(Y) return the absolute value of Y and the negative of the absolute value of Y, respectively. Zpush(Y) returns Y if Y≠0, otherwise the mutant is killed. Onetrip is identical to a DO statement except whose body is always executed at least once. Trap causes the mutant to be killed once the trap is executed. Twiddle operators (crp) will increment/decrement the integer constant by 1 and the real constant by 10% of its value, and will replace the zero by 0.01 and -0.01. The uoi operator, ++exp ($--exp$), increments (decrements) exp by 1 if exp is an arithmetic expression, and complements exp if exp is a logical expression.

‡ Mutant operators. Any variable (scalar or array) or constant replacement is done by using other variable names or constants occurring in the given program.

superset of the mutant operators for FORTRAN 77 except for the DATA statement alteration (dsa) that is not supported in C. Because a complete mutation–based testing tool for C does not exist, we first study the 22 mutant operators of MOTHRA in Table 5.1. The same analysis can be applied to mutant operators for C.

Because killing mutants is the criterion to be satisfied in mutation–based testing, we focus on the features of different mutant types and the different behavior between mutants and the original program. To analyze information from mutation–based testing by using the dynamic program slicing technique, we classify the mutants by regrouping mutant operators from the standpoint of program dependency (data and control flow) analysis.

1. *Statement analysis*: san and sdl. Mutants with san type force testers to create test cases for executing all statements/blocks for statement coverage, and sdl mutants help testers decide whether the selected statements actually make a difference in the program results. A class mutant operator stm — statement mutants — comprises these two mutant operators.

2. *Domain perturbation*: aor, src, abs, crp, dsa, and uoi. Mutants in this category will change the constant or operator in the r–expression (e.g., right hand side) of a selected original statement or in the logical expression of a selected original statement without introducing new variables. Perturbing the index of array reference is not considered as introducing new variables because the index is evaluated at run time. Therefore, the static data and control dependency of mutants are the same as the original program. The last four operators belong to a class mutant operator dmn — domain perturbation — that twiddles the value of the right hand side expression (r–expression) without changing program dependency.

3. *Operand (variable or constant) replacement on the use–part (right hand side) of a selected original statement*: aar, acr, asr, car, cnr, crp, csr, sar, scr, src, and svr where crp and src are included in this category only when the twiddled constant is in an array reference. The program dependency graph above the mutation (S_M) is

different from the one of the original statement (S_P) because of the replacement of the use–part.

4. *Operand (variable) replacement on the define–part (left hand side) of a selected original statement*: aar, acr, asr, car, cnr, crp, csr, sar, scr, src, and svr where acr, car, crp, csr, scr, and src are included in this category only when the replaced operand is in an array reference. The program dependency graph above the mutation (S_M) is still the same as the one of the original statement (S_P), but statements beyond and dependent on S_M are different from those of S_P because of the replacement of the define–part.

5. *Control dependency variation*: glr, lcr, ror, rsr, and the mutant operators that replace an operand of a predicate statement. The evaluation of the mutation predicate statement and the original predicate statement might cause different control flow between the corresponding mutants and the original program. lcr and ror belong to a class mutant operator prd — predicate mutants — that replace logical connectors and operators.

By examining the above mutants with the three characteristics mentioned in Chapter 3 — reachability, necessity, and sufficiency — as well as dynamic program slices with respect to corresponding test cases, we develop a set of heuristics (hints) for debugging. The three characteristics for killing mutants have been studied in the *constraint–based testing* [Off88, DO91] and are summarized here.

- Reachability: The mutated statement on line S in the mutant program M, S_M, is executed by a given test case.

- Necessity: The program state of M immediately following the execution of S_M is different from the program state of P at the same point S_P.

- Sufficiency: The final result of M is different from P.

Constraint–based testing will automatically generate test cases by satisfying the reachability and necessity conditions.

A simple and effective approach derived from studying mutants in the first group (statement analysis), referred to as *Critical Slicing*, is addressed in the next section. Other approaches will be presented later in this chapter.

5.1 Critical Slicing

While exploring san and sdl mutants, we are interested in the actual effect made by each statement, especially when programs are executed by error–revealing test cases. Faulty statements are likely in those statements that directly contribute to program failures. The dead sdl mutants identify a set of statements actually making a difference in program results and being critical to the program failures when the sdl mutants are killed by error–revealing test cases. The set of statements helps us develop Critical Slicing.

5.1.1 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 error–revealing test case t with a failure type $\mathcal{F}_i$, where S_i represents one statement. $\mathcal{F}$ is the set of different types of failures, and S_F is the statement where the failure $\mathcal{F}_i$ 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 wrong output variables, the incorrect values of the erroneous output variables are considered as features of the related failure type.

Definition 5.1 A statement S_i in S of P is *critical* to a selected variable v in the failure $\mathcal{F}_i$ at location S_F for test case t if and only if the execution of P without S_i (i.e., an sdl 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 $\mathcal{F}_i$. □

This means that not only is M killed because the execution against t has a different result from the original one in $\mathcal{F}_i$, 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 of the failure type. 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 5.2 A *Critical Slice* is based on a 4–tuples $\langle \mathcal{F}_i, v, S_F, t \rangle$ with definition CS$(\mathcal{F}_i, v, S_F, t) = \{S_i \mid S_i$ is critical to the variable v in the failure $\mathcal{F}_i$ at location S_F for test case $t\}$. □

5.1.2 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 between critical slices and other dynamic slices (e.g., exact dynamic slices and expanded dynamic 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 Expanded Dynamic Program Slicing, EDPS), we have to ensure parameters involved in both approaches are in the same domain. A dynamic slice, $Dyn(P, v, l, t)$, has four parameters — program P, variable v, location l, and test case t, where P will not be varied. Meanwhile, a critical slice, CS$(\mathcal{F}_i, v, S_F, t)$, also has four parameters — failure type $\mathcal{F}_i$, 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 $\mathcal{F}_i$. However, if there are multiple variables (e.g., in a set V_f) involved in $\mathcal{F}_i$, then we consider the union of slices with respect to *all* involved variables at location S_F for test case t, i.e., $\bigcup_{v \in V_f} Dyn(P, v, S_F, t)$ vs. $\bigcup_{v \in V_f}$ CS$(\mathcal{F}_i, v, S_F, t)$.

```
1: * !   input a;     /* input for a is  7 */
2: *#    x = 0;
3: *#!$  j = 5;
4: * !$  a = a - 10;  /* correct version  a = a + 10; */
5: * !$  if (a > j) {
6:          x = x + 1;
         }
         else {
7: *        z = 0;
         }
8: *# $  x = x + j;
9: *# $  print x;     /* x is 5,  but should be 6 */
```

Figure 5.1 An example derived from Figure 3.4 with indication (dollar sign $) of a critical slice. Statement labels with a star mark (*) are executed; those with a hash mark (#) are in the exact dynamic slice with respect to the output variable `x` at Statement 9; and those with an exclamation point (!) are in the potential effect slice with respect to the output variable `x` at Statement 9. Statements with a hash mark (#) or exclamation point (!) are in the expanded dynamic slice. Statement labels with a dollar sign ($) are in the critical slice with respect to the failure with wrong output value 5 for input data 7 at location Statement 9.

5.1.2.1 Relationships among CS, DPS, and EDPS

As the following two examples demonstrate, Critical Slicing (CS) and Exact Dynamic Program Slicing (DPS) are incomparable.

Example 5.1 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 5.1 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 5.2 If a predicate statement is PE1 type, then the statement will not be in the corresponding exact dynamic program slice as mentioned in Definition 3.1 (definition of the potential effect type I). However, 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 5.1 has PE1 type, and is in the critical slice but not in the exact dynamic slice. □

The relationship between Critical Slicing (CS) and Expanded Dynamic Program Slicing (EDPS) is described in the following theorem.

Theorem 5.1 If a given program does not have pointer or array variables (i.e., only scalar variables involved), then statements in a critical slice are a subset of statements in the corresponding expanded dynamic program slice, i.e., $CS \subseteq EDPS$.

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. Without pointer or array references, the reaching definition between scalar variables is straightforward.

For the first case — actually affecting the results — statements are included in the corresponding exact dynamic slice and expanded dynamic slice according to the definition of DPS and EDPS, respectively. For the second case — potentially affecting the results

— statements are included in the corresponding expanded dynamic slice according to the definition of EDPS. Therefore, CS is a subset of EDPS for a pointer (array) free program. □

However, it is not guaranteed that the same property is true for a program with unconstrained pointers (arrays). An example (statements with array reference) is given for illustration.

Example 5.3 In the following simple program, array variable `a(k)` at Statement S_4 is actually `a(3)` during execution. The output of this program is expected to be 21, but is actually 20 because of the faulty statement S_5. The expanded dynamic program slice with respect to erroneous output variable `x` at Statement S_6 for test case t includes Statements S_1 and S_5. This indicates that Statements S_2, S_3, and S_4 neither actually nor potentially affect the incorrect value of `x` (20) at Statement S_6.

```
S1:  read a(1), a(2), a(3);   /* test case t:  10, 20, 30 */
S2:  k = 2;
S3:  k = k + 1;
S4:  a(k) = a(k) + 5;
S5:  x = a(2);        ⟸  the correct version is  x = a(2) + 1;
S6:  print  x;
```

Meanwhile, a critical slice with respect to the failure (incorrect value of output variable `x` = 20) at Statement S_6 for test case t includes Statements S_1, S_3, and S_5. When we verify whether Statement S_3 is in the critical slice by not executing the statement, the value of `k` at Statement S_4 is 2. Thus, `a(k)` at S_4 is actually referring to `a(2)`, and the output of the program (value of `x`) becomes 25. This indicates that Statement S_3 is critical to the program failure and is thus in the critical slice. As a result, Statement S_3 is in the critical slice but not in the corresponding expanded dynamic slice. □

On the whole, relationships between Critical Slicing (CS), Exact Dynamic Program Slicing (DPS), and Expanded Dynamic Program Slicing (EDPS) are demonstrated in Figure 5.2 which integrates CS into Figure 3.2. For the case of statements in CS but not in

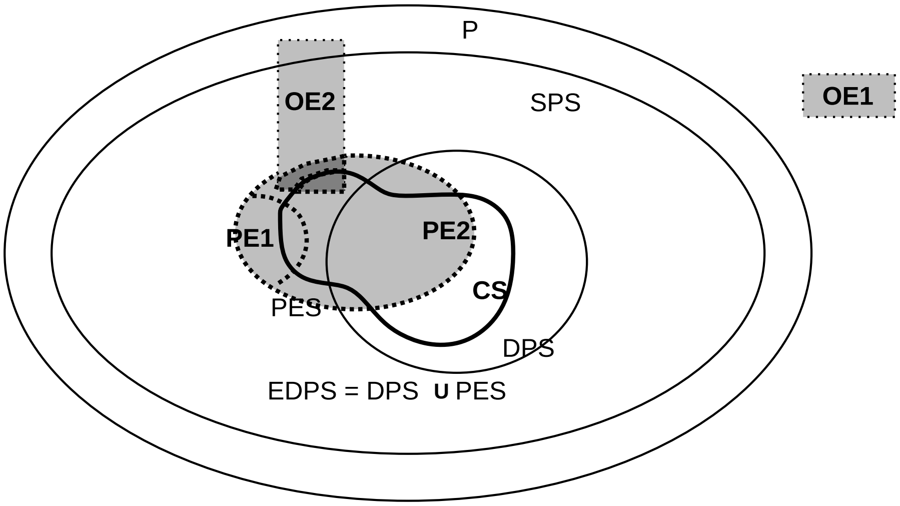


Figure 5.2 Relationships between CS, DPS, EDPS, and SPS for programs without array and pointer variables. Statements with special types are in grey area. This figure is an expansion of Figure 3.5.

EDPS (e.g., Statement S_3 in Example 5.3), this is caused by side–effects of array (pointer) references — `a(k)` at Statement S_4. Faulty statements not in EDPS have either OE1 or OE2 type, and statements with these two types are rarely highlighted by slicing techniques. In Chapter 3, we have chosen dynamic slices of EDPS as the bottom line dynamic instrumentation for debugging. We thus treat statements in CS but not in EDPS as noise for debugging purposes and ignore them. The relationships in Figure 5.2 can be generalized for programs with unconstrained pointers as a concept. Consequently, we suggest starting from statements in a selected expanded dynamic slice rather than the whole program as candidates for constructing a corresponding critical slice.

5.1.2.2 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. The cost is obviously high. As suggested above, we can construct a critical slice by selecting statements in the corresponding expanded dynamic program slices instead of all executable statements. This is one step for reducing the cost. Furthermore, if the debugging process is integrated with a mutation–based testing tool, we can get critical slices at no additional cost because the concept of building critical slices is derived from killing sdl mutants that is usually one of the criteria first being satisfied (san and sdl). Thus, we can obtain critical slices automatically while killing sdl mutants in the testing phase with minor instrumentation. In this case, the cost of constructing critical slices is not a concern at all.

Moreover, the research to perform program mutation in an efficient manner has been conducted. [Cho90, Kra91] With that support, the critical slices can be efficiently constructed during mutation–based testing.

5.1.2.3 Effectiveness

To study the effectiveness of Critical Slicing (CS), we are interested in the reduction rate among the size of critical slices, corresponding expanded dynamic program 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 expanded dynamic program slices (i.e., actually or potentially affecting the results) are examined for studying the limitation of CS.

From the relationships among CS, DPS, and EDPS presented in Section 5.1.2.1, two categories are classified for the statements not in a critical slice but in the corresponding expanded dynamic program slice. The first one is the assignment statement having no effect on the defined variable but actually or potentially affecting the results. Example 5.1 illustrates this case (i.e., Statement 2 in Figure 5.1). The other category is that the propagation of the effect of an executed statement (including path selections and reach definitions — data and control flows) is the same as the propagation without executing the statement. In this case the statement will not be in the corresponding critical slice because of the same result.

For instance, Statement 4 in Figure 5.1 will not be in the critical slice if the input for a is 1. The effect of Statement 4 decides the path selection at Statement 5. For input data 1, the value of a becomes -9 after Statement 4 is executed, and is still 1 at line 5 if Statement 4 is not executed. No matter if Statement 4 is executed or not, the evaluation of Statement 5 is always false which makes the propagation with or without the effect of Statement 4 to be the same. Thus Statement 4 is not critical to the result of incorrect output value 5.

If faulty statements belong to the above two categories, they will not be indicated by the corresponding critical slices. Then, EDPS is needed for fault localization.

We conducted an experiment to evaluate the frequency of faulty statements being covered by critical slicing as well as the reduction rate between the size of critical slices and corresponding expanded dynamic program slices. Results of effectiveness are reported in Chapter 6.2.4.

5.2 Debugging with Other Mutation–Based Testing Information

In this section, we explore information obtained from mutation–based testing along with dynamic instrumentation for debugging purposes. Methods for detailed analysis are first described in the next subsection. Then possible hints deduced from the analysis are illustrated. For clarity and simplicity, we assume there is only one faulty statement (S_f) in a given faulty program (i.e., single fault assumption). However, these hints can still be applied to the case of multiple faults.

We did not conduct experiments for heuristics and hints presented in this section because of the limitation of existing prototype tools. The current mutation–based testing tool MOTHRA is for programs in FORTRAN 77 by the interpretation approach, but SPYDER is for ANSI C based on GNU GCC and GDB. We consider the heuristics and hints in this section as a guide to provide primitive suggestions and directions for further fault localization.

5.2.1 Methods for Analyzing the Information

As mentioned at the beginning of this chapter, we have classified mutant types into five groups from the program dependency (control and data flow) analysis standpoint. We propose a method to thoroughly examine information obtained from mutation–based testing according to the regrouped mutant types, the behavior of a mutant and the original program, and dynamic program slices between a mutant and the original program.

The method is presented as a 5–tuple $\langle \mathcal{MT}, \mathcal{C}, \mathcal{P}, \mathcal{M}, \mathcal{O} \rangle$. $\mathcal{MT}$ represents the five classified mutant groups.

$\mathcal{MT}1$: statement analysis.

$\mathcal{MT}2$: domain perturbation.

$\mathcal{MT}3$: operand (variable or constant) replacement on the use–part (right hand side) of a selected original statement.

$\mathcal{MT}4$: operand (variable) replacement on the define–part (left hand side) of a selected original statement.

$\mathcal{MT}5$: control dependency variation.

$\mathcal{C}$ represents five possible situations by comparing results of a selected variable in the original program P and a mutant M for a given test case and checkpoint.

$\mathcal{C}$ase A: both P and M have the same correct result. M is still alive.

$\mathcal{C}$ase B: P has a correct result but M has an incorrect one. M is killed.

$\mathcal{C}$ase C: both P and M have incorrect results but different. M is killed.

$\mathcal{C}$ase D: both P and M have the same incorrect result. M is still alive.

$\mathcal{C}$ase E: P has an incorrect result but M has a correct one. M is killed.

The given test case is constructed to kill M, i.e., the reachability and necessity conditions with regard to the mutated statement on line S of M (S_M) and the original statement on line S of P (S_P) are satisfied. Then, the information propagated from S_M to the checkpoint makes the above five different situations. M will always be killed in Cases B, C, and E which implies the sufficiency condition is also satisfied. Test cases with results in Cases A and B are non–error–revealing test cases, and those in Cases C, D, and E are error–revealing test cases. After constructing a test case t to kill M, we verify one output variable v at a time by classifying v into one of the above five cases based on its value in P and M.

For Case E, we always first compare the original statement S_P with the mutated statement S_M to verify whether S_M is the correct version of the faulty statement S_P. If the verification is true, we have located the faulty statement S_P and we fix it with the correct version S_M. Otherwise, the other cases are considered.

$\mathcal{P}$ and $\mathcal{M}$ represent the scope of selected dynamic slices of program P and mutant M, respectively. In order to do program dependency analysis and to cover the maximum scope of suspicious statements, dynamic slices of EDPS that are chosen as the basis of dynamic instrumentation for debugging are employed for the analysis in this chapter. $Dyn(P, v, l, t)$ thus represents an expanded dynamic program slice. For killing a mutant, the output results of the original program and a mutant are compared. The location parameter of suspicious dynamic slices is often set to the end of program execution, and $Dyn(P, v, l, t)$ is presented as $Dyn(P, v, \$, t)$.

Furthermore, we employ the *forward dynamic slice* [HRB90] to observe the propagation of the local effect after the execution of the original statement and the mutated statement on line S (S_P and S_M). In other words, the concept of forward dynamic slicing is to indicate the propagation of program state changes from the place of necessity condition to the checkpoint (usually the end of program execution), i.e., the information flow regarding the sufficiency condition. A forward dynamic slice, $FDyn(P, x, S, t)$, includes all statements actually and potentially affected by the variable x at line S for test case t where the actual and potential effect have the same semantics as those in exact dynamic slices (DPS) and potential effect slices (PES) mentioned in Chapter 3.3.

Different scopes of selected dynamic slices in $\mathcal{P}$ and $\mathcal{M}$ are listed as follows.

- $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ with respect to the selected variable v as mentioned above at location l (usually the end of execution) for test case t, respectively. These two dynamic slices are the basis for debugging.

- scope of $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ before line S where the statement is mutated.

- scope of $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ after line S.

- $Dyn(P, x, S, t)$ and $Dyn(M, x', S, t)$ where x' is the variable replacing x in P on line S for mutant M (i.e., x in S_P and x' in S_M). They represent dynamic slices regarding the satisfaction of reachability and necessity conditions of P and M, respectively.

- forward dynamic slices $FDyn(P, x, S, t)$ and $FDyn(M, x', S, t)$ where x' is the mutation variable of x in P on line S for mutant M.

If line S is inside a loop while considering the scope of $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ before/after line S, the first appearance of S is used. In the meantime, the last appearance of S is used in $Dyn(P, x, S, t)$ and $Dyn(M, x', S, t)$.

S_P and S_M are the code specified by mutation–based testing criteria. If they are not in the basis dynamic slices for debugging (i.e., $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$, respectively), then they do not contribute to the program failures. In this case, further dependency analysis on them could not provide helpful information for fault localization. Therefore, we always first examine whether S_P and S_M are in $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$, respectively.

Within the scope of $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ before S, the execution paths between P and M are identical because both programs are executed against a given test case and statements above line S of both programs are the same. The sufficiency condition is exposed in the scope of $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ after S. We thus examine the situation of different execution paths between P and M in the scope of $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ after S for further analysis.

$\mathcal{O}$ represents the logical operations for dynamic slices of $\mathcal{P}$ and $\mathcal{M}$. This includes intersection ($\mathcal{P} \cap \mathcal{M}$), union ($\mathcal{P} \cup \mathcal{M}$), and two difference operations ($\mathcal{P} - \mathcal{M}$ and

$\mathcal{M} - \mathcal{P}$). These four operations help us focus on the common or different parts between two selected dynamic slices and thus reduce the search domain.

By employing the proposed method $\langle \mathcal{MT}, \mathcal{C}, \mathcal{P}, \mathcal{M}, \mathcal{O} \rangle$, we suggest a possible region for further analysis. The possible results from the analysis are:

- N/A: not applicable;
- N/I: no information can be derived;
- Never: faulty statements will never appear in the region;
- Sometimes: faulty statements will sometimes appear in the region. In this case, we will evaluate the possibility of containing faults within the reduced region by labeling -1, 0, and 1 for the lowest, normal, and highest chance, respectively; and
- Always: faulty statements are always in the region. This is the most promising result.

The analysis table of this method is too large to be listed here, and only a few entries show promising results. We thus summarize a set of effective hints derived from thorough examination and illustrate them according to groups of $\mathcal{MT}$ in the following sections. Notation and terminology in Chapter 4 and Appendix A are reused here.

For each $\mathcal{MT}$ group, we first illustrate features of dynamic slices in $\mathcal{P}$ and $\mathcal{M}$ that are target regions to be examined. Then, detailed analysis is presented in the following hierarchical order: 1) whether S_P and S_M being in $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$, respectively; 2) the scope of $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ after line S; and 3) the five possible cases of $\mathcal{C}$. In the second level, the situation of different execution paths between P and M in $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ after S are analyzed.

5.2.2 Statement Analysis — $\mathcal{MT}1$

The result of statement coverage after killing san mutants is the same as the one of branch/block coverage testing methodologies. Only reachability is considered in the statement, block, or branch coverage, and no further information is provided. We thus focus on sdl mutants. In $\mathcal{MT}1$, the necessity condition is assumed to be satisfied when reachability is achieved. However, the execution flow of the sufficiency condition of sdl mutants has not been examined. The scope of $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ after S, which exposes the sufficiency condition, is therefore analyzed.

For dynamic slices regarding the original and mutated statements ($Dyn(P, x, S, t)$ and $Dyn(M, x', S, t)$) of sdl mutants, $Dyn(M, x', S, t)$ is undefined because of the deletion of S in M. $Dyn(P, x, S, t)$ is defined only if S_P on line S is an assignment statement and x is the defined variable. So is the case for forward dynamic slices $FDyn(P, x, S, t)$ and $FDyn(M, x', S, t)$.

Because S_P is deleted in M, our first consideration whether S_P and S_M are in $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$, respectively, has only two cases: $S_P \notin Dyn(P, v, \$, t)$ or $S_P \in Dyn(P, v, \$, t)$.

Case 1. $S_P \notin Dyn(P, v, \$, t)$

In this case, $Dyn(P, v, \$, t)$ equals $Dyn(M, v, \$, t)$, and we cannot obtain extra information from mutant M. Therefore, heuristics in Chapter 4 by varying variables with correct or incorrect values are employed to reduce the search domain.

Case 2. $S_P \in Dyn(P, v, \$, t)$

The effect of S_P is shown in both $\mathcal{P}3$ and $\mathcal{M}3$ which are examined for further analysis.

Case 2.1 The same execution path in the scope of $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ after line S.

This case implies that the effect without executing S_P in M does not change the evaluation of predicate statements that are dependent on S_P, if any. If S_P does not affect any predicate statement that affects the result, then the variable v in $Dyn(P, v, \$, t)$ is only affected by the data dependency from S_P. In this case, it is unlikely that P and M will have the same value on v (i.e., $\mathcal{C}$ases A and D), unless S_P is an assignment

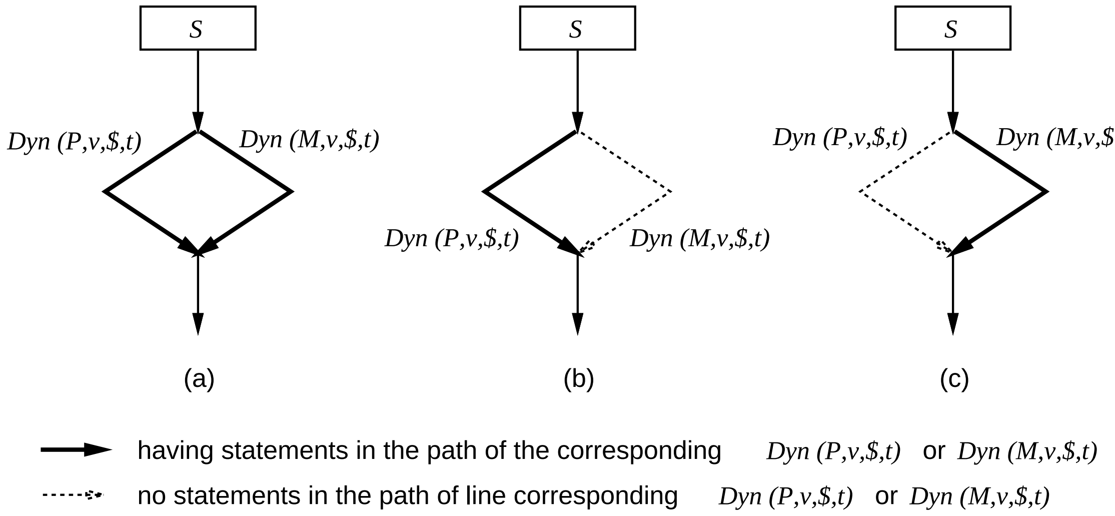

S
Dyn (P,v,$,t)
Dyn (M,v,$,t)
(a)
S
Dyn (P,v,$,t)
Dyn (M,v,$,t)
(b)
S
Dyn (P,v,$,t)
Dyn (M,v,$,t)
(c)
having statements in the path of the corresponding Dyn (P,v,$,t) or Dyn (M,v,$,t)
no statements in the path of line corresponding Dyn (P,v,$,t) or Dyn (M,v,$,t)

Figure 5.3 Examples of the different execution paths between the scope of $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ after line S. $Dyn(P, v, \$, t)$ goes through the right hand side path and $Dyn(M, v, \$, t)$ goes the left hand side path. Three possible cases for having statements in the different paths within the scope of $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ after S indicated in (a), (b), and (c).

statement with no effect on the defined variable. On the other hand, if S_P affects some predicate statements that affect the result, then the value of the defined variable in S_P after or before the execution of S_P will always choose the same path decided by the predicate statements. These two cases are as follows.

Case 2.1.1 S_P does not affecting any predicate statement that affects the result.

For $\mathcal{C}$ase A where both P and M have the same correct result, we suggest ignoring $\{Dyn(P, v, \$, t) - Dyn(M, v, \$, t)\}$ and focusing on other regions. In this category (Case 2.1.1), S_P affects v in $\mathcal{P}1$ via data dependency. If S_f is in the difference set, then it must be coincidentally correct letting P as well as M have the same correct result. We prefer to postpone the examination of the coincidental correctness after other regions being explored.

For $\mathcal{C}$ase D where both P and M have the same incorrect result, we suggest examining $\{Dyn(P, v, \$, t) \cap Dyn(M, v, \$, t)\}$ first. Statements involved in both incorrect execution have more chance to be faulty. The faulty statement is unlikely in $Dyn(P, x, S, t)$ because $Dyn(P, x, S, t)$ is related to S_P which is removed in M. If S_f is in $Dyn(P, x, S, t)$, there exists at least one instance of coincidental correctness letting M have the same incorrect result as P has.

For $\mathcal{C}$ase E where P has an incorrect result but M has a correct one, we suggest examining $\{Dyn(P, v, \$, t) - Dyn(M, v, \$, t)\}$ first, i.e., statements only involved in the execution of incorrect results. In $\mathcal{C}$ase E, S_f should be in $\{Dyn(P, v, \$, t)$ because of the incorrect result of P. If S_f is also in $Dyn(M, v, \$, t)$, then the effect of skipping S_P in M must adjust the effect of executing S_f to let M have a correct result. We defer the examination of this coincidental situation after statements leading to incorrect results are examined.

For $\mathcal{C}$ases B and C, no significant hints are concluded.

Case 2.1.2 S_P affects predicate statements that affect the result.

For $\mathcal{C}$ases A, D and E, we obtain the same hints as indicated in Case 2.1.1 mentioned above.

Case 2.2 Different execution paths between the scope of $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ after line S (e.g., the bold paths in Figure 5.3 — referred to as region R).

Line S either affects the decision of a predicate statement or is a predicate statement itself. Results of M are affected by the deletion of line S and/or the possible effect from the faulty statement S_f.

For $\mathcal{C}$ases A and D where both P and M have the same result, we suggest ignoring statements in the region R (i.e., statements in the bold paths of Figure 5.3). If S_f is in R, then there exists one instance of coincidental correctness for P and another one for M to let P and M have the same result.

For $\mathcal{C}$ase B where P has a correct result but M has an incorrect one, we cannot derive further hints because the cause of the incorrect result of M could just be the sdl effect.

5.2.3 Domain Perturbation — $\mathcal{MT}2$

The program dependency graphs of the original program and mutants in this category are the same because no new variables are introduced for mutants. The variable x of $Dyn(P, x, S, t)$ and $Dyn(M, x', S, t)$ is the variable being perturbed, if any. If the mutant operators are applied to a constant of the statement at line S, then the defined variables in S_P and S_M are used for $Dyn(P, x, S, t)$ and $Dyn(M, x', S, t)$, respectively. $Dyn(P, x, S, t)$ and $Dyn(M, x', S, t)$ do not exist, if there is no defined variable at line S. Generally speaking, if the existence of faulty statement S_f is manifested while satisfying the necessity condition, then $Dyn(P, x, S, t)$ (if existing) and statements affected by the variable x at S_P (i.e., $FDyn(P, x, S, t)$) should be examined first.

There are four possible cases to be considered whether S_P and S_M are in $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$, respectively.

Case 1. $S_P \notin Dyn(P, v, \$, t)$ and $S_M \notin Dyn(M, v, \$, t)$

In this case, $Dyn(P, v, \$, t)$ equals $Dyn(M, v, \$, t)$ which is the same as Case 1 of $\mathcal{MT}1$ in Section 5.2.2. We thus have the same conclusion — to employ heuristics in Chapter 4 for reducing the search domain.

Case 2. $S_P \notin Dyn(P, v, \$, t)$ but $S_M \in Dyn(M, v, \$, t)$

Case 3. $S_P \in Dyn(P, v, \$, t)$ but $S_M \notin Dyn(M, v, \$, t)$

If P is an array or pointer free program, these cases will not happen at all. There are two possible scenarios for the behavior of the statement at line S of P and M (S_P and S_M) that have no array or pointer reference. First, the statement does not affect any predicate statements. In this scenario, P and M will have the same execution path. The program dependency graphs of P and M are identical in this mutant category. Therefore, S_P and S_M will have the same effect on the result (i.e., $S_P \in Dyn(P, v, \$, t)$ if and only if $S_M \in Dyn(M, v, \$, t)$).

Second, the statement affects at least one predicate statement. In this scenario, the execution paths of P and M could be different. For expanded dynamic program slices, if the predicate statement has control dependency on the result, both actual and potential effect will be included in EDPS. Thus, we get the same conclusion as the one in the first scenario, “S_P is in $Dyn(P, v, \$, t)$ if and only if S_M is in $Dyn(M, v, \$, t)$.”

However, it is possible that S_P is in a DPS and S_M is not in the corresponding DPS and vice versa, especially for the second scenario — statements affecting at least one predicate statement.

For P with array or pointer reference, if mutant operators of this category ($\mathcal{MT}2$) are applied to the index of array/pointer reference in P, then the program dependency graphs of P and M can be different because the perturbation on the index of array/pointer reference may introduce new variables. In this case, the claim “$S_P \in Dyn(P, v, \$, t)$ if and only if $S_M \in Dyn(M, v, \$, t)$” are not always true. Example 5.3 illustrates a similar but not identical case.

Because these two cases (Cases 2 and 3) only happen for programs with array or pointer reference while the execution of S_M affects the index of arrays (or pointers), it is a difficult task to derive further information for debugging in this complicated circumstance.

Case 4. $S_P \in Dyn(P, v, \$, t)$ and $S_M \in Dyn(M, v, \$, t)$

Information flow from line S to the end of program execution is shown in the scope of $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ after line S, which indicate the propagation of the effect after S_P and S_M are executed, respectively. Therefore, we examine $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ after S for further analysis.

Case 4.1 The same execution path in the scope of $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ after line S.

The effect of executing S_M in M selects the same path in predicate statements, which are dependent on S_M, as S_P does in P. While analyzing whether line S affects any predicate statement that affects the result, we have the same generic analysis as Case 2.1 of $\mathcal{MT}1$ in Section 5.2.2 and examine the following two subcases.

Case 4.1.1 S_P does not affect any predicate statement that affects the result.

For $\mathcal{C}$ase A where both P and M have the same correct result, we suggest ignoring $Dyn(P, x, S, t) \cup \{FDyn(P, x, S, t) \cap$ ($Dyn(P, v, \$, t)$ after S)$\}$ which implies the faulty statement S_f is unlikely in the above region. In this category (Case 4.1.1), S_P affects v in $Dyn(P, v, \$, t)$ via data dependency. If S_f is in the above region, then it must be coincidentally correct letting P and M have the same correct result. We prefer to postpone examination of coincidental correctness until other regions are explored.

For $\mathcal{C}$ases C, D and E where P has an incorrect result, we suggest examining $Dyn(P, x, S, t) \cup \{FDyn(P, x, S, t)$ in the scope of $Dyn(P, v, \$, t)$ after $S\}$ because the faulty statement is triggered while M is killed. The necessity condition is satisfied via $Dyn(P, x, S, t)$, and the sufficiency condition is propagated via $FDyn(P, x, S, t)$ in the scope of $Dyn(P, v, \$, t)$ after S. More specifically, the necessity condition is not significant enough to distinguish P and M in $\mathcal{C}$ase D, and the effect of both the necessity condition and the faulty statement makes M have the correct result in $\mathcal{C}$ase E.

For $\mathcal{C}$ase B, we do not know that the incorrect result of M is caused by the necessity condition, the faulty statement, or both. No further hints can be derived from this complicated unknown situation.

Case 4.1.2 S_P affects predicate statements that affect the result.

We accomplish the same hints of Case 4.1.1 mentioned above.

Case 4.2 Different execution paths between the scope of $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ after line S (e.g., the bold paths in Figure 5.3 — referred to as region R).

For $\mathcal{C}$ase A where both P and M have the same correct result, we would suggest ignoring statements in the union of $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ after line S but not in the intersection of $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ (i.e., the scope of

$\{(Dyn(P, v, \$, t) \cup Dyn(M, v, \$, t)) - (Dyn(P, v, \$, t) \cap Dyn(M, v, \$, t))\}$ after line S). Otherwise, an instance of coincidental correctness must exist to let both P and M generate the same correct result if the faulty statement is in the above region.

For $\mathcal{C}$ase C where P and M have different incorrect results, the faulty statement could be in $\{(Dyn(P, x, S, t)\cap$ ($Dyn(P, v, \$, t)$ before S)) $\cup$ ($FDyn(P, x, S, t)\cap$ ($Dyn(P, v, \$, t) \cap Dyn(M, v, \$, t)$ after S)$\}$. The faulty statement is triggered by either the flow to satisfy the necessity condition (the former subset) or the flow propagated for the sufficiency condition (the latter subset).

For $\mathcal{C}$ase D where both P and M have the same incorrect result, we suggest examining $\{Dyn(P, v, \$, t)\cap Dyn(M, v, \$, t)\}$ because of the same incorrect result that is caused by the faulty statement S_f under the single faulty statement assumption. In this case, the region R will be ignored.

For $\mathcal{C}$ase E where P has an incorrect result but M has a correct one, we suggest examining $\{Dyn(P, v, \$, t) - Dyn(M, v, \$, t)\}$ and ignoring $\{(Dyn(M, v, \$, t) - Dyn(P, v, \$, t))$ before line $S\}$ because M could skip the execution of the faulty statement and generate a correct result.

For $\mathcal{C}$ase B, no further hints can be derived for the same reason mentioned in Case 4.1.1.

5.2.4 Operand Replacement on the Use–part of a Statement — $\mathcal{MT}3$

In this category, P and M have the same program dependency graph below the statement at line S where mutant operators are applied. The necessity condition lets the value of the operand in S_P not equal the value of the replaced operand in S_M. $Dyn(P, x, S, t)$ and $Dyn(M, x', S, t)$, if defined, are different because of the operand replacement.

If the faulty statement S_f is triggered and causes program failures while satisfying the necessity condition, then statements leading to the satisfaction of the necessity condition as well as statements being affected by the execution of S_P and S_M should be examined. Statements of the former case are in $\{Dyn(P, x, S, t) \cup Dyn(M, x', S, t)\}$, and those of the latter one are in $\{FDyn(P, x, S, t) \cup FDyn(M, x', S, t)\}$. These two sets will be often referred to in the following analysis.

Case 1. $S_P \notin Dyn(P, v, \$, t)$ and $S_M \notin Dyn(M, v, \$, t)$
Case 2. $S_P \notin Dyn(P, v, \$, t)$ but $S_M \in Dyn(M, v, \$, t)$
Case 3. $S_P \in Dyn(P, v, \$, t)$ but $S_M \notin Dyn(M, v, \$, t)$

The same hints for Cases 1, 2, and 3 of $\mathcal{MT}2$ in Section 5.2.3 are derived for $\mathcal{MT}3$ and are summarized as follows. For Case 1, $Dyn(P, v, \$, t)$ is equal to $Dyn(M, v, \$, t)$. Cases 2 and 3 only happen for programs with array or pointer reference under certain circumstances because P and M have the same program dependency graph after the statement at line S. In other words, "$S_P \in Dyn(P, v, \$, t)$ if and only if $S_M \in Dyn(M, v, \$, t)$" is true for programs without array or pointer reference. Although, it is possible that S_P is in a DPS and S_M is not in the corresponding DPS and vice versa.

For these three cases, heuristics in Chapter 4 are employed to reduce the search domain.

Case 4. $S_P \in Dyn(P, v, \$, t)$ and $S_M \in Dyn(M, v, \$, t)$

Statements in the scope of $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ after line S are examined because they indicate the propagation of the effect after the execution of S_P and S_M.

Case 4.1 The same execution path in the scope of $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ after line S.

This case implies that the effect of executing S_M in M selects the same path in predicate statements, which are dependent on S_M, as S_P does in P. From the analysis in Case 2.1 of $\mathcal{MT}1$ in Section 5.2.2 and Case 4.1 of $\mathcal{MT}2$ in Section 5.2.3, we investigate two subcases (Case 4.1.1 and 4.1.2) for whether S_P affects any predicate statement that affects the result, and derive the same hints for both subcases as follows.

For $\mathcal{C}$ase A where both P and M have the same correct result, we suggest ignoring statements in $\{(Dyn(P, x, S, t) \cup Dyn(M, x', S, t))$ in the scope of $Dyn(M, v, \$, t)$ before $S\}$ as well as $\{(FDyn(P, x, S, t) \cup FDyn(M, x', S, t))$ in the scope of

$Dyn(M, v, \$, t)$ after $S\}$. If the faulty statement S_f is in the dynamic slices associated with M, then either the necessity condition of S_M offsets the effect of S_f to let M have the same correct result as P or both the effect of the necessity condition and S_f do not make an incorrect result simultaneously. We defer the examination of this unusual situation, and thus assume that S_f is unlikely in $Dyn(M, v, \$, t)$ by ignoring the statements leading to and being affected by the necessity condition in M.

For $\mathcal{C}$ase B where P has a correct result but M has an incorrect one, there are two possible regions for investigation: $\{(Dyn(P, x, S, t) \cup Dyn(M, x', S, t))$ in the scope of $Dyn(M, v, \$, t)$ before $S\} \cup \{(FDyn(P, x, S, t) \cup FDyn(M, x', S, t))$ in the scope of $Dyn(M, v, \$, t)$ after $S\}$ and $\{Dyn(M, v, \$, t) - Dyn(P, v, \$, t)\}$. The first one is for S_f in $Dyn(P, v, \$, t)$ with an instance of coincidental correctness that is broken by the necessity condition in M. The second one is for S_f not in $Dyn(P, v, \$, t)$ but triggered in $Dyn(M, v, \$, t)$ to generate an incorrect result of M. However, if the incorrect result of M is only caused by the necessity condition which implies $S_f \notin Dyn(M, v, \$, t)$, then the second region is not effective.

For $\mathcal{C}$ase C where P and M have different incorrect results, we suggest examining $\{(Dyn(P, x, S, t) \cup Dyn(M, x', S, t))$ in the scope of $Dyn(M, v, \$, t)$ before $S\} \cup \{(FDyn(P, x, S, t) \cup FDyn(M, x', S, t))$ in the scope of $Dyn(M, v, \$, t)$ after $S\}$. The faulty statement S_f is triggered when the necessity condition is satisfied and/or the effect of executing S_M is propagated to the end.

For $\mathcal{C}$ase D where both P and M have the same incorrect result, we suggest ignoring statements in $\{(Dyn(P, x, S, t) \cup Dyn(M, x', S, t))$ in the scope of $Dyn(M, v, \$, t)$ before $S\}$ as well as $\{(FDyn(P, x, S, t) \cup FDyn(M, x', S, t))$ in the scope of $Dyn(M, v, \$, t)$ after $S\}$. For $S_f \notin Dyn(M, v, \$, t)$, the effect of the necessity condition in M must generate the same incorrect result as P. That is an unusual situation. We thus ignore the statements indicated above. On the other hand, if $S_f \in Dyn(M, v, \$, t)$, then the same incorrect result of M is caused by S_f or the combination of the necessity effect and S_f. Because S_f is in $Dyn(P, v, \$, t)$ for $\mathcal{C}$ase D, the region $\{Dyn(P, v, \$, t) \cap Dyn(M, v, \$, t)\}$ or the region mentioned in $\mathcal{C}$ase C is worth being examined as the second choice.

For $\mathcal{C}$ase E where P has an incorrect result but M has a correct one, we suggest examining the region mentioned in $\mathcal{C}$ase C and ignoring the region mentioned in $\mathcal{C}$ase A. This combines the arguments in $\mathcal{C}$ase C for an incorrect result of P and the one in $\mathcal{C}$ase A for a correct result of M.

Case 4.2 Different execution paths between the scope of $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ after line S (e.g., the bold paths in Figure 5.3 — referred to as region R).

For $\mathcal{C}$ase A where both P and M have the same correct result, it is likely that the faulty statement S_f is not in the different execution paths P and M, i.e., the region R. Because P and M have different execution paths, if the faulty statement is in the above region, then there exist two different occurrences of coincidental correctness in P and M to generating the same correct result at the same time. We postpone the examination of this unusual situation by ignoring the region R.

For $\mathcal{C}$ase B where P has a correct result and M has an incorrect one, there are two possible regions for investigation: $\{(Dyn(P, x, S, t) \cup Dyn(M, x', S, t))$ in the scope of $Dyn(M, v, \$, t)$ before $S\} \cup \{(FDyn(P, x, S, t) \cup FDyn(M, x', S, t))$ in the scope of $Dyn(M, v, \$, t)$ after $S\}$ and $\{Dyn(M, v, \$, t) - Dyn(P, v, \$, t)\}$. The first one is for S_f in $Dyn(P, v, \$, t)$ with an instance of coincidental correctness that is broken by the necessity condition in M. At the same time, we can ignore statements in $\{R \cap Dyn(M, v, \$, t)\}$ because of the different execution paths between the scope of $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ after line S. The second one is for S_f not in $Dyn(P, v, \$, t)$ but triggered in $Dyn(M, v, \$, t)$ to generate the incorrect result of M. However, if the incorrect result of M is only caused by the necessity condition which implies $S_f \notin Dyn(M, v, \$, t)$, then the second region is not effective.

For $\mathcal{C}$ase C where P and M have different incorrect results, we cannot conclude further hints except the region suggested at the beginning of this category $\mathcal{MT}3$,

$\{(Dyn(P, x, S, t) \cup Dyn(M, x', S, t))$ in the scope of $Dyn(M, v, \$, t)$ before $S\} \cup \{(FDyn(P, x, S, t) \cup FDyn(M, x', S, t))$ in the scope of $Dyn(M, v, \$, t)$ after $S\}$.

For $\mathcal{C}$ase D where both P and M have the same incorrect result, we suggest first examining $\{Dyn(P, v, \$, t) \cap Dyn(M, v, \$, t)\}$ and ignoring R because the same incorrect results of P and M could be generated by the execution of the faulty statement S_f in both P and M. If S_f is in R that implies S_f in $\{Dyn(M, v, \$, t) \cap R\}$ for $\mathcal{C}$ase D, then the effect of the necessity condition in M must generate the same incorrect result as P. That is an unusual situation. We thus ignore statements in R. For the same reason, while considering the case $S_f \notin \mathcal{M}1$, we could ignore the statements in $\{(Dyn(P, x, S, t) \cup Dyn(M, x', S, t))$ in the scope of $Dyn(M, v, \$, t)$ before $S\}$ as well as $\{(FDyn(P, x, S, t) \cup FDyn(M, x', S, t))$ in the scope of $Dyn(M, v, \$, t)$ after $S\}$.

For $\mathcal{C}$ase E where P has an incorrect result but M has a correct one, we suggest examining the region mentioned in $\mathcal{C}$ase C and ignoring the region R as indicated in $\mathcal{C}$ase A. This combines the arguments in $\mathcal{C}$ase C for an incorrect result of P and the one in $\mathcal{C}$ase A for a correct result of M. Moreover, if the faulty statement S_f is in both $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$, then either the necessity condition offsets the effect of S_f to let M have the correct result or both the effect of the necessity condition and S_f do not contribute to the correct result of M. The examination of this complicated unusual situation is deferred. We thus can ignore statements in $\{Dyn(P, v, \$, t) \cap Dyn(M, v, \$, t)\}$ at the beginning and re-examine them later if needed.

5.2.5 Operand Replacement on the Define–part of a Statement — $\mathcal{MT}4$

In this category, P and M have the same program dependency graph above the statement at line S where mutant operators are applied. $Dyn(P, x, S, t)$ and $Dyn(M, x', S, t)$ are the same because the right hand side (use–part) of line S is not changed.

Let S'_P be the last statement which defines x' before line S (i.e., the reaching definition of x' at line S) and S'_M be the last statement which defines x before line S (i.e., the reaching definition of x at line S). Assume the use–part in line S is $g(\vec{y})$, i.e., $x = g(\vec{y})$ in S_P and $x' = g(\vec{y})$ in S_M.

$$\begin{array}{lllll}
 & P & \qquad\qquad & & M \\
 & \vdots & & & \vdots \\
S'_M\colon & x = \ldots\,; & & S'_M\colon & x = \ldots\,; \\
S'_P\colon & x' = \ldots\,; & & S'_P\colon & x' = \ldots\,; \\
 & \vdots & & & \vdots \\
S_P\colon & x = g(\vec{y})\,; & & S_M\colon & x' = g(\vec{y})\,; \\
 & \vdots & & & \vdots
\end{array}$$

There are four different types of the necessity condition while x and x' are compared before and after the execution of S_P and S_M at line S. The least restricted one ($N0$) does not make any comparison. As long as the reachability of the statement is satisfied, we assume the necessity is also achieved. The second one ($N1$) requires that the value of $g(\vec{y})$ does not equal the value of x' before S is executed, i.e., $g(\vec{y})$ of $S \neq x'$ of S'_P. But the value of $g(\vec{y})$ could equal the value of x before the statement at line S is executed. The third one ($N2$) requires that the value of $g(\vec{y})$ does not equal the value of x before S is executed, i.e., $g(\vec{y})$ of $S \neq x$ of S'_M. But the value of $g(\vec{y})$ could equal the value of x' before the statement at line S is executed. The most restricted one ($N3$) is the combination of $N1$ and $N2$, i.e., $g(\vec{y})$ of $S \neq x'$ of S'_P and $g(\vec{y})$ of $S \neq x$ of S'_M. In this study, we chose the most restricted one ($N3$) as the necessity condition to ensure that different program states exist between P and M right after the execution of S_P and S_M.

In the scope of $N3$, not only variables x and x' of line S but also x' of S'_P as well as x of S'_M are involved to satisfying the necessity condition. In addition to $Dyn(P, x, S_P, t)$ and $Dyn(M, x', S_M, t)$ that are the same as mentioned above, we introduce $Dyn(P, x', S'_P, t)$ and $Dyn(M, x, S'_M, t)$ for later usage. For the same reason, we introduce two new forward dynamic slices $FDyn(P, x', S'_P, t)$ and $FDyn(M, x, S'_M, t)$ associated with S'_P and S'_M, respectively. If the faulty statement S_f is triggered and causes program failures while satisfying the necessity condition, then statements leading to the satisfaction of the necessity condition, $\{Dyn(P, x, S_P, t) \cup Dyn(P, x', S'_P, t) \cup Dyn(M, x, S'_M, t)\}$ as well as statements being affected by the execution of S_P and S_M, $\{FDyn(P, x, S, t) \cup FDyn(M, x', S, t) \cup FDyn(P, x', S'_P, t) \cup FDyn(M, x, S'_M, t)\}$, should be examined.

Case 1. $S_P \notin Dyn(P, v, \$, t)$ and $S_M \notin Dyn(M, v, \$, t)$

Because S_P and S_M do not affect the results of P and M, $Dyn(P, v, \$, t)$ equals $Dyn(M, v, \$, t)$. Only $\mathcal{C}$ases A and D where both P and M have the same result will occur in this case, and we cannot derive further hints. Thus heuristics in Chapter 4 are needed to reduce the search domain.

Case 2. $S_P \notin Dyn(P, v, \$, t)$ but $S_M \in Dyn(M, v, \$, t)$
Case 3. $S_P \in Dyn(P, v, \$, t)$ but $S_M \notin Dyn(M, v, \$, t)$
Case 4. $S_P \in Dyn(P, v, \$, t)$ but $S_M \in Dyn(M, v, \$, t)$

For these three cases, at least one of the original or the mutated statement affects the results of P or M, respectively. Except for the forward dynamic slices, the same analysis is applied to these three cases. $FDyn(P, x, S, t)$ in Case 2 ($S_P \notin Dyn(P, v, \$, t)$) and $FDyn(M, x', S, t)$ in Case 3 ($S_M \notin Dyn(M, v, \$, t)$) should be ignored because the corresponding statements does not affect the results. Detailed analysis for these three cases is the same as the one mentioned in Case 4 for $\mathcal{MT}3$ in Section 5.2.4. Therefore, we only present the derived hints as follows.

Case 4.1 The same execution path in $\mathcal{P}3$ and $\mathcal{M}3$.

> For $\mathcal{C}$ase A where both P and M have the same correct result, we suggest ignoring statements in $\{(Dyn(P, x, S_P, t) \cup Dyn(P, x', S'_P, t) \cup Dyn(M, x, S'_M, t))$ in the scope of $Dyn(M, v, \$, t)$ before $S\}$ as well as $\{(FDyn(P, x, S, t) \cup FDyn(M, x', S, t) \cup FDyn(P, x', S'_P, t) \cup FDyn(M, x, S'_M, t))$ in the scope of $Dyn(M, v, \$, t)$ after $S\}$.

For $\mathcal{C}$ase B where P has a correct result but M has an incorrect one, there are two possible regions for investigation: $\{(Dyn(P,x,S_P,t) \cup Dyn(P,x',S'_P,t) \cup Dyn(M,x,S'_M,t))$ in the scope of $Dyn(P,v,\$,t)$ before $S\} \bigcup \{(FDyn(P,x,S,t) \cup FDyn(M,x',S,t) \cup FDyn(P,x',S'_P,t) \cup FDyn(M,x,S'_M,t))$ in the scope of $Dyn(P,v,\$,t)$ after $S\}$ and $\{Dyn(M,v,\$,t) - Dyn(P,v,\$,t)\}$.

For $\mathcal{C}$ase C where P and M have different incorrect results, we suggest examining $\{(Dyn(P,x,S_P,t) \cup Dyn(P,x',S'_P,t) \cup Dyn(M,x,S'_M,t))$ in the scope of $Dyn(P,v,\$,t)$ before $S\}$ $\bigcup$ $\{(FDyn(P,x,S,t) \cup FDyn(M,x',S,t) \cup FDyn(P,x',S'_P,t) \cup FDyn(M,x,S'_M,t))$ in the scope of $Dyn(P,v,\$,t)$ after $S\}$.

For $\mathcal{C}$ase D where both P and M have the same incorrect result, we suggest ignoring statements in $\{(Dyn(P,x,S_P,t) \cup Dyn(P,x',S'_P,t) \cup Dyn(M,x,S'_M,t))$ in the scope of $Dyn(M,v,\$,t)$ before $S\}$ as well as $\{(FDyn(P,x,S,t) \cup FDyn(M,x',S,t) \cup FDyn(P,x',S'_P,t) \cup FDyn(M,x,S'_M,t))$ in the scope of $Dyn(M,v,\$,t)$ after $S\}$. Moreover, the region $\{Dyn(P,v,\$,t) \cap Dyn(M,v,\$,t)\}$ or the region mentioned in $\mathcal{C}$ase C is worth being examined as the second try.

For $\mathcal{C}$ase E where P has an incorrect result but M has a correct one, we suggest examining the region mentioned in $\mathcal{C}$ase C and ignoring the region mentioned in $\mathcal{C}$ase A.

Case 4.2 Different execution paths between the scope of $Dyn(P,v,\$,t)$ and $Dyn(M,v,\$,t)$ after line S (e.g., the bold paths in Figure 5.3 — referred to as region R).

For $\mathcal{C}$ase A where both P and M have the same correct result, it is likely that the faulty statement S_f is not in the different execution paths of P and M, i.e., the region R. We thus ignore statements in R.

For $\mathcal{C}$ase B where P has a correct result but M has an incorrect one, there are two possible regions for investigation: $\{(Dyn(P,x,S_P,t) \cup Dyn(P,x',S'_P,t) \cup Dyn(M,x,S'_M,t))$ in the scope of $Dyn(P,v,\$,t)$ before $S\} \bigcup \{(FDyn(P,x,S,t) \cup FDyn(M,x',S,t) \cup FDyn(P,x',S'_P,t) \cup FDyn(M,x,S'_M,t))$ in the scope of $Dyn(P,v,\$,t)$ after $S\}$ and $\{Dyn(M,v,\$,t) - Dyn(P,v,\$,t)\}$.

For $\mathcal{C}$ase C where P and M have different incorrect results, we cannot conclude further hints except the region suggested at the beginning of this category $\mathcal{MT}4$, $\{(Dyn(P, x, S_P, t) \cup Dyn(P, x', S'_P, t) \cup Dyn(M, x, S'_M, t))$ in the scope of $Dyn(P, v, \$, t)$ before $S\}$ $\bigcup$ $\{(FDyn(P, x, S, t) \cup FDyn(M, x', S, t) \cup FDyn(P, x', S'_P, t) \cup FDyn(M, x, S'_M, t))$ in the scope of $Dyn(P, v, \$, t)$ after $S\}$.

For $\mathcal{C}$ase D where both P and M have the same incorrect result, we suggest first examining $\{Dyn(P, v, \$, t) \cap Dyn(M, v, \$, t)\}$ and ignoring R. While considering the case $S_f \notin Dyn(M, v, \$, t)$, we could ignore the statements in $\{(Dyn(P, x, S_P, t) \cup Dyn(P, x', S'_P, t) \cup Dyn(M, x, S'_M, t))$ in the scope of $Dyn(M, v, \$, t)$ before $S\}$ as well as $\{(FDyn(P, x, S, t) \cup FDyn(M, x', S, t) \cup FDyn(P, x', S'_P, t) \cup FDyn(M, x, S'_M, t))$ in the scope of $Dyn(M, v, \$, t)$ after $S\}$.

For $\mathcal{C}$ase E where P has an incorrect result but M has a correct one, we suggest examining the region mentioned in $\mathcal{C}$ase C and ignoring the region R as indicated in $\mathcal{C}$ase A. Statements in $\{Dyn(P, v, \$, t) \cap Dyn(M, v, \$, t)\}$ can be ignored at the beginning and re-examined later if needed.

5.2.6 Control Dependency Variation — $\mathcal{MT}5$

The glr and rsr mutants will radically change the transformation of execution flow and cause mutants to be easily killed without manifesting the existence of faults. This extreme situation is unlikely to provide useful information for debugging purposes. We therefore focus on the control dependency variation made by predicate statements. In program mutation, all hidden paths [DLS78] implicit in a compound predicate will be tested. The compound predicate is unfolded into a series of simple predicates that all paths are exercised in mutation–based testing. For simplicity, we assume that all compound predicates have been unfolded. The predicate statement in our analysis is a simple logical expression at line S. S_P is the original predicate statement of P, and S_M is the mutation predicate statement of M.

To satisfy the necessity condition of a given predicate statement, the values of all variables in S_P and S_M are considered to differentiate the evaluation results of S_P and S_M. Thus, $Dyn(P, x, S, t)$ and $Dyn(M, x', S, t)$ will be the union of dynamic slices with respect to all variables in S_P and S_M, respectively. So are the forward dynamic slices $FDyn(P, x, S, t)$ and $FDyn(M, x', S, t)$. The statements leading to the satisfaction of the necessity condition are in $\{Dyn(P, x, S, t) \cup Dyn(M, x', S, t)\}$, and those being affected by the execution of S_P and S_M are in $\{FDyn(P, x, S, t) \cup FDyn(M, x', S, t)\}$. If faulty statement S_f is triggered and causes program failures when the necessity condition is satisfied, then these statements mentioned above should be examined. In consequence of the satisfaction of the necessity condition, P and M will execute different statement blocks decided by the predicate at line S. That implies there exist different execution paths between the scope of $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ after line S (e.g., the bold paths in Figure 5.3 — referred to as region R).

Case 1. $S_P \notin Dyn(P, v, \$, t)$ and $S_M \notin Dyn(M, v, \$, t)$

In this case, $Dyn(P, v, \$, t)$ equals $Dyn(M, v, \$, t)$ that is the same as Case 1 of $\mathcal{MT}1$ in Section 5.2.2. We thus have the same conclusion — to employ heuristics in Chapter 4 for reducing the search domain.

Case 2. $S_P \notin Dyn(P, v, \$, t)$ but $S_M \in Dyn(M, v, \$, t)$
Case 3. $S_P \in Dyn(P, v, \$, t)$ but $S_M \notin Dyn(M, v, \$, t)$

These two cases happen only when using Exact Dynamic Program Slicing (EDPS) to construct $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$. As for EDPS, the predicate statement at line S will be included for both the potential and actual effect, and only Cases 1 and 4 are considered. The analysis for Cases 2 and 3 based on EDPS is similar to the one of Case 4 as follows.

Case 4. $S_P \in Dyn(P, v, \$, t)$ and $S_M \in Dyn(M, v, \$, t)$

While examining the change of program dependency graph between P and M, we classify the mutation predicate statements into two scenarios: 1) no variable replacement involved in S_P and S_M; and 2) replacement for the variable operand of the predicate statement. The first scenario includes the logical connector replacement (lcr), the relational operator replacement (ror), and the constant replacement for the constant operand of a predicate statement (crp and src, if any).

For the first scenario, the program dependency graphs of P and M are the same. Because there exist different execution paths between the scope of $Dyn(P, v, \$, t)$ and $Dyn(M, v, \$, t)$ after line S as mentioned above, this circumstance is the same as the Case 4.2 of $\mathcal{MT}2$ in Section 5.2.3. For the second scenario, the program dependency graphs of P and M have the same program dependency graph below the statement at line S. This circumstance is the same as the Case 4.2 of $\mathcal{MT}3$ in Section 5.2.4. Results for both circumstances are not repeated here.

In this category $\mathcal{MT}5$, if a faulty predicate statement is a compound one and has a wrong logical connector or relational operator, then two kinds of error–indicating mutants will help us locate the fault. The first one is the mutant having the correct version of the faulty predicate (i.e., the connector replacement of lcr or the relational operator replacement of ror). The second one is the mutant with the LEFTOP or RIGHTOP of lcr as well as the FALSEOP or TRUEOP of ror, where LEFTOP returns the left operand (the right is ignored); RIGHTOP returns the right operand; FALSEOP always returns FALSE; and TRUEOP always returns TRUE. When trying to kill mutants in the second case, we are forced to construct test cases for exercising certain hidden paths in order to satisfy the necessity condition, and these test cases likely cause program failures because of different execution paths decided by the faulty predicate.

5.3 Summary

An effective approach to identify statements directly affecting the program failure, Critical Slicing, has been developed in this chapter. It is derived from the statement deletion mutant operator of mutation–based testing. An experiment to confirm the effectiveness of critical slicing is presented in Chapter 6.2.4. Also, a set of suggestions that are deduced from the information of mutation–based testing has been presented. We believe the suggestions provide valuable hints to help users focus on the right place for further analysis. By combining the approach proposed in this chapter with the heuristics in the previous chapter, we can further reduce the search domain for fault localization.

6. A PROTOTYPE IMPLEMENTATION AND EXPERIMENTATION

This chapter gives an overview of the prototype debugging tool SPYDER to demonstrate the feasibility of applying the proposed heuristics on fault localization. A set of preliminary experiment results is then presented to confirm the effectiveness of our approach by providing examples.

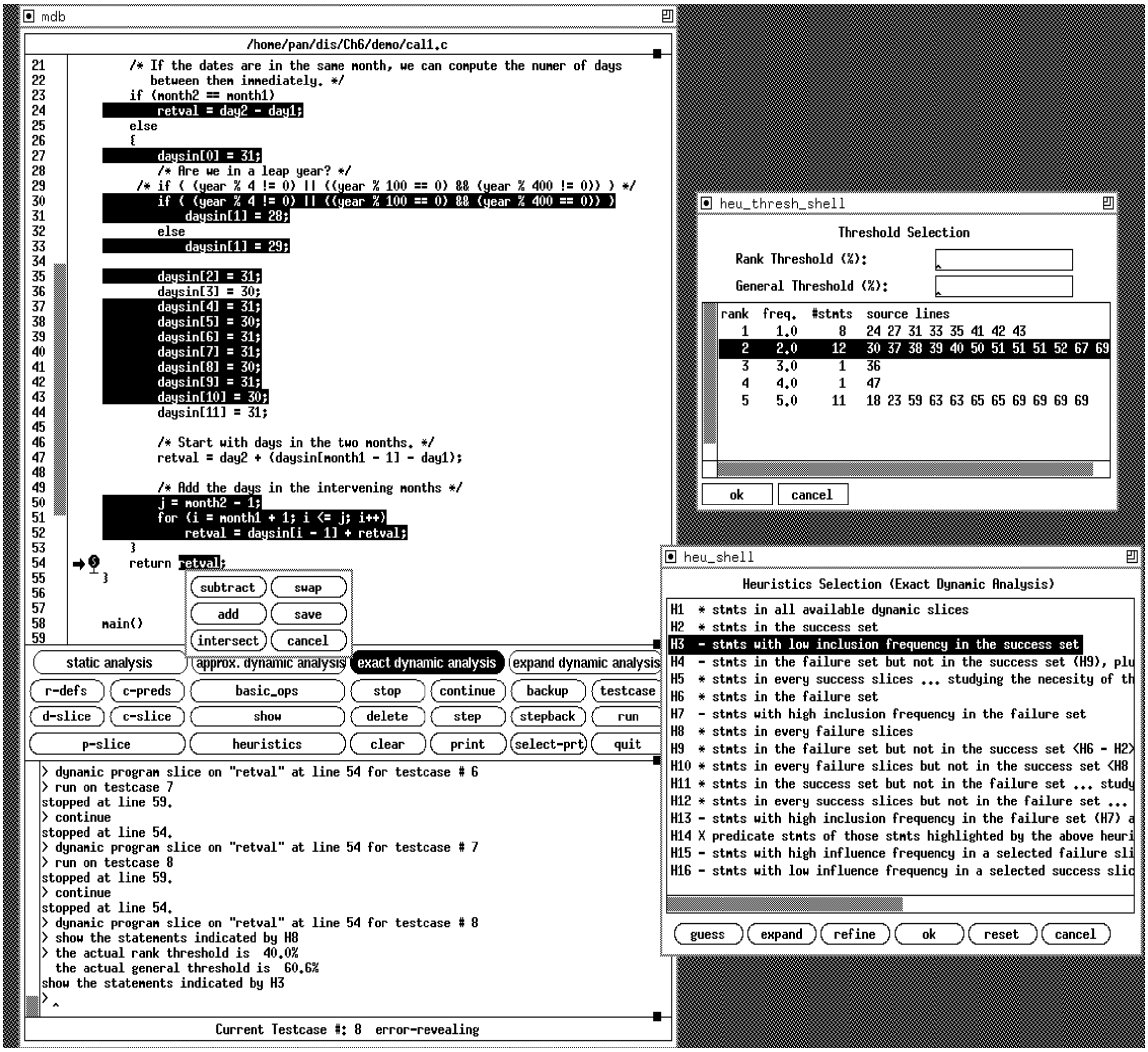
The focus of the implementation overview is on the integration of our approach into SPYDER. A brief summary of the software components and the user interface of SPYDER is illustrated without details.

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 is evaluated by the accuracy and the size of the reduced search domain. The information provided by the first factor decides whether the reduced domain still contains faulty statements without misleading users. At the same time, the second factor demonstrates the effort which could be saved in terms of the number of the suspicious statements to be examined. We therefore develop two comparison methods to evaluate the effectiveness of the proposed heuristics of Chapter 4 and the Critical Slicing of Chapter 5.1.

6.1 SPYDER: A Prototype Debugger

In order to support the new debugging paradigm proposed in Figure 1.1 of Chapter 1.2, a prototype debugging tool, SPYDER [ADS91a, ADS91b, AS88, Agr91], has been constructed to perform the slicing and backtracking functions. Features of SPYDER are briefly summarized here. Readers are referred to [Agr91] for details of the implementation and functions of SPYDER.

Heuristics in Chapter 4 and the Expanded Dynamic Program Slicing in Chapter 3.3 are integrated into SPYDER to enhance the capability of fault localization in the proposed



```
21          /* If the dates are in the same month, we can compute the numer of days
22             between them immediately. */
23          if (month2 == month1)
24              retval = day2 - day1;
25          else
26          {
27              daysin[0] = 31;
28              /* Are we in a leap year? */
29           /* if ( (year % 4 != 0) || ((year % 100 == 0) && (year % 400 != 0)) ) */
30              if ( (year % 4 != 0) || ((year % 100 == 0) && (year % 400 == 0)) )
31                  daysin[1] = 28;
32              else
33                  daysin[1] = 29;
34
35              daysin[2] = 31;
36              daysin[3] = 30;
37              daysin[4] = 31;
38              daysin[5] = 30;
39              daysin[6] = 31;
40              daysin[7] = 31;
41              daysin[8] = 30;
42              daysin[9] = 31;
43              daysin[10] = 30;
44              daysin[11] = 31;
45
46              /* Start with days in the two months. */
47              retval = day2 + (daysin[month1 - 1] - day1);
48
49              /* Add the days in the intervening months */
50              j = month2 - 1;
51              for (i = month1 + 1; i <= j; i++)
52                  retval = daysin[i - 1] + retval;
53          }
54          return retval;
55      }
56
57
58      main()
59
```

```
Rank Threshold (%):
General Threshold (%):

rank  freq.  #stmts  source lines
  1    1.0      8    24 27 31 33 35 41 42 43
  2    2.0     12    30 37 38 39 40 50 51 51 51 52 67 69
  3    3.0      1    36
  4    4.0      1    47
  5    5.0     11    18 23 59 63 63 65 65 69 69 69 69
```

```
H1  * stmts in all available dynamic slices
H2  * stmts in the success set
H3  - stmts with low inclusion frequency in the success set
H4  - stmts in the failure set but not in the success set (H9), plu
H5  * stmts in every success slices ... studying the necesity of th
H6  * stmts in the failure set
H7  - stmts with high inclusion frequency in the failure set
H8  * stmts in every failure slices
H9  * stmts in the failure set but not in the success set <H6 - H2>
H10 * stmts in every failure slices but not in the success set <H8
H11 * stmts in the success set but not in the failure set ... study
H12 * stmts in every success slices but not in the failure set ...
H13 - stmts with high inclusion frequency in the failure set (H7) a
H14 X predicate stmts of those stmts highlighted by the above heuri
H15 - stmts with high influence frequency in a selected failure sli
H16 - stmts with low influence frequency in a selected success slic
```

```
> dynamic program slice on "retval" at line 54 for testcase # 6
> run on testcase 7
stopped at line 59.
> continue
stopped at line 54.
> dynamic program slice on "retval" at line 54 for testcase # 7
> run on testcase 8
stopped at line 59.
> continue
stopped at line 54.
> dynamic program slice on "retval" at line 54 for testcase # 8
> show the statements indicated by H8
> the actual rank threshold is  40.0%
  the actual general threshold is  60.6%
show the statements indicated by H3
> ^
```

Current Testcase #: 8 error-revealing

Figure 6.1 X Window screen dump from SPYDER during a software debugging session

Table 6.1 Software components of SPYDER.

Component	Description
tstgen	Generate a software test/debug directory structure.
tblgen	Compile source files with debug mode (-g) and generate data tables for debugging (e.g., program dependency graph).
tcgen	Manually generate program test data with correctness confirmation.
mdb	Invoke SPYDER X Window interface (main window).
tcmark	Update the test case database (attributes of each test case).
gettcs	Display all test cases.
tblview	Display the data tables in a given software test/debug directory.

debugging paradigm. Table 6.1 summarizes the software components of SPYDER. Figure 6.1 provides a snapshot of the SPYDER interface during a debugging session with the heuristics window.

6.1.1 Screen of SPYDER

The main window of SPYDER (i.e., the left window in Figure 6.1) is divided into five parts from top to bottom as follows. [Agr91]

- File Label: contains the name of the source file currently displayed in Source Panel.
- Source Panel: displays the source code being debugged along with line numbers. The program execution counter (an arrow icon) and setting of break-points (stop icons) are also displayed to the left of associated source lines. Statements in a selected program slice or a search domain suggested by the heuristics will be highlighted in reverse video.
- Commands Panel: contains buttons for functions provided by SPYDER that will be discussed in the next section.

- Output Message Panel: displays functions (command buttons) being invoked and the response messages.
- Test Case Label: contains the testcase number currently being selected (or executed).

The main window of the heuristics selection (i.e., the right lower window in Figure 6.1) is divided into three parts from top to bottom as follows.

- Label: indicates the dynamic slicing option (i.e., approx. dynamic, exact dynamic, or expanded dynamic) selected for this heuristics session.
- Selection Panel: contains a list of all available heuristics.
- Commands Panel: contains command buttons to traverse the heuristics family tree proposed in Chapter 4.2. Details of these commands will be discussed in the next section.

6.1.2 Functions of SPYDER

As mentioned above, we give only a brief summary of functions in SPYDER. Details of the fault localization (heuristics) facility are addressed here.

Functions provided by SPYDER can be characterized into the following five groups and are invoked by clicking buttons in the Commands Panel.

- Slicing Criteria Selection: Four program slicing analysis criteria are available via a set of toggle buttons that are located on the top row of the Command Panel and are labeled as *static analysis*, *approximate dynamic analysis*, *exact dynamic analysis*, and *expanded dynamic analysis.* Users have to select one of the slicing criteria to obtain associated program slices. For dynamic analysis, a test case must be specified via the *testcase* button.
- Program Slicing Commands: Functions in this set allow users to select not only a program slice with respect to the specified variable, location, and test case for dynamic analysis, but also components of the program slice such as data slice, control slice,

reaching definition, and control predicates. They are labeled as *p-slice*, *d-slice*, *c-slice*, *r-defs*, and *c-preds*.

- Backtracking Commands: Like forward execution commands, users are allowed to backtrack program execution one step at a time (*stepback*) or to a preset breakpoint (*backup*).

- Traditional Debugging Commands: These include setting and removing breakpoints (*stop* and *delete*), rerunning and continuing the program execution (*run* and *continue*), printing contents of selected variables (*print* and *select-prt*), and selecting test cases (*testcase*).

- Fault Localization Commands: Heuristics supported by SPYDER are invoked by clicking the *heuristics* button. Moreover, SPYDER provides basic operation commands (*subtract*, *add*, *intersect*, *swap*, and *save* under the button *basic_ops*) to let users conduct their own guessing strategies by dealing with one program slice at a time.

After clicking the heuristics button, SPYDER executes all enabled test cases one by one to collect program slices associated with the dynamic slicing criterion (approx., exact, or expanded dynamic analysis) with respect to a specified variable and location. Then the main window of heuristics pops up. Users can select any heuristics based on their own judgment or use the *random guess*, *expand*, or *refine* commands to traverse the heuristics family tree proposed in Chapter 4.2. Each heuristic will highlight a set of suspicious statements as a reduced search domain. The family tree indicates the superset and subset relationships between its members. The *expand* button will traverse the tree in the bottom–up direction to expand the search domain gradually. On the other hand, the *refine* button will traverse the tree in the top–down direction to refine the search domain. The *guess* button will randomly select a heuristic as a first try. The purpose of fault localization is achieved by employing these heuristics to obtain different reduced search domains.

6.1.3 Implementation Features

The development environment of SPYDER is on a Sun SPARCstation 1 running SunOS-4.1.1. SPYDER is built into the GNU C compiler “gcc” [Sta90] and the GNU source–level debugger “gdb” [Sta89]. Instead of writing a new compiler and debugger, modifying an existing compiler and debugger is preferred because our goal is to show the feasibility of new approaches in a prototype. GNU tools are chosen because they are available for delivery and support full ANSI C. Modification of “gcc” as well as “gdb” is illustrated in [Agr91].

While integrating heuristics into SPYDER, we have to choose the basic unit to be counted — executable statements or nodes (vertices) in a program dependency graph — for the inclusion and influence frequency.

A straightforward and intuitive way to define an executable statement is that a simple statement ends with a semicolon except `if`–statement, `for`–statement, `while`–statement, and `switch/case`–statement. However, each of the following statements is counted as one executable statement: `if (expression)`, `for (expression; expression; expression;)`, `while (expression)`, `switch (expression)`, and `case constant-expression`. The nodes (vertices) in a program dependency graph as defined in [Agr91] correspond to simple statements and predicates where a simple statement is defined as a statement with one memory modification (e.g., an assignment statement “$a = b + c$;”). The `for (expression; expression; expression;)` will be represented by three vertices (one for each expression). Other than the `for`–statement, statements of the exceptions mentioned above will be represented by one vertex per statement. However, if there is more than one variable reference in the read statement, then each memory modification (i.e., variable reference) will be one vertex.

Because dynamic program slicing is the basic instrumentation supported by SPYDER and is implemented based on the program dependency graph, all the dependency analysis is internally associated with nodes rather than executable statements. Although users often deal with source code based on executable statements, information provided by nodes of the program dependency graph will be more accurate than those of executable statements in terms of dependency analysis. In addition, this is consistent between internal representation and external display that statements suggested by a heuristic are based on the corresponding nodes. We thus chose to implement the heuristics based on the nodes in a program dependency graph to avoid confusion. However, the interpretation of experimental results based on executable statements addressed in [PS93] is similar to those based on the vertices in a program dependency graph addressed in the next section.

6.2 An Experiment

We conducted a simple experiment to confirm the effectiveness and feasibility of the proposed heuristics. Results of coverage and effectiveness analysis are presented. The coverage analysis determines whether the faulty statements are in the reduced search domain suggested by our approaches, and the effectiveness comparison evaluates the size of the reduced search domain. Exhaustive experiments were not the goal of this dissertation and are suggested as promising future work.

Heuristics proposed in Chapter 4 (i.e., heuristics without the assistance of further testing information) and Critical Slicing proposed in Chapter 5.1 are examined. As mentioned in Chapter 5.2, we did not conduct experiments for the heuristics and hints proposed in that section because of the limitation of existing prototype tools.

In this section, we first describe the experimental methods as well as 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 [Lip79] that indicate 26% logic faults, 18% data handling faults, 9% computational faults, . . ., etc. Although the sample space of tested programs is not large, we try to balance the features of our samples according to 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 shows that the proposed fault localization techniques are a promising approach to handle faulty programs with similar features of our tested programs.

6.2.1 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, referred to as *coverage 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.*

Coverage Analysis

This analysis assures that the search domains suggested by fault localization approaches 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 heuristics. Moreover, for each heuristic, the percentage of total tested programs with positive coverage analysis results is calculated. The higher the positive coverage percentage a heuristic has, the more the affirmative effect of fault localization a heuristic 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.

For heuristics proposed in Chapter 4, the size of the reduced search domain suggested by each heuristic is compared with the size of the original program as well as the size of the root heuristic (H1). The following two ratios are computed for each heuristic with respect to a tested program.

$$\mathcal{R}'_a = \frac{\text{\# of statements indicated by a heuristic}}{\text{\# of executable statements of a tested program}} \tag{6.1}$$

$$\mathcal{R}'_h = \frac{\text{\# of statements indicated by a heuristic}}{\text{\# of statements indicated by H1}} \tag{6.2}$$

As mentioned in Chapter 6.1.3, the implementation of heuristics in SPYDER is based on vertices of the program dependency graph of a given tested program rather than the executable statements of the program. The above two equations are rewritten as:

$$\mathcal{R}'_a = \frac{\text{\# of vertices indicated by a heuristic}}{\text{\# of vertices in the corresponding program dependency graph}} \tag{6.3}$$

$$\mathcal{R}'_h = \frac{\text{\# of vertices indicated by a heuristic}}{\text{\# of vertices indicated by H1}} \tag{6.4}$$

In Chapter 4.1, two kinds of threshold for heuristics with threshold requirements (Heuristics 3, 4, 7, and 13) are introduced. They are measured in this experiment.

$$\text{rank threshold} = \frac{\text{the rank of the critical level}}{\text{\# of ranked levels}} \tag{6.5}$$

$$\text{general threshold} = \frac{\text{\# of vertices (stmts) within and below the critical level}}{\text{\# of vertices (stmts) in a selected heuristic}} \tag{6.6}$$

A similar experiment based on executable statements in Equations 6.1 and 6.2 has been reported in [PS93].

For Critical Slicing as described in Chapter 5.1, 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 expanded dynamic program slice, and the size of the corresponding exact dynamic program slice. The first and second ones 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 error–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.

$$\mathcal{R}'_{cs} = \frac{\text{\# of statements in a CS}}{\text{\# of executable statements of a tested program}} \tag{6.7}$$

$$\mathcal{R}'_e = \frac{\text{\# of statements in a CS}}{\text{\# of statements in the corresponding EDPS}} \tag{6.8}$$

$$\mathcal{R}'_d = \frac{\text{\# of statements in a CS}}{\text{\# of statements in the corresponding DPS}} \tag{6.9}$$

While considering the degree of improvement presented in $\mathcal{R}'_a$, $\mathcal{R}'_h$, $\mathcal{R}'_{cs}$, and $\mathcal{R}'_e$, we are actually interested in the reduction rate between our approaches and the selected domain, i.e., the percentage of reduction from the size of a tested program to the size of region suggested by a heuristic. Therefore, these ratios are redefined as follows to be easily interpreted.

$$\mathcal{R}_a = \begin{cases} 1 - \mathcal{R}'_a & \text{if } \mathcal{R}'_a > 0 \\ 0 & \text{if } \mathcal{R}'_a = 0 \end{cases} \tag{6.10}$$

$$\mathcal{R}_h = \begin{cases} 1 - \mathcal{R}'_h & \text{if } \mathcal{R}'_h > 0 \\ 0 & \text{if } \mathcal{R}'_h = 0 \end{cases} \tag{6.11}$$

$$\mathcal{R}_{cs} = \begin{cases} 1 - \mathcal{R}'_{cs} & \text{if } \mathcal{R}'_{cs} > 0 \\ 0 & \text{if } \mathcal{R}'_{cs} = 0 \end{cases} \tag{6.12}$$

$$\mathcal{R}_e = \begin{cases} 1 - \mathcal{R}'_e & \text{if } \mathcal{R}'_e \neq 0 \\ 0 & \text{if } \mathcal{R}'_e = 0 \end{cases} \tag{6.13}$$

$$\mathcal{R}_d = \begin{cases} 1 - \mathcal{R}'_d & \text{if } \mathcal{R}'_d \neq 0 \\ 0 & \text{if } \mathcal{R}'_d = 0 \end{cases} \tag{6.14}$$

To compare effectiveness of heuristics by analyzing $\mathcal{R}_a$ and $\mathcal{R}_h$, a large value of a ratio (reduction rate) indicates that the degree of effectiveness of the associated heuristic is high. $\mathcal{R}_{cs}$, $\mathcal{R}_e$, and $\mathcal{R}_d$ 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 has, the more effective a corresponding approach is.

Table 6.2 Tested programs. #ES, #V, #Bl, #De, #P-u, and #A-u represent the number of executable statements, vertices in its program dependency graph, blocks, decisions, p–uses, and all–uses, respectively.

Program	#ES	#V	#Bl	#De	#P-u	#A-u	fault types
P1: aveg	35	57	36	18	40	79	wrong logical expression
P2: calend	29	51	22	10	16	31	wrong logical operator
P3: find1	33	53	32	18	80	124	wrong variable reference
P4: find2	33	53	32	18	80	124	wrong variable reference
P5: find3	32	52	32	18	80	122	missing a statement & faults of find1 and find2
P6: gcd	57	97	57	36	124	230	wrong initialization (value)
P7: naur1	37	60	28	18	48	80	missing simple logical expression
P8: naur2	37	60	28	18	50	82	missing simple logical expression
P9: naur3	36	58	28	18	46	78	missing predicate statement
P10: transp	155	319	156	73	135	361	wrong initialization (value)
P11: trityp	37	55	47	39	99	113	wrong logical operators

6.2.2 Tested Programs

Eleven test programs were selected and constructed from seven programs. Most of these programs were collected from previous studies and are well–known programs with previously studied faults.[1] They have different characteristics in terms of program size, number of functions, the type of program application (e.g., matrix calculation vs. text processing), and fault types as well as locations.

Table 6.2 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 testing tool — ATAC (Automatic Test Analysis for C programs) [HL91], developed at Bellcore.

[1] The programs are described in Appendix C.

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.

The data flow coverage criteria (columns 4 to 7) help us understand the complexity of a tested program. Fault types of 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 heuristics 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. We believe that for P5 and P11 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. As mentioned in Chapter 3.3, 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 coverage analysis. The ratio of effectiveness comparison in this case will be compared with others to perceive any significant difference.

Table 6.3 ATAC's measurement of test case adequacy

Prog.	TC	#	% blocks		% decisions		% p-uses		% all-uses	
P1: aveg	T_s	8 (9)	100	(36/36)	100	(18/18)	73	(29/40)	81	(64/79)
	T_f	6	100	(36/36)	94	(17/18)	70	(28/40)	77	(61/79)
P2: calend	T_s	5	100	(22/22)	90	(9/10)	94	(15/16)	97	(30/31)
	T_f	3	91	(20/22)	60	(6/10)	69	(11/16)	74	(23/31)
P3: find1	T_s	6	100	(32/32)	100	(18/18)	79	(63/80)	84	(104/124)
	T_f	3	97	(31/32)	94	(17/18)	76	(61/80)	82	(102/124)
P4: find2	T_s	5	100	(32/32)	100	(18/18)	79	(63/80)	84	(104/124)
	T_f	3	97	(31/32)	94	(17/18)	74	(59/80)	81	(100/124)
P5: find3	T_s	6	100	(32/32)	100	(18/18)	80	(64/80)	85	(104/122)
	T_f	4	97	(31/32)	94	(17/18)	75	(60/80)	80	(98/122)
P6: gcd	T_s	8(10)	100	(57/57)	89	(32/36)	69	(85/124)	71	(163/230)
	T_f	9	95	(54/57)	81	(29/36)	64	(79/124)	67	(153/230)
P7: naur1	T_s	12	100	(28/28)	100	(18/18)	65	(31/48)	66	(53/80)
	T_f	2	96	(27/28)	89	(16/18)	56	(27/48)	61	(49/80)
P8: naur2	T_s	2	100	(28/28)	100	(18/18)	66	(33/50)	67	(55/82)
	T_f	3	100	(28/28)	100	(18/18)	62	(31/50)	65	(53/82)
P9: naur3	T_s	6	100	(28/28)	100	(18/18)	70	(32/46)	69	(54/78)
	T_f	7	100	(28/28)	100	(18/18)	78	(36/46)	79	(62/78)
P10: transp	T_s	5(7)	94	(146/156)	89	(65/73)	81	(110/135)	85	(307/361)
	T_f	4	96	(150/156)	90	(66/73)	79	(107/135)	84	(304/361)
P11: trityp	T_s	6(14)	98	(46/47)	97	(38/39)	76	(75/99)	78	(88/113)
	T_f	3	66	(31/47)	51	(20/39)	37	(37/99)	39	(44/113)

In Chapter 3.1, we suggest conducting a thorough test before applying the proposed heuristics. ATAC was used to conduct the thorough test by obtaining two test case sets, the non–error–revealing (success) test case set T_s and the error–revealing (failure) test case set 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). Each test case satisfies the criteria to a certain degree when executed against the tested program. A summary of the degree of satisfaction presented in Table 6.3 consists of both percentage and counts of all four criteria to show the adequacy of selected test cases. ATAC was employed to satisfy the coverage of criteria as much as possible and to guarantee the adequacy. In our experiment, test cases in T_s were added to improve the degree of satisfaction without causing program failures. Test cases in T_f were added to improve the degree of satisfaction under program failures. Information presented in Table 6.3 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 6.3.

According to the analysis in Chapter 3.1, only test cases associated with input domains causing failure are useful for debugging. 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_s of P1, the total number of non–error–revealing test cases obtained from ATAC is nine, which is in a pair of parentheses, but only eight of them are related to program failures and useful for debugging. Other non–error–revealing test cases are the "noise" in the debugging process. So are programs P6, P10, and P11.

6.2.3 Results of Heuristics without the Assistance of Further Testing Information

In this experiment, the exact dynamic program slicing approach (DPS) is used because all faulty statements (except the missing statements) are always covered by the corresponding exact dynamic slices. Also, DPS deals with a smaller region than EDPS does.

The group of heuristics under H2 (success set) without threshold requirements (i.e., H5, H11, and H12) are looking for statements highly involved in the success slices, and the purpose of employing these heuristics is to understand the necessity of the statements for correct results. Therefore, it is not appropriate to compare results of these heuristics with others. Results of this group and other heuristics are separately presented.

6.2.3.1 Coverage Analysis

Heuristics 1 to 16 Except H5, H11, and H12

Table 6.4 presents the result of coverage analysis for heuristics that consider all available test cases (i.e., Heuristics 1 to 14) except H5, H11, and H12. They are grouped according to their characteristics in the heuristics family tree. In the table, a symbol $\surd$ indicates that the reduced domain suggested by a heuristic still contains faulty statements (i.e., a positive result), and a symbol $\times$ indicates a negative result. Thresholds of the heuristics with

Table 6.4 Coverage analysis for Heuristics 1–4, 6–10, and 13–14 proposed in Chapter 4 except H5, H11, and H12. √ and × indicate positive and negative results, respectively. The last column indicates the percentage of positive results.

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	%
Heu.	aveg	calend	find1	find2	find3	gcd	naur1	naur2	naur3	transp	trityp	Pos.
Root and subroot heuristics in the family tree												
H1	√	√	√	√	√	√	√	√	×	√	√	91%
H2	√	√	√	√	√	×	√	√	×	×	√	73%
H6	√	√	√	√	√	√	√	√	×	√	√	91%
Heuristics under H6 (failure set) without threshold												
H8	√	√	√	√	√	√	√	√	×	√	√	91%
H10	×	×	×	×	×	√	×	×	×	√	×	19%
H9	×	×	×	×	×	√	×	×	×	√	×	19%
Heuristics with threshold requirements												
H3	√	√	√	√	√	×	√	√	×	×	√	73%
H4	√	√	√	√	√	√	√	√	×	√	√	91%
H7	√	√	√	√	√	√	√	√	×	√	√	91%
H13	√	√	√	√	√	√	√	√	×	√	√	91%
Supplemental heuristics for predicate statements												
H14	√	√	√	√	√	×	√	√	×	×	√	N/A

Table 6.5 Coverage Analysis for Heuristic 16 — statements with low influence frequency in a selected *success* slice. $|T_s|$, $\surd$, and $\times$ indicate the number of non–error–revealing (success) test cases, a positive result, and a negative result, respectively. The last column indicates the percentage of positive results.

																%
Prog.	$\|T_s\|$	t_{s_1}	t_{s_2}	t_{s_3}	t_{s_4}	t_{s_5}	t_{s_6}	t_{s_7}	t_{s_8}	t_{s_9}	$t_{s_{10}}$	$t_{s_{11}}$	$t_{s_{12}}$	$t_{s_{13}}$	$t_{s_{14}}$	Pos.
P1:	8(9)	N/A	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$						100%
P2:	5	$\times$	$\times$	$\surd$	$\surd$	$\times$										40%
P3:	6	$\times$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$									84%
P4:	5	$\times$	$\surd$	$\surd$	$\times$	$\surd$										60%
P5:	6	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$									100%
P6:	8(10)	N/A	N/A	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$				0%
P7:	12	$\surd$	$\surd$	$\times$	$\times$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$			84%
P8:	2	$\times$	$\surd$													50%
P9:	6	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$									0%
P10:	5(7)	$\times$	N/A	$\times$	N/A	$\times$	$\times$	$\times$								0%
P11:	6(14)	$\surd$	N/A	N/A	$\surd$	N/A	N/A	$\surd$	$\surd$	$\surd$	N/A	N/A	N/A	N/A	$\surd$	100%

threshold requirements will be set to 100% for the coverage analysis. The percentage of positive results for each heuristic with respect to all tested programs is presented in the last column.

Heuristic 14 is designed to highlight only predicate statements for supplemental purposes. If the faulty statements are not predicates, then they will not be covered by H14. Thus the percentage of positive results is not applicable for this heuristics.

Without considering P9 (with a missing statement) in Table 6.4, we conclude that most heuristics have high positive coverage and the faulty statements of each tested program can always be highlighted by many heuristics. For programs with the missing statement fault, it is improper to conduct coverage analysis because the missing statements will not be highlighted by any dynamic slices. We thus change the measurement of coverage analysis from statements to basic statement blocks. If any statements in the basic block the missing statement also belongs to is highlighted by a heuristic, then the coverage analysis becomes positive. We believe that examining statements in the same block the missing statements were supposed to be is a strong clue to discover missing statement faults. In this case,

Table 6.6 Coverage analysis for heuristics under H2 (success set) without threshold requirements proposed in Chapter 4. √ and × indicate positive and negative results, respectively. The last column indicates the percentage of positive results.

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	%
Heu.	aveg	calend	find1	find2	find3	gcd	naur1	naur2	naur3	transp	trityp	Pos.
Heuristics under H2 (success set) without threshold												
H5	√	×	×	×	√	×	×	×	×	×	√	28%
H12	×	×	×	×	×	×	×	×	×	×	×	0%
H11	×	×	×	×	×	×	×	×	×	×	×	0%
$\overline{\text{H5}}$	×	√	√	√	×	√	√	√	×	√	×	64%
$\overline{\text{H12}}$	√	√	√	√	√	√	√	√	×	√	√	91%
$\overline{\text{H11}}$	√	√	√	√	√	√	√	√	×	√	√	91%

we get 100% coverage analysis for all heuristics in Table 6.4 except H9 and H10 that are extremely well suited to identify wrong initialization only.

Heuristics based on the influence frequency (i.e., Heuristics 15 and 16) are not included in Table 6.4 because they deal with one test case at a time. The coverage analysis for H16 is presented in Table 6.5. The entry with "N/A" means that the corresponding non–error–revealing test case is not used by our heuristics for fault localization. The negative result of an entry indicates that the faulty statements of a tested program are not covered by the corresponding success slice at all. The behavior of the faulty statements will decide the percentage of positive coverage analysis for the given program.

The coverage analysis for H15 is not needed because the heuristic highlights statements in a selected failure slice. At least one of the faulty statements in the program will be included in the failure slice. Thus, the coverage analysis is always positive, except for the missing statement fault type.

Group of Heuristics Under H2 (Success Set) Without Threshold Requirements

This group of heuristics (H5, H12, and H11), presented in Table 6.6, has the worst positive coverage because they are looking for statements highly involved in the success slices. Only few statements are highlighted by these heuristics as indicated in the effectiveness comparison. These statements are used for further analysis after studying the necessity of them for correct results. Thus, they should not be merely treated as a reduced search domain containing faults. It is not appropriate to employ these heuristics with others at the same time.

6.2.3.2 Effectiveness Comparison

This part of the experiment was conducted on SPYDER, which has the proposed heuristics implemented based on the vertices of the program dependency graph as mentioned before. Equations 6.10, 6.11, 6.5, and 6.6 are chosen to calculate$\mathcal{R}_a$, $\mathcal{R}_h$, the rank threshold, and the general threshold, respectively. Results of the comparison are presented in figures to show the overall effectiveness and the degree of improvement. Heuristics with positive coverage analysis are indicated by the symbol $\surd$ in the figures for reference. The presentation of effectiveness comparison is partitioned into three groups.

- Group 1: heuristics considering all available test cases except H5, H11, and H12, i.e., heuristics in Table 6.4. Figures 6.2 and 6.3 present the results in the order of the heuristics.

- Group 2: heuristics under H2 (success set) without threshold requirements, i.e., heuristics in Table 6.6. Figure 6.4 presents the results in the order of the heuristics.

  Figure D.1 in Appendix D has the same results as in Figures 6.2 to 6.4 (both Groups 1 and 2) but presents the results in the order of the programs.

- Group 3: heuristics based on the influence frequency, i.e., H15 in Figure 6.5 and H16 in Figure 6.6.

- Group 4: comparison between the rank threshold and the general threshold. Figures 6.7 and D.2 present the comparison results of the most effective heuristics

H3, H4, H7, and H13 in different format. Thresholds for influency frequency are presented in Figures 6.8 and 6.9.

To measure the effectiveness of heuristics with threshold requirements (i.e., Heuristics 3, 4, 7, 13, 15, and 16), we set the threshold on the critical point defined in Chapter 4.1 to highlight a minimum set of statements still containing faulty statements. The ratios $\mathcal{R}_a$ and $\mathcal{R}_h$ are then calculated based on the minimum set of statements. For program “P9: naur3” with the fault type — missing statements, which cannot by highlighted by any of our approaches as mentioned earlier, setting the threshold to 100% does not provide helpful information for analysis. Therefore, an alternative method is employed. We first identify some statements that are closely related to the missing statement and would lead to discovering the missing one. Three statements are identified in P9. Then, the threshold is enlarged until all three statements are included, and a reduced domain is obtained. Finally, $\mathcal{R}_a$ and $\mathcal{R}_h$ for heuristics applied to P9 are calculated based on the domain. In this case, the effectiveness comparison for programs with missing statements is similar to, although not the same as, the one for programs with other fault types.

Group 1: Heuristics 1 to 14 in Table 6.4

Figures 6.2 and 6.3 illustrate the effectiveness of each heuristic against all tested programs. We are interested in the ratios with high reduction rate in the figures.

Because H1 is the root of the heuristics family tree and highlights all statements involved in the proposed heuristics (i.e., the maximum size), $\mathcal{R}_h$ of H1 is always 0.0 that means no reduction and can be ignored. $\mathcal{R}_a$ of H1 in Figure 6.2 indicates the reduction from a whole tested program to the set of statements involved in all dynamic slices (i.e., the basis set) for our heuristics. The $\mathcal{R}_a$ of H1 for tested programs ranges from 0.35 to 0.67, and most of them reside between 0.40 and 0.45. From the overall point of view, we conclude that in this experiment our proposed heuristics (the basis set) averages around a 43% reduction from the whole program.

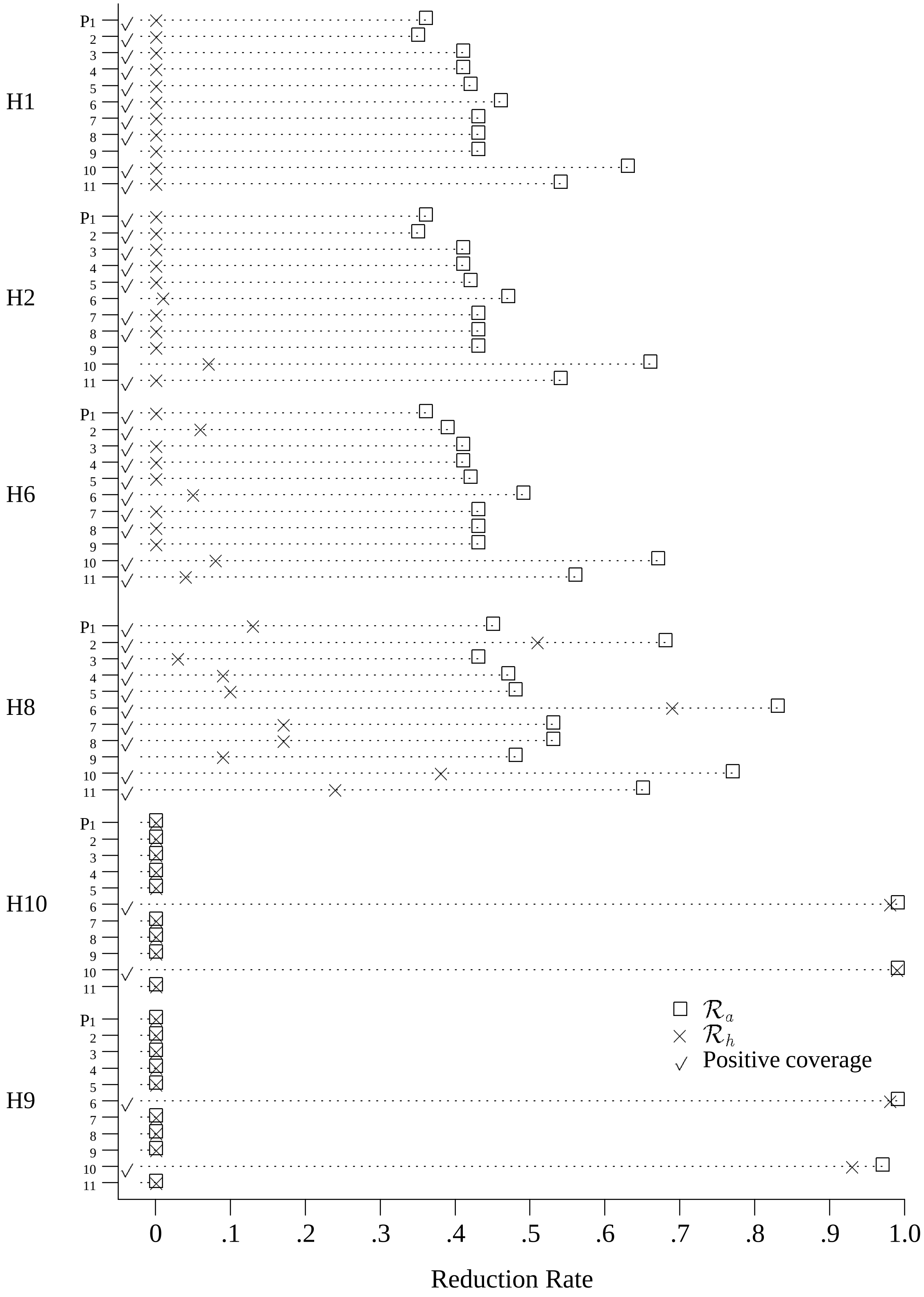


Figure 6.2 Effectiveness comparison of heuristics, part I — heuristics as the root and subroots in the family tree and heuristics under H6 (failure set) without threshold requirements.

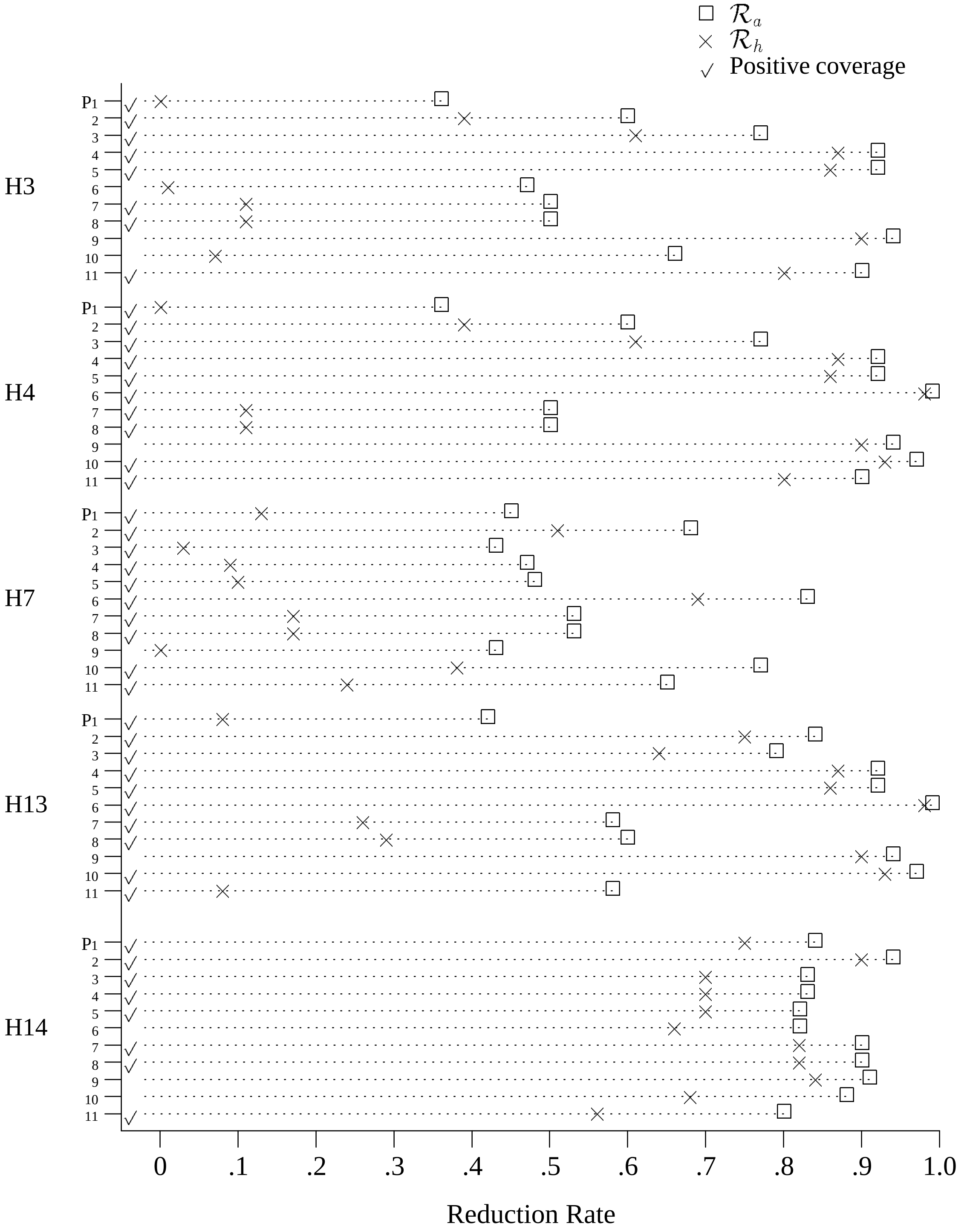


Figure 6.3 Effectiveness comparison of heuristics, part II — heuristics with threshold requirements and the supplemental heuristics for predicate statements.

The scope of each heuristic (i.e., the suggested search domain) is a subset of the scope of the corresponding H1 slice. This means that the search domain of other heuristics in the tree will be further refined from the one of H1.

H2 and H6 are subroots in the family tree for the success set and the failure set, respectively. Thus, their results are similar to those of H1.

H9 and H10 deal with statements in the difference between the failure set and the success set, and have extreme results as shown in Figure 6.2. For most programs, these two heuristics will highlight nothing because of the rigorous approach. However, if the suggested search domain is not empty, the domain is very small and contains faulty statements, especially for failed programs with single faulty statement. Programs “P6: gcd and “P10: transp” have the same fault type — wrong variable initialization. H9 and H10 are extremely effective for these two programs, i.e., having 99% reduction rate with positive coverage analysis. From this experiment, we suggest employing these two heuristics first for the extremely small search domain. If the domain is not empty, then our goal is quickly achieved. Otherwise, continue to employ other heuristics.

Heuristics with threshold requirements (i.e., Heuristics 3, 4, 7, and 13) are presented in Figure 6.3. By observing $\mathcal{R}_h$, we find most of these heuristics will provide a further refined search domain from the basis H1. Around two–third’s of $\mathcal{R}_h$ are above 0.2 (20% reduction) which means the corresponding domains are effectively reduced from H1. Moreover, all $\mathcal{R}_a$, which are associated with the whole program, are above 0.35. Many $\mathcal{R}_h$ and $\mathcal{R}_a$ with high reduction rate in Figure 6.3 show the promise of these heuristics. In addition, H4, H7, and H13 have positive coverage for all programs except P9 (with a missing statement). We conclude that heuristics in this category are the most effective ones in the heuristics family.

Heuristic 14 is designed to enhance the location of faults in predicate expressions when other heuristics are not effective enough. The total number of predicates in a program is used as the numerator of the corresponding $\mathcal{R}_a$ and $\mathcal{R}_h$. Therefore the ratios of H14 in Figure 6.3 only show the percentage of predicate statements in a tested program.

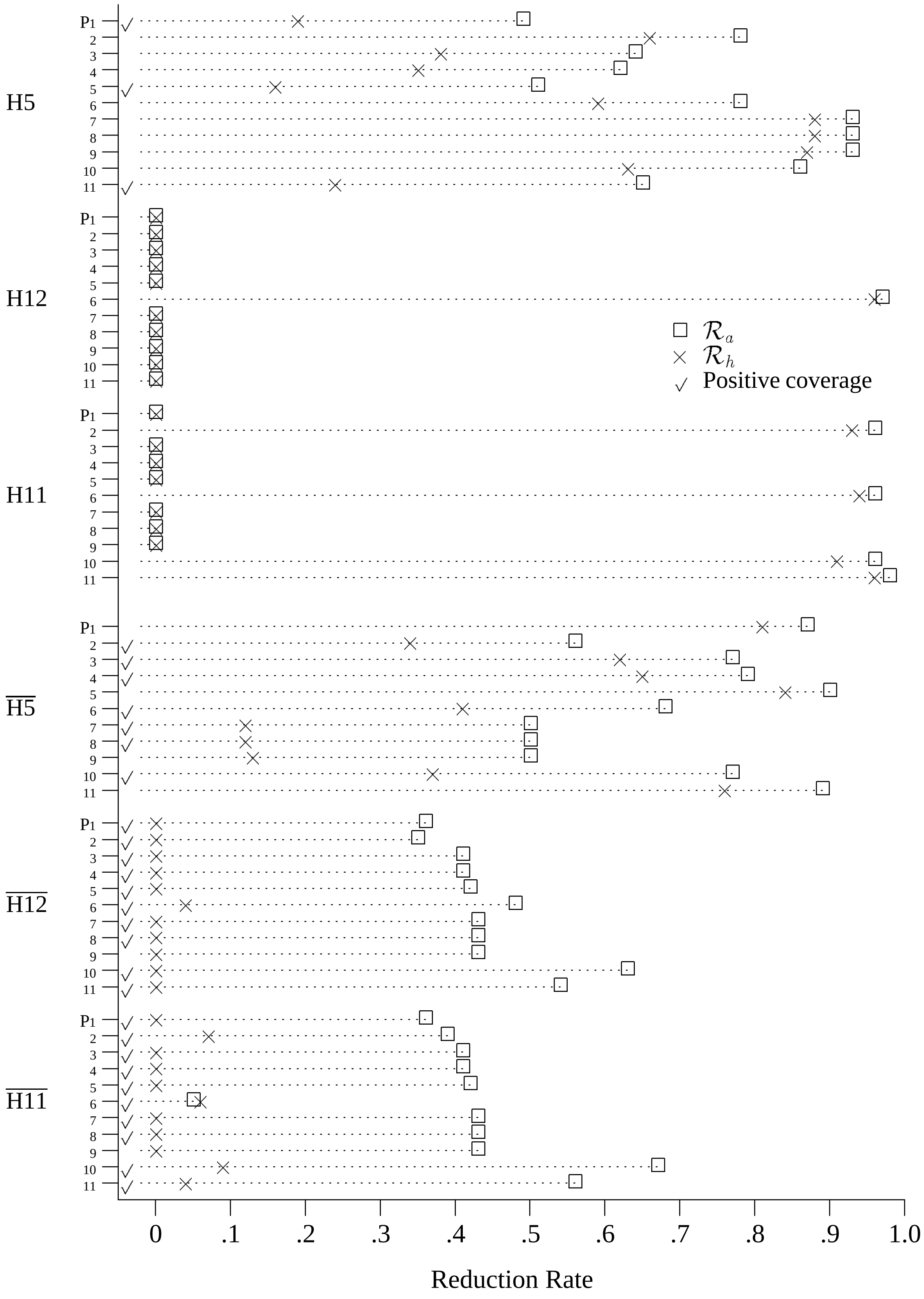


Figure 6.4 Effectiveness comparison of heuristics, part III — heuristics under H2 (success set) without threshold requirements.

Group 2: Heuristics under H2 (success set) without threshold requirements.

As mentioned in the previous section, heuristics in this group are looking for statements highly involved in the success slices. H11 and H12 have a more rigorous approach to focus on statements in the success set but not in the failure set. Therefore, these two heuristics will not highlight many statements. However, studying the necessity of the few highlighted statements (e.g., H12 and H11 on P6) for correct results and semantics of the statements could help users discover the faults. If an empty set is provided by one of these three heuristics, the corresponding complement heuristic is ignored because in this case the search domain indicated by the complement heuristic will have the same size as the one suggested by H1.

From our experimental results of this group heuristics presented in Figure 6.4, we suggest first applying other heuristics that are more effective than heuristics in this group. Statements highlighted by this group heuristics will be used by others.

Figure D.1 presents the results in Figures 6.2 to 6.4 from a different point of view. The result is presented by the order of the programs employing all heuristics. We found that for every tested program there exists at least one heuristic with positive coverage analysis as well as high reduction rate of both $\mathcal{R}_a$ and $\mathcal{R}_h$. This supports our belief that we can always find at least one effective heuristic to localize faults in a tested program.

Group 3: Heuristics based on the influence frequency (Heuristics 15 and 16)

By comparing H15 in Figure 6.5 and H16 in Figure 6.6, we notice that H15 is more effective than H16 in terms of high $\mathcal{R}_a$ and $\mathcal{R}_h$. H15 deals with error–revealing test cases causing the program to fail. For H15 in Figure 6.5, around two–third's of $\mathcal{R}_h$ are above 0.2 (20% reduction rate) which means the corresponding domains are effectively reduced from H1. Moreover, all $\mathcal{R}_a$ of H15, which is associated with the whole program, are above 0.35. The large amount of $\mathcal{R}_h$ and $\mathcal{R}_a$ in Figure 6.5 with high reduction rate shows the promise of H15. From this experiment, we suggest applying H15 rather than H16 while considering the influence frequency.

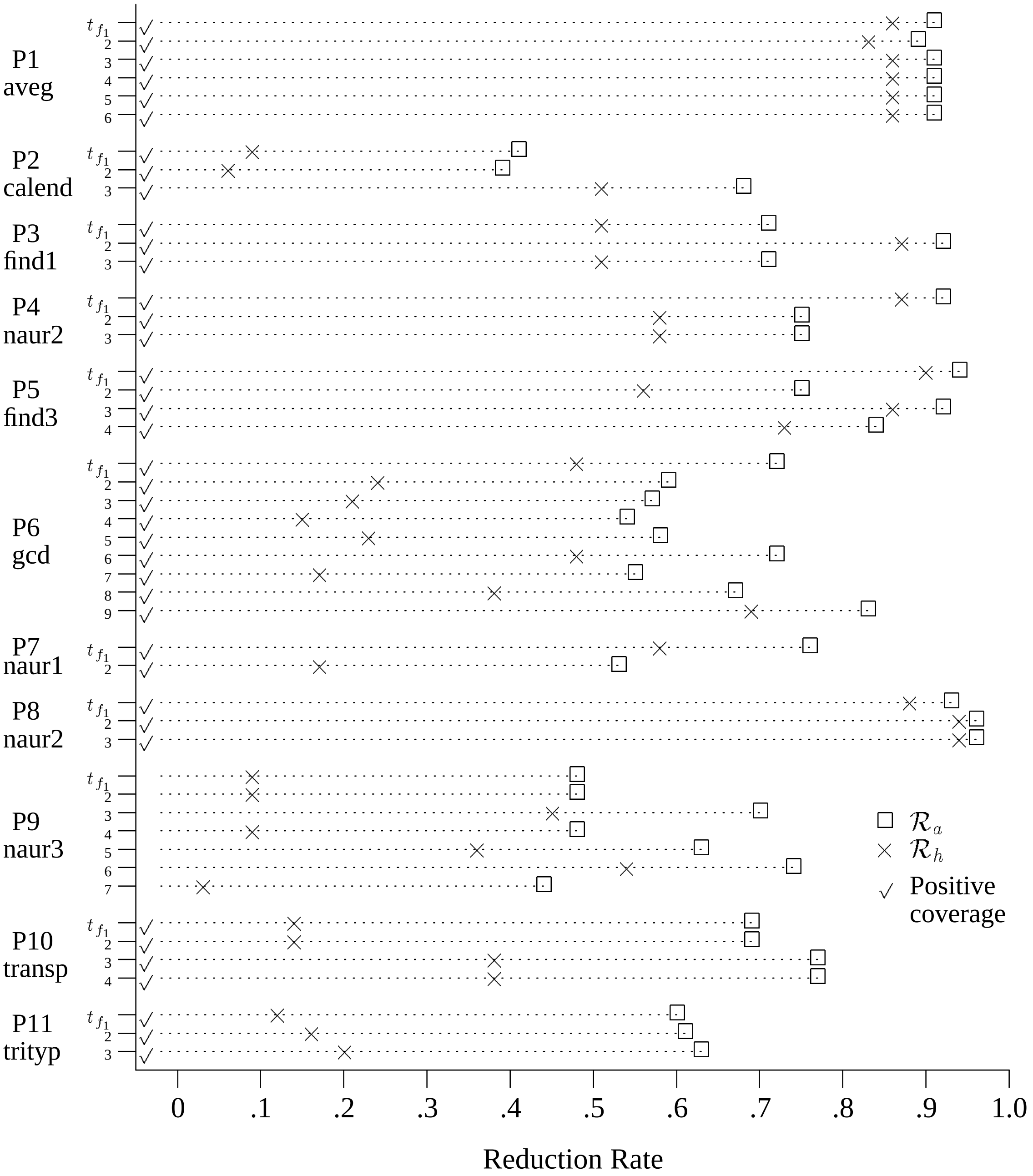


Figure 6.5 Effectiveness comparison of Heuristic 15.

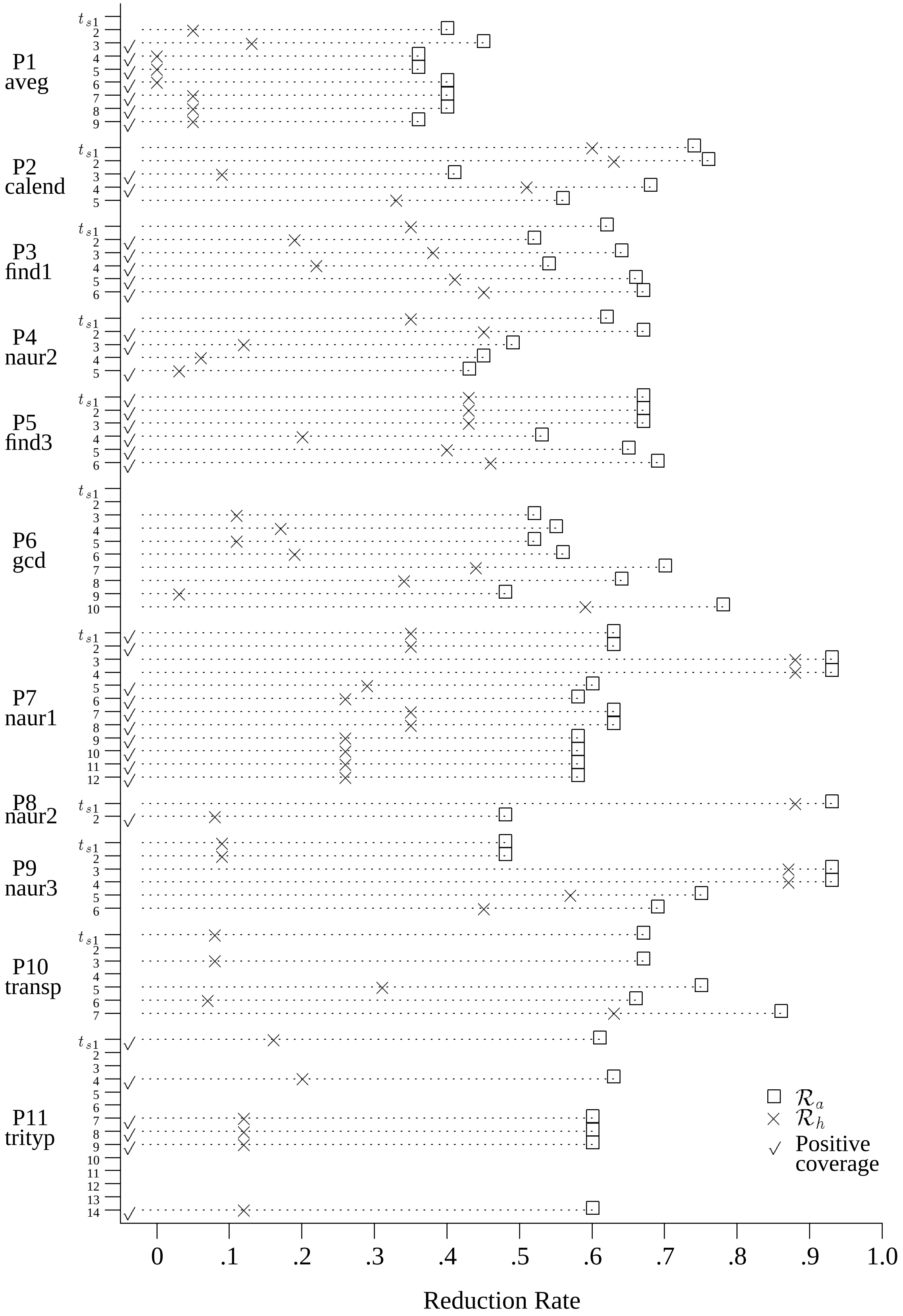


Figure 6.6 Effectiveness comparison of Heuristic 16.

Group 4: Comparison between the rank threshold and the general threshold

The general and rank thresholds for Heuristics 3, 4, 7, and H13 are presented in Figure 6.7. The same results presented by the order of tested programs is in Figure D.2.

We chose the thresholds of H3, indicating statements with low inclusion frequency in the success set — $\bigcup_{i=1}^{|T_s|} Dyn(P, v, l, T_{s_i})$, on program "P2: calend" as an example to explain the construction of ranks, groups, and the critical level as introduced in Chapter 4.1. The screen dump of SPYDER for this example is presented in Figure 6.1. Five groups with different inclusion frequency are obtained from the success set of P2: the first group (ranked 1) has the lowest inclusion frequency 1 and contains eight vertices; the second group (ranked 2) contains twelve vertices with inclusion frequency 2; the third group (ranked 3) contains one vertex with inclusion frequency 3; the fourth group (ranked 4) contains one vertex with inclusion frequency 4; and the fifth group (ranked 5) contains eleven vertices 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). To get an effective threshold, we had better set the threshold to the critical point so that those twenty vertices (including the faulty one) in the first group (having eight statements) as well as the second group (having twelve statements) are highlighted. Thus we set the rank threshold (defined in Equation 6.5) as $2/5 = 0.4$ and the general threshold (defined in Equation 6.6) as $20/33 = 0.6$.

The threshold of a heuristic will be gradually enlarged until 1.0. If a heuristic has 1.0 threshold but does not have the indication of positive coverage ($\surd$), this means the faulty statements cannot be highlighted by the heuristic, e.g., e.g., H3 on P6 and P10, and H7 on P9.

As mentioned in Chapter 4.1, an efficient threshold, which makes the suggested domain reasonably small and consistently contain faults, is highly desirable for the first guess. In Figure 6.7, general thresholds ($\diamond$) are distributed from 0.02 to 1.0. A standard threshold cannot be decided from this situation. However, we find that around 69% of rank thresholds ($\bullet$) have a value ≤ 0.5 and around 91% of rank thresholds have a value ≤ 0.75. This provides a consistent metric for evaluation. Moreover, the rank threshold is easy to use

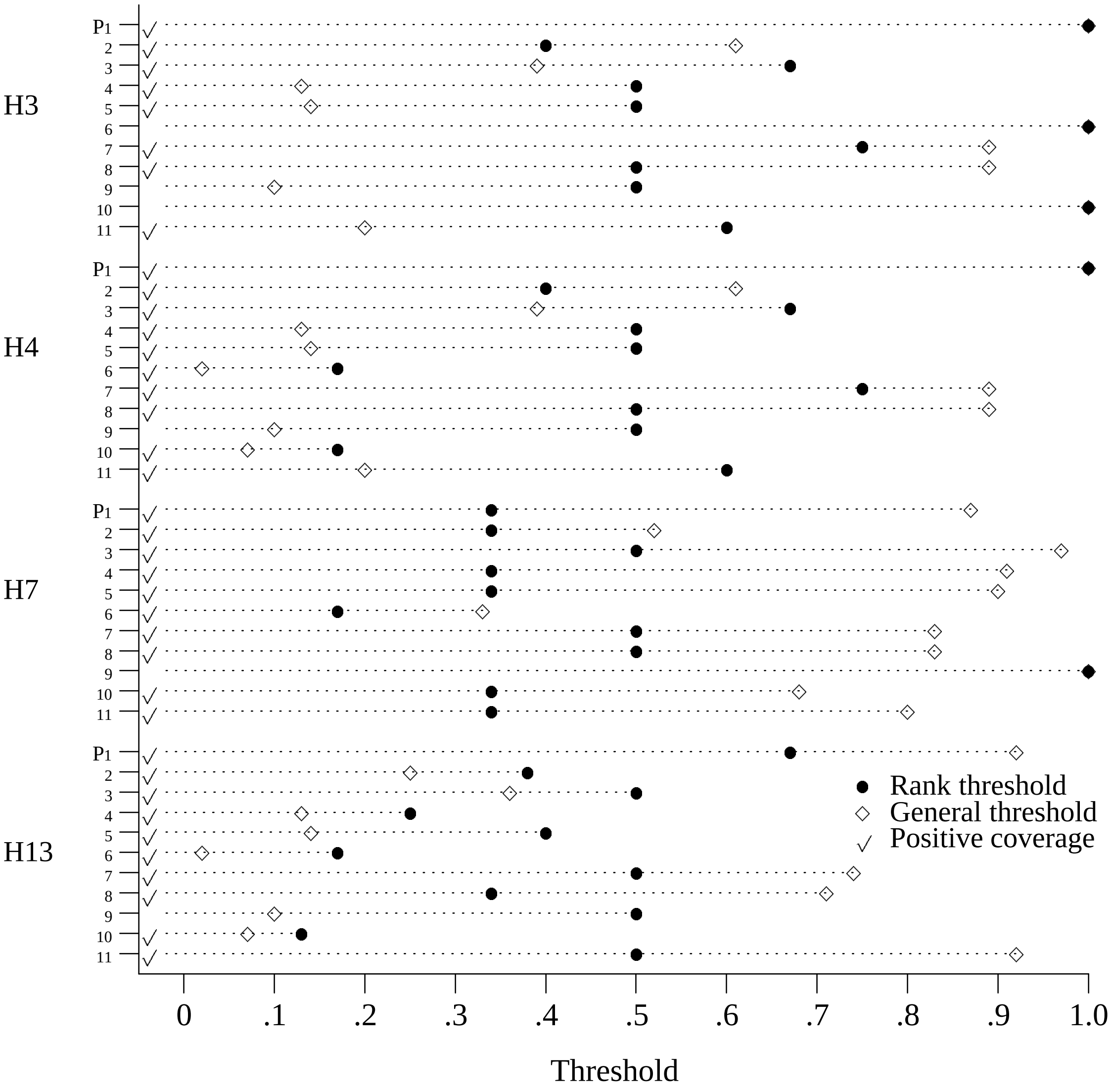


Figure 6.7 Effectiveness comparison between rank threshold and general threshold for Heuristics 3, 4, 7, and 13.

(e.g., from the first to the last ranked level, gradually). Therefore, we chose the rank threshold with the first 75% (or 50%) of ranked levels as a standard threshold for the first–time criterion when employing H3, H4, H7, and H13.

The rank and general thresholds of H15 and H16 cannot be characterized according to their distribution in Figures 6.8 and 6.9. For H16 in Figure 6.9, the range of the general threshold is even smaller than the range of the rank threshold. The general threshold is therefore preferred in H16. Because these two heuristics are based on the influence frequency for one dynamic slice at a time, users employ them for detailed (local) analysis with respect to a selected test case. Thus, the effect of choosing the standard threshold for H15 and H16 is not an important issue.

From the coverage analysis and effective comparison mentioned above, results of this experiment support our claim in Chapter 4 that the overall debugging power from uniting these heuristics is expected to surpass that of currently used debugging tools and to provide effective fault localization techniques.

6.2.4 Results of Critical Slicing

According to the definition and features of Critical Slicing (CS) in Chapter 5.1.1, statements are the basic unit for constructing critical slices. Therefore, Equations 6.12, 6.13, and 6.14 are used to evaluate experimental results of critical slices. Every error–revealing test case will construct one critical slice, and all non–error–revealing test cases are not used in this experiment.

6.2.4.1 Coverage Analysis

Table 6.7 presents results of coverage 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_2} of program “P3:find1”,

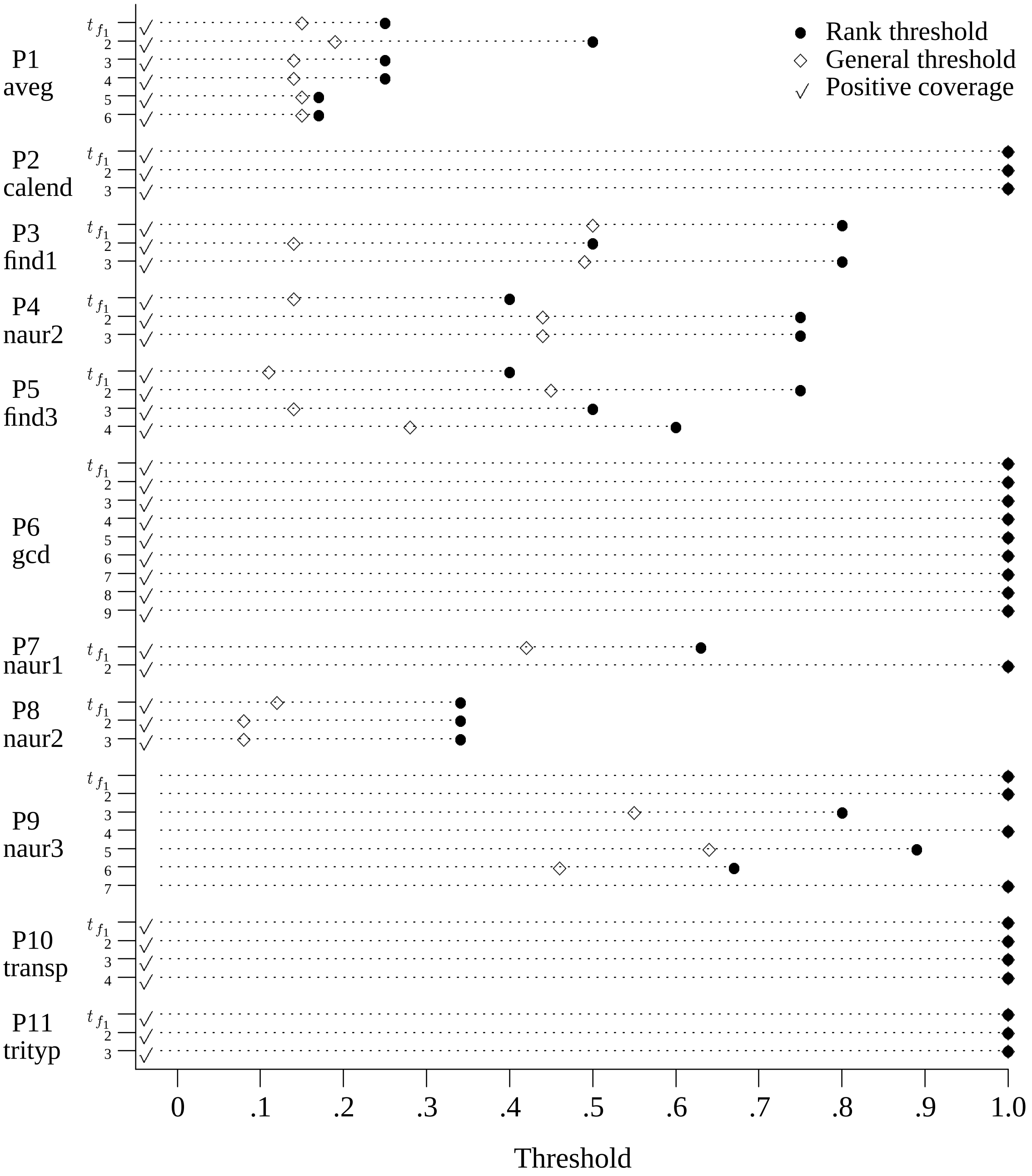


Figure 6.8 Effectiveness comparison between rank and general threshold for Heuristic 15.

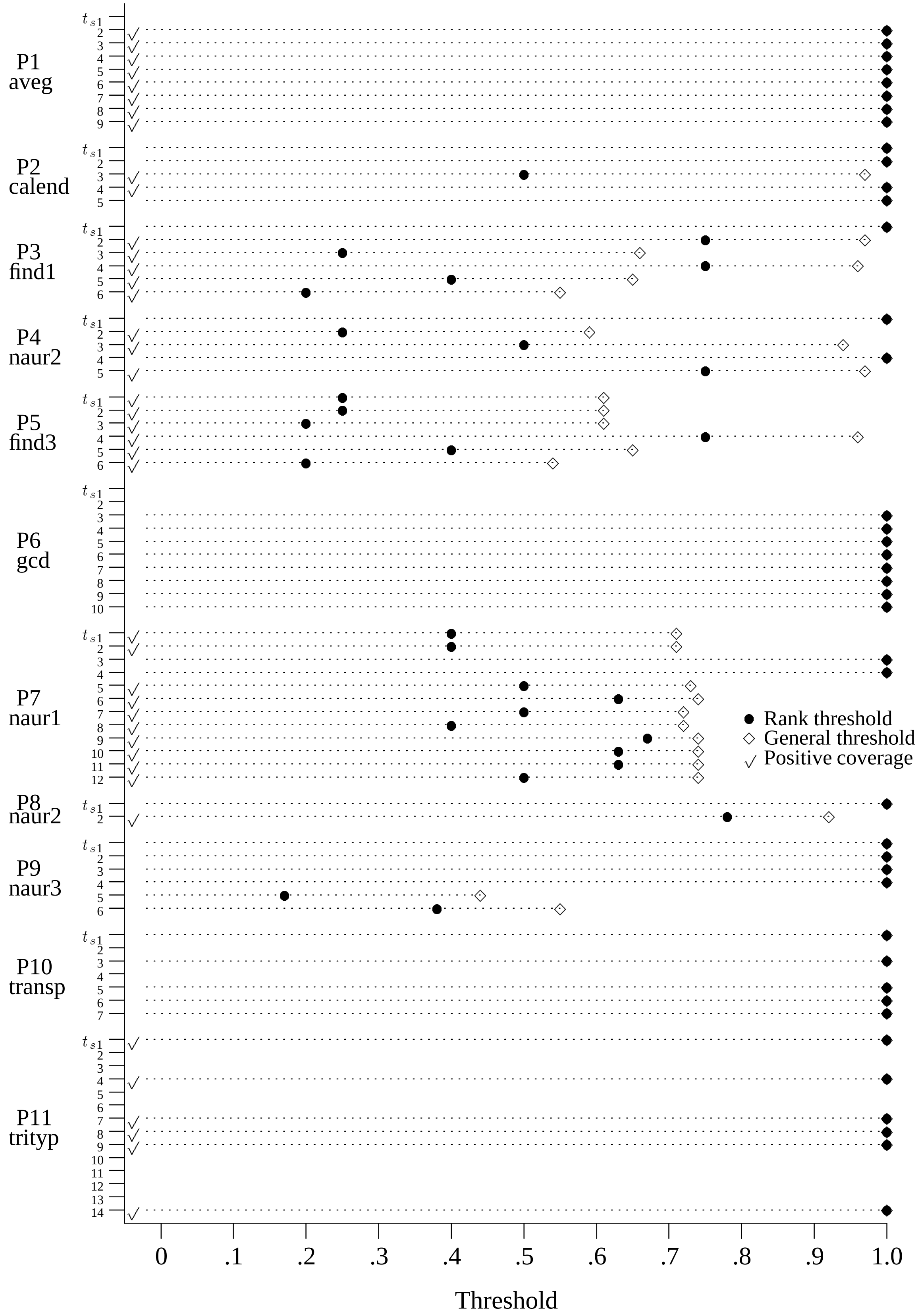


Figure 6.9 Effectiveness comparison between rank and general threshold for Heuristic 16.

Table 6.7 Coverage Analysis for Critical Slicing. $|T_f|$ and #FV represent the number of error–revealing (failure) test cases and variables involved in the failure, respectively. In each entry, $\surd$ indicates a positive result and $\times$ indicates a negative one.

Program	$\lvert T_f \rvert$	#FV	t_{f_1}	t_{f_2}	t_{f_3}	t_{f_4}	t_{f_5}	t_{f_6}	t_{f_7}	t_{f_8}	t_{f_9}
P1: aveg	6	2	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$			
P2: calend	3	1	$\surd$	$\surd$	$\surd$						
P3: find1	3	1	$\surd$	$\times$	$\surd$						
P4: find2	3	1	$\surd$	$\surd$	$\surd$						
P5: find3	4	1	$\surd$	$\surd$	$\surd$	$\surd$					
P6: gcd	9	2	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
P7: naur1	2	1	$\surd$	$\surd$							
P8: naur2	3	1	$\surd$	$\surd$	$\surd$						
P9: naur3	7	1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$		
P10: transp	4	2	$\surd$	$\surd$	$\surd$	$\surd$					
P11: trityp	3	1	$\surd$	$\surd$	$\surd$						

% 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%

the execution of P3 without the faulty statement against t_{f_2} generates 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_2} 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 expanded dynamic program slices, except for P9 with a missing statement. This supports our claim in Chapter 5.1.2.3 that expanded dynamic slices are used for fault localization when the corresponding critical slices fail to include the faulty statements.

6.2.4.2 Effectiveness Comparison

Figure 6.10 presents the degree of reduction for the size of a critical slice being refined from the corresponding exact dynamic slice (DPS) and expand dynamic slice (EDPS).

The $\mathcal{R}_{cs}$ of all critical slices range from 0.38 to 0.90 with medium 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 expanded dynamic slices because DPS is a subset of EDPS as indicated in Chapter 5.1.2. In addition, there is no superset/subset relationship between CS and DPS, because of the incomparability between CS and DPS as mentioned in Chapter 5.1.2.1, Thus, the value of $\mathcal{R}_d$, 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. $\mathcal{R}_d$ of the program “P10:transp” in the figure demonstrates this special case (i.e., all four $\mathcal{R}_d$ with negative value) which often happens for programs with heavy array and pointer reference.

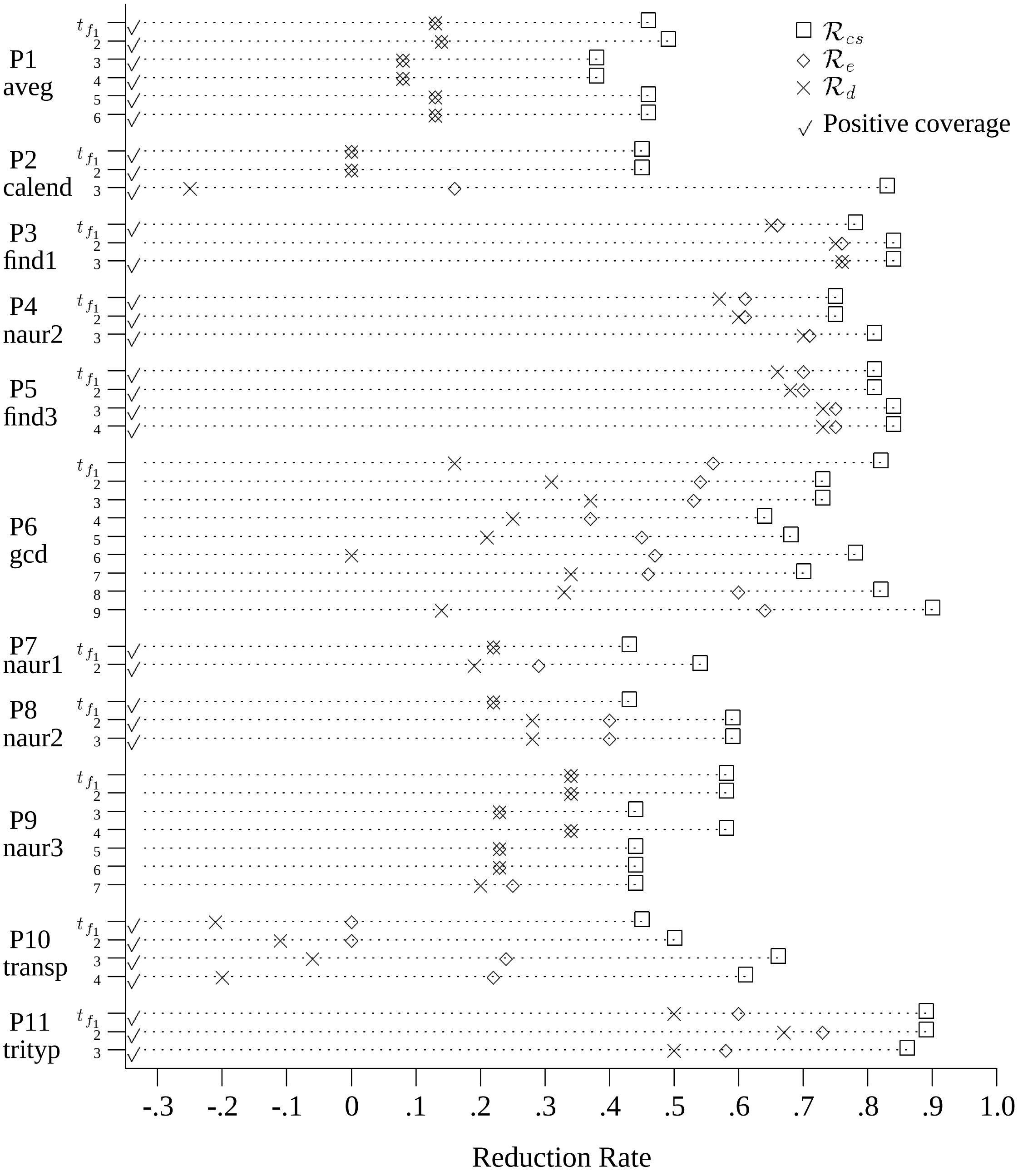


Figure 6.10 Effectiveness comparison of critical slices.

In this circumstance, excluding a statement with array/pointer reference will have serious side-effects and let the statement be easily included into the corresponding critical slice. We also notice that statements in a critical slice of P10 may not be in the corresponding expanded dynamic slice because of the array/pointer reference as indicated in Chapter 5.1.2.

In our experimental results, $\mathcal{R}_e$ ranges from 0.0 to 0.76 with medium 0.37, mean 0.366, and standard deviation 0.247. Meanwhile, $\mathcal{R}_d$ ranges from -0.25 to 0.76 with medium 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 EDPS and DPS, respectively.

Our experiment indicates that Critical Slicing (CS) not only has the power of containing faulty statements as EDPS 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 heavy reference of arrays or pointers, the size of a critical slice is smaller than the size of the corresponding exact dynamic slice, and most of the $\mathcal{R}_d$ in Figure 5.2 will be greater than 0.2.

6.3 Summary

Heuristics for fault localization have been integrated into the prototype debugging tool SPYDER. This version of SPYDER will suggest confined search domains to let users continue the debugging process within the scope of the domain. SPYDER was used for our preliminary experiment. The results provide evidence to support the effectiveness of the proposed heuristics and Critical Slicing for fault localization.

A graphic summary of our experimental results is presented in Figures 6.11 and 6.12. Figure 6.11 illustrates the reduction rate from the whole program to statements highlighted by the heuristics proposed in Chapter 4 ($\mathcal{R}_a$) or to a critical slice ($\mathcal{R}_{cs}$) for each tested programs with positive coverage analysis. Figure 6.12 demonstrates the reduction rate from the base domain of our approach (e.g., H1), $\mathcal{R}_h$ and $\mathcal{R}_d$. Both figures give us an overall picture about the distribution of our experimental results based on the heuristics

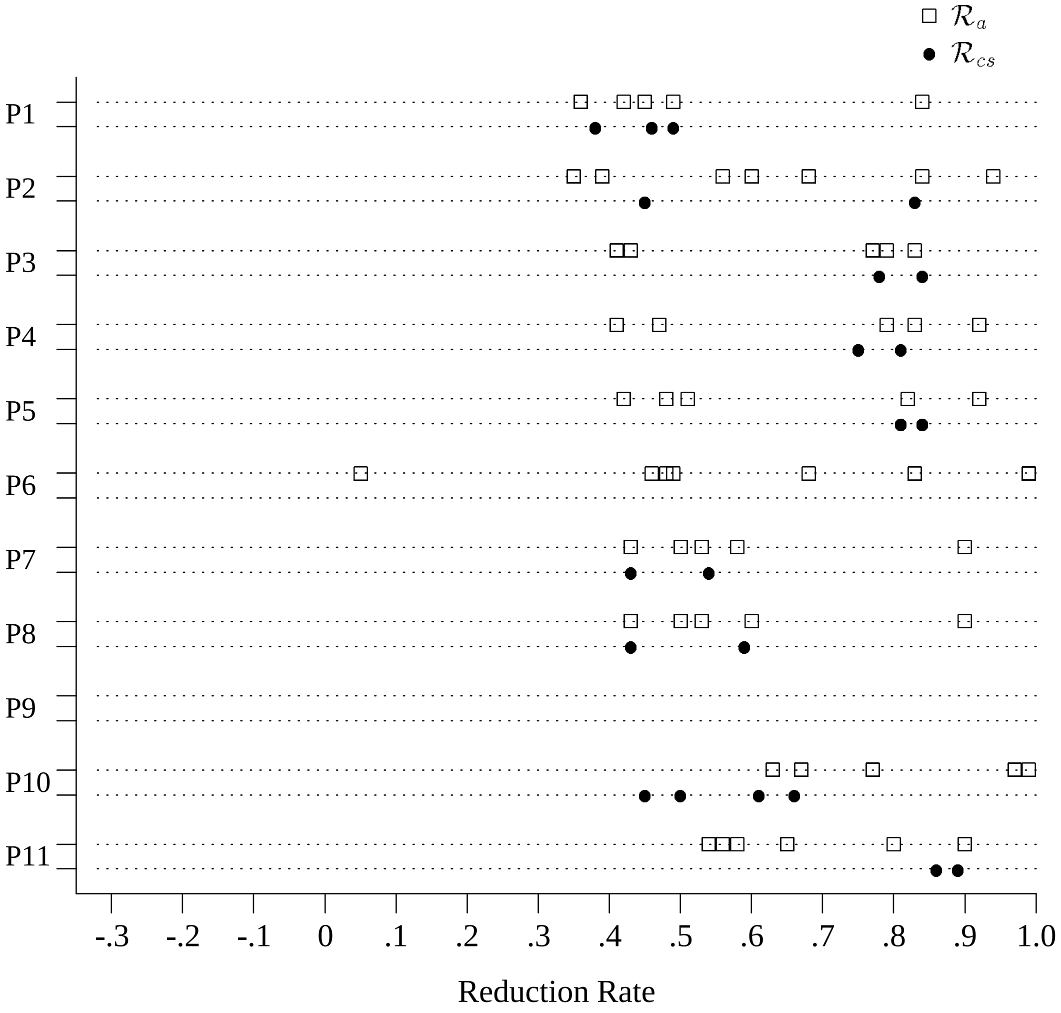


Figure 6.11 A graphic summary of the effectiveness comparison between $\mathcal{R}_a$ (the heuristics proposed in Chapter 4) and $\mathcal{R}_{cs}$ (Critical Slicing) for all tested programs with positive coverage analysis.

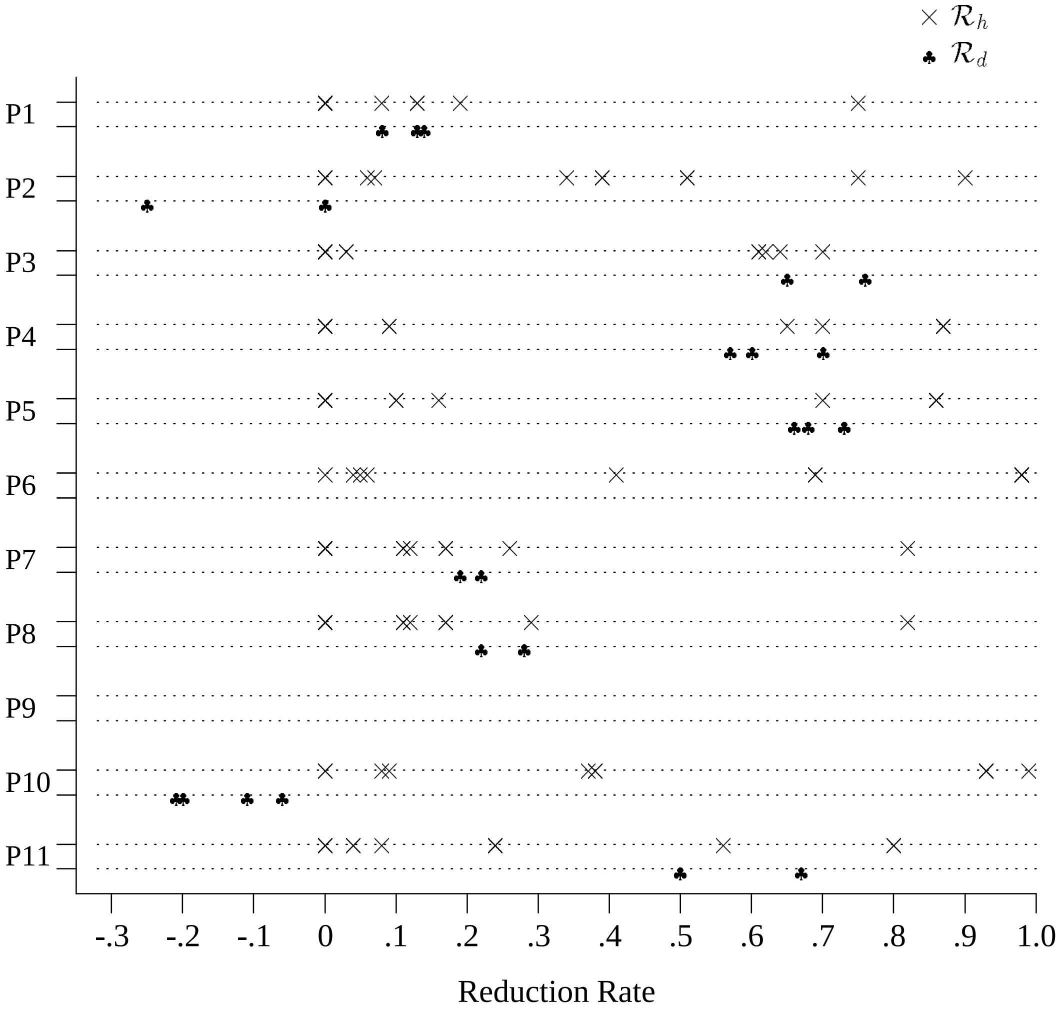


Figure 6.12 A graphic summary of the effectiveness comparison between $\mathcal{R}_h$ (the heuristics proposed in Chapter 4) and $\mathcal{R}_d$ (Critical Slicing) for all tested programs with positive coverage analysis.

and the Critical Slicing approaches. The reduction rate of Critical Slicing for each tested program is more stable than those of the heuristics, and is within a smaller range.

Valuable experience and suggestions are derived from this experiment. For the heuristics of Chapter 4, the 43% average reduction rate from the whole program as the base scope (i.e., region suggested by H1) for the heuristics family tree is a significant improvement for reducing the search space. In addition, some heuristics (e.g., H9 and H10) can handle certain fault types with greater than 90% reduction rate from the base scope of H1, although not often. Generally speaking, heuristics with threshold requirements are more effective than others. To set the rank threshold with 75% or 50% as the standard value for the first guess is an efficient way to start employing the heuristics with threshold requirements.

On the whole, we suggest employing the heuristics in the following order:

1. Invoke H9/H10, which provides a very small region easy to check although often empty.

2. Apply heuristics with threshold requirements by varying thresholds for different purposes after the standard one is used for the first try.

3. Apply other heuristics and extend the search domain to the basis suggested by H1, if needed.

By uniting the heuristics and Critical Slicing, we expect the power of fault localization to surpass that of any existing debugging tools. Algorithms for applying the heuristics and the Critical Slicing as one fault localization process are presented in Appendix E. In short, the high positive coverage percentage of the heuristics and Critical Slicing as well as the significant reduction of the confined search domain confirms the effectiveness of the proposed fault localization techniques.

7. CONCLUSIONS 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 dissertation, we have recognized that designating suspicious statements related to program failures is an important step done by experienced programmers to locate faults, and 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 chapter, we summarize the experimental results of our approach, review the new debugging paradigm, discuss some lessons learned from the implementation, and suggest future work.

7.1 Summary of Our Approach

From our research, a list of affirmable statements about our approach is presented as follows:

- Results with a high percentage of positive coverage analysis assure that the reduced search domain suggested by our fault localization techniques often contains faulty statements and will not mislead users in further debugging. Moreover, if our heuristics are based on basic blocks instead of statements, then we get positive coverage analysis for all heuristics as long as the reduced domain is not empty.

- From the overall point of view, our proposed heuristics (the basis set, H1) average 43% reduction from the whole program for programs in our study. Other heuristics will further refine the search domain. This is a significant improvement on fault localization, and we believe our approach can be generalized to real programs.

- Heuristics 9 and 10, which deal with statements in the difference between the failure set and the success set, are extremely effective in identifying a faulty statement with wrong variable initialization by indicating a region with 99% reduction from the whole program as well as positive coverage analysis for programs in our study. We suggest employing these two heuristics first.

- Heuristics with threshold requirements (i.e., H3, H4, H7, and H13) are the most effective ones in the heuristic family. From our experiments, we decided to use the rank threshold with the first 75% of ranked levels as a standard threshold for the first try while employing these heuristics.

- Heuristics under H2 (success set) without threshold requirements as presented in Table 6.6 are not powerful enough in both coverage analysis and effective comparison. We can ignore them at the beginning.

- For local analysis based on the influency frequency, we suggest employing H15 rather than H16.

- Critical Slicing is a low cost approach with high percentage of positive coverage analysis as well as significant reduction from the whole program for reducing search domain. The average reduction rate from the whole program for programs in our study is around 64%. We strongly recommend this approach for fault localization, especially when mutation–based testing is used during the testing phase. In this case, we get critical slices at no additional cost.

7.2 A New Debugging Paradigm

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.

7.3 Limitation of the Paradigm

As suggested by the proposed model in Chapter 3, a thorough test is preferred before the debugging process is started in order to collect enough information. We then identify the non–error–revealing test cases that are only related to program failures based on the proposed domain failure analysis for debugging purposes. Among current testing methodologies, only input domain testing and partition analysis can achieve the goal of domain failure analysis without extra effort. For other testing methodologies, special effort is needed to classify test cases, which are created to satisfy different testing criteria, into domain failure analysis according to the specification and the program behavior. Without this step, the approach is still applicable but may not be efficient enough because of those irrelevant non–error–revealing test cases.

The display of slices does not explicitly reflect the exact behavior among multiple occurrences of the same statement, i.e., statements inside a loop. For both dynamic program slicing and critical slicing, the execution of a loop is unfolded as consecutive identical blocks for analysis. The effect of each iteration is passed to the next iteration. Slicing analysis will highlight statements based on the unfolded execution flow. A statement could be highlighted because of the effect of one iteration only. Nevertheless, the display of slices cannot distinguish whether highlighted statements in the loop are referred to once or multiple times. Extra effort is needed to figure out the exact behavior in this scenario.

7.4 Lessons Learned from the Prototype Implementation

In this section, we focus on the implementation of fault localization techniques. As mentioned in Chapter 6.1.3, a given program is converted to a program dependency graph for internal analysis and usage. The basic unit of a program dependency graph is a vertex that represents a simple statement or a predicate where a simple statement is defined as a statement with one memory modification (e.g., an assignment statement $a = b + c$;). On the other hand, a statement is the basic presentation unit of a program for testing and debugging purposes.

We decided to implement heuristics based on the vertices in a program dependency graph to take advantage of the internal implementation techniques of program slicing. Detailed discussion is in Chapter 6.1.3. In this case, the consistency between internal reference (vertices) and external display (statements) must be considered, and should be handled at an early stage in order to store the necessary information. Instead of the whole statement being highlighted while a slice is displayed, only part of the statement which corresponds to a vertex of the slice is highlighted.

There might be a problem when we try to integrate mutation–based testing with SPYDER. A mutant is a slight change of the original program. If the mutant has the same program dependency graph as the original one, then we can reuse the original one to get dynamic slices with respect to the new program parameter. Nevertheless, the execution history and dynamic program dependency graph could be different from the original one. If a mutant changes the variable reference, then the whole dependency graph could be changed. An efficient way to construct, save, and retrieve the dependency graph (static and especially the dynamic) of mutants is needed to reduce the overload. The compiler integrated testing to support program mutation [Kra91] is a possible avenue of research to resolve the problem.

7.5 Future Work

In this section, new directions of this research are discussed.

7.5.1 Fault Guessing Heuristics

The reduced search domain suggested by the proposed fault localization techniques can be immediately used as the working scope to locate faults. Moreover, we believe this region contains valuable information that can be used to predict possible fault types and locations. To develop a set of fault guessing heuristics for identifying possible fault types and locations by analyzing testing criteria in this region is promising future research. Other white-box testing methodologies, especially data flow testing, should be re-examined for this purpose.

7.5.2 Exhaustive Experiments

The experiment in this dissertation 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 and also indicate the situations which can best be handled by each heuristic. 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.

7.5.3 Failure Analysis

The failure and fault analysis mentioned in Chapter 3 is worth completing. Because of the variety of software applications, a generic classification of failure modes is not feasible. Focusing on specific applications will make it easier for us to identify various failure modes. A study of real world program failures and faults must be conducted. Then, the possible cause and effect relationships between failures and faults should be examined. By analyzing the relationships and other information, we may be able to develop another set of heuristics to locate faults more efficiently. In addition, the classification of failure modes can be used in other areas such as software reliability and system integration testing.

7.5.4 Large Scale Programs

For debugging programs with a large number of statements and many function calls, it is not practical to consider all statements at one time. A feasible way is to first identify the functions (or models) with abnormal behavior, and then we can construct a dependency graph for the large program by using functions/models as the basic nodes. The graph should carry more information about the interface among functions and will be a superset of a corresponding simple call graph. Then, the principle of our approach can be extended to localizing suspicious nodes in the dependency graph. After reducing the search domain of possibly faulty functions, we then employ our approach again to localize faulty statements. This kind of top–down method helps us debug large scale programs more efficiently.

7.5.5 Other Applications

As mentioned in Chapter 2, we have considered only structured programs written in C or Pascal in this research. An immediate extension is to examine how the proposed approaches can be applied to programs for different environments, such as parallel and distributed programs, as well as programs written in other languages, such as object-oriented languages and functional languages. In order to construct more powerful techniques for debugging, testing methodologies on the above domains should also be explored.

Another possible application of this approach is for software maintenance and regression testing. Because the dynamic information of the given program has been recorded and analyzed in our approach, only the parts related to the faults fixed should be retested after the debugging process. With the dynamic information and the testing criteria related to faults, we can designate the testing criteria needed to be satisfied when the program is retested to assure that the old program failures have been eliminated and no new faults introduced.

BIBLIOGRAPHY

BIBLIOGRAPHY

[AC73] W. Amory and J.A. Clapp. *A Software Error Classification Methodology, MTR-2648, Vol. VII.* Mitre Corp., Bedford, Massachusetts, 30 June 1973. (Also published as RADC-TR-74-324, Vol. VII).

[ADH$^+$89] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. H. Spafford. Design of mutant operators for the C programming language. Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana, March 1989.

[ADS91a] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the 1991 Symposium on Software Testing, Analysis, and Verification (TAV4)*, pages 60–73, Victoria, British Columbia, Canada, October 8–10 1991.

[ADS91b] H. Agrawal, R. A. DeMillo, and E. H. Spafford. An execution backtracking approach to program debugging. *IEEE Software*, 8(3):21–26, May 1991.

[ADS93] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, 23(6):589–616, June 1993.

[AFC91] Keijiro Araki, Zengo Furukawa, and Jingde Cheng. A general framework for debugging. *IEEE Software*, 8(3):14–20, May 1991.

[Agr91] 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).

[AH90] 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).

[AL80] Anne Adam and Jean-Pierre Laurent. LAURA, a system to debug student programs. *Artificial Intelligence*, 15(1,2):75–122, November 1980.

[AM86] Evan Adams and Steven S. Muchnick. Dbxtool: A window–based symbolic debugger for Sun workstations. *Software–Practice and Experience*, 16(7):653–669, July 1986.

[ANS83] ANSI/IEEE. *IEEE Standard Glossary of Software Enginnering Terminology.* IEEE Std 729–1983. IEEE, New York, 1983.

[AS88] 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.

[AS89] H. Agrawal and E. H. Spafford. A bibliography on debugging and backtracking. *ACM Software Engineering Notes*, 14(2):49–56, April 1989.

[Bal69] R. M. Balzer. EXDAMS: Extendible debugging and monitoring system. In *AFIPS Proceedings, Spring Joint Computer Conference, vol. 34*, pages 567–580, Montvale, New Jersey, 1969. AFIPS Press.

[BDLS80] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Theoretical and empirical studies of using program mutation to test the functional correctness of programs. In *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, pages 220–233, Las Vegas, January 1980.

[Bea83] Bert Beander. VAX DEBUG: An interactive, symbolic, multilingual debugger. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 173–179, Pacific Grove, California, March 1983. (*ACM Software Engineering Notes*, 8(4), August 1983; *ACM SIGPLAN Notices*, 18(8), August 1983).

[BEL75] 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).

[Boe81] B. W. Boehm. *Software Enginnering Economics*. Prentice–Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[Bow80] John B. Bowen. Standard error classification to support software reliability assessment. In *AFIPS Proceedings of 1980 National Computer Conference, Vol. 49*, pages 697–705, Anaheim, CA, May 19–22 1980.

[BP84] Victor R. Basili and Barry T. Perricone. Software errors and complexity: An empirical investigation. *Communications of the ACM*, 27(1):42–52, January 1984.

[Bra70] Gordon H. Bradley. Algorithm and bound for the greatest common divisor of n integers. *Communications of the ACM*, 13(7):433–436, July 1970.

[Bud80] T. A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven, Connecticut, 1980.

[CC87] 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.

[CDK$^+$89] B. J. Choi, R. A. DeMillo, E. W. Krauser, R. J. Martin, A. P. Mathur, A. J. Offutt, H. Pan, and E. H. Spafford. The Mothra tool set. In *Proceedings of the 22nd Annual Hawaii International Conference on Systems Sciences*, pages 275–284, Kona, Hawaii, January 1989.

[Cho90] Byoungju Choi. *Software Testing Using High Performance Computers*. PhD thesis, Purdue University, West Lafayette, Indiana, December 1990.

[CHT79] Thomas E. Cheatham, Glenn H. Holloway, and Judy A. Townley. Symbolic evaluation and the analysis of programs. *IEEE Transactions on Software Engineering*, SE-5(4):402–417, July 1979.

[CPRZ89] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, SE-15(11):1318–1332, November 1989.

[CR81] Lori A. Clarke and Debra J. Richardson. Symbolic evaluation methods — implementations and applications. In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing*, pages 65–102. North–Holland Publishing Co., 1981.

[CR83] 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).

[DE88] 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.

[Deu79] M. Deutsch. Verification and validation. In R. W. Jensen and C. C. Tonies, editors, *Software Engineering*, pages 329–408. Prentice–Hall, Inc., Englewood Cliffs, New Jersey, 1979.

[DGK$^+$88] R. A. DeMillo, D. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. In *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, pages 142–151, Banff, Canada, July 1988.

[DLS78] 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.

[DO91] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, SE-17(9):900–910, September 1991.

[Dun86] 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.

[Fer87] G. Ferrand. Error diagnosis in logic programming, an adaptation of E. Y. Shapiro's method. *Journal of Logic Programming*, 4(3):177–198, September 1987.

[FGKS91] 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.

[FW88] Phyllis G. Frankl and Elaine J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, SE-14(10):1483–1498, October 1988.

[FW91] 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 1991 Symposium on Software Testing, Analysis and Verification (TAV4)*, pages 154–164, Victoria, British Columbia, Canada, October 8–10 1991.

[GD74] John D. Gould and Paul Drongowski. An exploratory study of computer program debugging. *Human Factors*, 16(3):258–277, May-June 1974.

[Gel78] M. Geller. Test data as an aid in proving program correctness. *Communications of the ACM*, 21(5):368–375, May 1978.

[GG75] 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.

[Gou75] John D. Gould. Some psychological evidence on how people debug computer programs. *International Journal of Man–Machine Studies*, 7(2):151–182, March 1975.

[Gus78] F. Gustavson. Remark on algorithm 408. *ACM Transactions on Mathematical Software*, 4:295, 1978.

[Har83] Mehdi T. Harandi. Knowledge–based program debugging: A heuristic model. In *Softfair Proceedings: A Conference on Software Development Tools, Techniques, and Alternatives*, pages 282–288, Los Alamitos, California, July 1983. IEEE CS Press.

[HL91] J. R. Horgan and S. London. Data flow coverage and the C language. In *Proceedings of the 1991 Symposium on Software Testing, Analysis, and Verification (TAV4)*, pages 87–97, Victoria, British Columbia, Canada, October 8–10 1991.

[Hoa61] C. Hoare. Algorithm 65: FIND. *Communications of the ACM*, 4(1):321, April 1961.

[How77] William E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, SE-3(4):266–278, July 1977.

[How82] William E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, SE-8(4):371–379, July 1982.

[HR83] T. Haerder and A. Reuter. Principles of transaction–oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkeley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990. (A preliminary version appeared in the *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 35–46, Atlanta, Georgia, June 22-24, 1988. *ACM SIGPLAN Notices*, 23(7), July 1988).

[HT90] Dick Hamlet and Ross Taylor. Partition analysis does not inspire confidence. *IEEE Transactions on Software Engineering*, SE-16(12):1402–1411, December 1990. (An early version presented at *Proceedings of the Second Workshop on Software Testing, Analysis, and Verification* (TAV2), pages 206–215, Banff, Canada, July 1988).

[Joh83] Mark Scott Johnson, editor. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, Pacific Grove, California, March 1983. (*ACM Software Engineering Notes*, 8(4), August 1983; *ACM SIGPLAN Notices*, 18(8), August 1983).

[JS84] W. Lewis Johnson and Elliot Soloway. Intention–based diagnosis of programming errors. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-84*, pages 162–168, Austin, Texas, August 6–10 1984. American Association of Artificial Intelligence.

[JS85] W. Lewis Johnson and Elliot Soloway. PROUST: Knowledge–based program understanding. *IEEE Transactions on Software Engineering*, SE-11(3):267–275, March 1985.

[JSCD83] W. Lewis Johnson, Elliot Soloway, Benjamin Cutler, and Steven Draper. Bug catalogue: I. Technical Report YaleU/CSD/RR #286, Department of Computer Science, Yale University, New Haven, Connecticut, October 1983.

[Kat79] H. Katsoff. SDB: A symbolic debugger. In *UNIX Programmers Manual*. AT&T Bell Laboratories, Murray Hill, New Jersey, 1979.

[KL88a] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 26 1988.

[KL88b] 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.

[KL90] Bogdan Korel and Janusz Laski. Dynamic slicing of computer programs. *The Journal of Systems and Software*, 13(3):187–195, November 1990.

[KL91] 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.

[Knu89] D. E. Knuth. The errors of TEX. *Software Practice and Experience*, 19(7):607–685, July 1989.

[KO91] K. N. King and A. Jefferson Offutt. A fortran language system for mutation–based software testing. *Software Practice and Experience*, 21(7):685–718, July 1991.

[Kor86] Bogdan Korel. *Dependence–Based Modeling in the Automation of Error Localization in Computer Programs*. PhD thesis, Oakland University, Rochester, Michigan, 1986.

[Kor88] Bogdan Korel. PELAS – program error-locating assistant system. *IEEE Transactions on Software Engineering*, SE-14(9):1253–1260, September 1988.

[Kra91] Edward W. Krauser, Jr. *Compiler–Integrated Software Testing*. PhD thesis, Purdue University, West Lafayette, Indiana, December 1991. (Also released as Technical Report SERC-TR-118-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana, July 1992).

[Las90] Janusz Laski. Data flow testing in STAD. *The Journal of Systems and Software*, 12(1):3–14, April 1990.

[Lau79] Soren Lauesen. Debugging techniques. *Software–Practice and Experience*, 9(1):51–63, January 1979.

[Lip79] Myron Lipow. Prediction of software failures. *The Journal of Systems and Software*, 1(1):71–75, 1979.

[LK83] Janusz Laski and Bogdan Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, SE-9(3):347–354, May 1983.

[Luk80] F. J. Lukey. Understanding and debugging programs. *International Journal of Man–Machine Studies*, 12(2):189–202, February 1980.

[LW87] 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.

[Lyl84] James R. Lyle. *Evaluating Variations on Program Slicing for Debugging.* PhD thesis, University of Maryland, College Park, Maryland, December 1984.

[MB79] J. F. Maranzano and S. R. Bourne. A tutorial introduction to ADB. In *UNIX Programmers Manual.* AT&T Bell Laboratories, Murray Hill, New Jersey, 1979.

[McN71] J. M. McNamee. Algorithm 408: A sparse matrix package (part I) [f4]. *Communications of the ACM*, 14(4):265–273, April 1971.

[MM83] J. Martin and C. McClure. *Software Maintenance – The Problem and Its Solution*. Prentice–Hall, Inc., Englewood Cliffs, New Jersey, 1983.

[Mor90] Larry J. Morell. A theory of fault–based testing. *IEEE Transactions on Software Engineering*, SE-16(8):844–857, August 1990.

[Mur85] William R. Murray. Heuristic and formal methods in automatic program debugging. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence, IJCAI-85*, pages 15–19, Los Angeles, California, August 1985.

[Mur86a] William R. Murray. *Automatic Program Debugging for Intelligent Tutoring Systems.* PhD thesis, The University of Texas at Austin, Austin, Texas, June 1986. (Also released as Technical Report AI TR 86-27, AI Laboratory, The University of Texas at Austin, Austin, Texas, June 1986).

[Mur86b] William R. Murray. TALUS: Automatic program debugging for intelligent tutoring systems. Technical Report AI TR 86–32, AI Laboratory, The University of Texas at Austin, Austin, Texas, August 1986.

[Mye78] Glenford J. Myers. A controlled experiment in program testing and code walkthrough/inspections. *Communications of the ACM*, 21(9):760–768, September 1978.

[Mye79] Glenford J. Myers. *The Art of Software Testing.* John Wiley & Sons, Inc., New York, 1979.

[Nau69] P. Naur. Programming by action clusters. *BIT*, 9:250–258, 1969.

[Nta88] Simeon C. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, SE-14(6):868–874, June 1988.

[Off88] A. J. Offutt. *Automatic Test Data Generation.* PhD thesis, Department of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia, 1988.

[Ost84] 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.

[OW84] Thomas J. Ostrand and Elaine J. Weyuker. Collecting and categorizing software error data in an industrial environment. *The Journal of Systems and Software*, 4(4):289–300, November 1984.

[Pan91] 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.

[PC90] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, SE-16(9):965–979, September 1990.

[Per86] L. M. Pereira. Rational debugging in logic programming. In *Proceedings of the 3rd Logic Programming Conference*, pages 203–210, London, England, July 1986.

[Pre87] Roger S. Pressman. *Software Enginnering: A Practitioner's Approach*. McGraw–Hill, Inc., New York, second edition, 1987.

[PS93] Hsin Pan and Eugene. H. Spafford. Fault localization methods for software debugging. *Journal of Computer and Software Engineering*, 1993. (to appear. A preliminary version appeared in *Proceedings of the 10th Pacific Northwest Software Quality Conference*, pages 192–209, Portland, Oregon, October 19–21 1992).

[Ran75] Brian Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[RC85] Debra J. Richardson and Lori A. Clarke. Partition analysis: A method combining testing and verification. *IEEE Transactions on Software Engineering*, SE-11(12):1477–1490, December 1985. (An early version presented at *Proceedings of the 5th International Conference on Software Engineering*, pages 244–253, San Diego, CA, May 1981).

[Ren82] 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).

[RHC76] 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.

[RW85] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[Sch71] Jacob T. Schwartz. An overview of bugs. In Randall Rustin, editor, *Debugging Techniques in Large Systems*, pages 1–16. Prentice–Hall, Inc., Englewood Cliffs, New Jersey, 1971.

[SCML79] Sylvia B. Sheppard, Bill Curtis, Phil Milliman, and Tom Love. Modern coding practices and programmer performance. *Computer*, 12(12):41–49, December 1979.

[SD60] T. G. Stockham and J. B. Dennis. FLIT: Flexowriter interrogation tape: A symbolic utility for TX-O. Memo 5001-23, Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts, July 1960.

[Sev87] Rudolph E. Seviora. Knowledge–based program debugging systems. *IEEE Software*, 4(3):20–32, May 1987.

[Sha83] Ehud Y. Shapiro. *Algorithmic Program Debugging*. The MIT Press, Cambridge, Massachusetts, 1983. (PhD thesis, Yale University, New Haven, Connecticut, 1982).

[SPL$^+$85] James C. Spohrer, Edgar Pope, Michael Lipman, Warren Sack, Scott Freiman, David Littman, Lewis Johnson, and Elliot Soloway. Bug catalogue: II, III, IV. Technical Report YaleU/CSD/RR #386, Department of Computer Science, Yale University, New Haven, Connecticut, May 1985.

[ST83] Robert L. Sedlmeyer and William B. Thompson. Knowledge–based fault localization in debugging. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 25–31, Pacific Grove, California, March 1983. (*ACM Software Engineering Notes*, 8(4), August 1983; *ACM SIGPLAN Notices*, 18(8), August 1983).

[Sta89] Richard M. Stallman. *GDB Manual, third edition, GDB version 3.4*. Free Software Foundation, Cambridge, Massachusetts, October 1989.

[Sta90] Richard M. Stallman. *Using and Porting GNU CC, version 1.37.* Free Software Foundation, Cambridge, Massachusetts, January 1990.

[Sto67] T. G. Stockham. Reports on the working conference on on–line debugging. *Communications of the ACM*, 10(9):590–592, September 1967.

[SW65] R. Saunders and R. Wagner. On–line debugging systems. In *Proceedings of IFIP Congress*, pages 545–546, 1965.

[Tra79] M. Tratner. A fundamental approach to debugging. *Software–Practice and Experience*, 9(2):97–99, February 1979.

[TRC93] Margaret C. Thompson, Debra J. Richardson, and Lori A. Clarke. Information flow transfer in the RELAY model. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'93)*, pages 182–192, Cambridge, Massachusetts, June 28–30 1993.

[Ver78] Joost S. M. Verhofstad. Recovery techniques for database systems. *ACM Computing Surveys*, 10(2):167–195, June 1978.

[Ves85] Iris Vessey. Expertise in debugging computer programs: A process analysis. *International Journal of Man–Machine Studies*, 23(5):459–494, November 1985.

[Vir91] Chonchanok Viravan. Fault investigation and trial. Technical Report SERC-TR-104-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana, September 1991.

[Wei71] G. Weinberg. *Psychology of Computer Programming.* Van Nostrand Reinhold Company, New York, 1971.

[Wei82] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[WJ91] Elaine J. Weyunker and Bingchiang Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, SE-17(7):703–711, July 1991. (An early version presented at *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification* (TAV3), pages 38–47, Key West, Florida, December 1989).

[WL86] 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.).

[WO80] E. J. Weyuker and T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, SE-6(3):236–246, May 1980.

APPENDICES

Appendix A: Notation and Terminology

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

T_f : a set of error–revealing test cases that detect the existence of faults in P thus causing P to fail. $|T_f|$ represents the number of test cases in T_f.

T_s : a set of non–error–revealing test cases that do not detect the existence of faults in P thus letting P have success results. $|T_s|$ represents the number of test cases in T_s.

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

L_f **(**L_s**)** : 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 affecting the value of v at location l when P is executed against test case t. The dynamic slice is either an exact dynamic program slice (DPS) or an expanded dynamic program slice (EDPS).

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

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

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

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

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

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

$\bigcap_{i=1}^{|T_f|} Dyn(P, v, l, T_{f_i})$: intersection of the dynamic slices for different test case parameters with respect to all error–revealing test cases (i.e., statements appearing in every slice of the failure set).

$\bigcap_{i=1}^{|T_s|} Dyn(P, v, l, T_{s_i})$: intersection of the dynamic slices for different test case parameters with respect to all non–error–revealing test cases (i.e., statements appearing in every slice of the success set).

Inclusion frequency of a statement : number of dynamic slices containing the statement, $\mathcal{F}_c$.

Influence frequency of a statement in $Dyn(P, v, l, t)$: number of times the statement was referred to in terms of data and control dependency in $Dyn(P, v, l, t)$, $\mathcal{F}_f$.

Some slightly different notations are also used in this study: $Dyn(P, v, L_{f_k}, t)$, $Dyn(P, v, L_{s_k}, t)$, $\bigcup_{j=1}^{|V_f|} Dyn(P, V_{f_j}, l, t)$, $\bigcup_{j=1}^{|V_s|} Dyn(P, V_{s_j}, l, t)$, $\bigcap_{j=1}^{|V_f|} Dyn(P, V_{f_j}, l, t)$, and $\bigcap_{j=1}^{|V_s|} Dyn(P, V_{s_j}, l, t)$. Their interpretations are similar to those presented above.

Appendix B: A List of Heuristics Proposed in Chapter 4

Heuristic	Description
H1(t)	$\{ \bigcup_{i=1}^{\lvert T_s \rvert} Dyn(P, v, l, T_{s_i}) \} \bigcup \{ \bigcup_{i=1}^{\lvert T_f \rvert} Dyn(P, v, l, T_{f_i}) \}$
H2(t)	$\{ \bigcup_{i=1}^{\lvert T_s \rvert} Dyn(P, v, l, T_{s_i}) \}$
H3($t, \mathcal{F}_c$)	{ statements in $\bigcup_{i=1}^{\lvert T_s \rvert} Dyn(P, v, l, T_{s_i})$ with inclusion frequency $\leq \mathcal{F}_c$ }, where $\mathcal{F}_c$ is an inclusion frequency number decided by the threshold given by users to select low inclusion frequency.
H4($t, \mathcal{F}_c$)	{ H3($t, \mathcal{F}_c$) $\bigcup$ H9(t) }
H5(t)	$\{ \bigcap_{i=1}^{\lvert T_s \rvert} Dyn(P, v, l, T_{s_i}) \}$
H6(t)	$\{ \bigcup_{i=1}^{\lvert T_f \rvert} Dyn(P, v, l, T_{f_i}) \}$
H7($t, \mathcal{F}_c$)	{ statements in $\bigcup_{i=1}^{\lvert T_f \rvert} Dyn(P, v, l, T_{f_i})$ with inclusion frequency $\geq \mathcal{F}_c$ }, where $\mathcal{F}_c$ is an inclusion frequency number decided by the threshold given by users to select high inclusion frequency.
H8(t)	$\{ \bigcap_{i=1}^{\lvert T_f \rvert} Dyn(P, v, l, T_{f_i}) \}$
H9(t)	$\{ \bigcup_{i=1}^{\lvert T_f \rvert} Dyn(P, v, l, T_{f_i}) \} - \{ \bigcup_{i=1}^{\lvert T_s \rvert} Dyn(P, v, l, T_{s_i}) \}$, i.e., { H6 – H2 }
H10(t)	$\{ \bigcap_{i=1}^{\lvert T_f \rvert} Dyn(P, v, l, T_{f_i}) \} - \{ \bigcup_{i=1}^{\lvert T_s \rvert} Dyn(P, v, l, T_{s_i}) \}$, i.e., { H8 – H2 }
H11(t)	$\{ \bigcup_{i=1}^{\lvert T_s \rvert} Dyn(P, v, l, T_{s_i}) \} - \{ \bigcup_{i=1}^{\lvert T_f \rvert} Dyn(P, v, l, T_{f_i}) \}$, i.e., { H2 – H6 }
H12(t)	$\{ \bigcap_{i=1}^{\lvert T_s \rvert} Dyn(P, v, l, T_{s_i}) \} - \{ \bigcup_{i=1}^{\lvert T_f \rvert} Dyn(P, v, l, T_{f_i}) \}$, i.e., { H5 – H6 }
H13	Indicate statements in all dynamic slices with high inclusion frequency in the failure set and low inclusion frequency in the success set.
H14	A supplemental heuristic for predicate statements.
H15 $(T_{f_i}, \mathcal{F}_f)$	{ statements in a $Dyn(P, v, l, T_{f_i})$ with influence frequency $\geq \mathcal{F}_f$ }, where $\mathcal{F}_f$ is an influence frequency number decided by the threshold given by users to select high influence frequency.
H16 $(T_{s_i}, \mathcal{F}_f)$	{ statements in a $Dyn(P, v, l, T_{s_i})$ with influence frequency $\leq \mathcal{F}_f$ }, where $\mathcal{F}_f$ is an influence frequency number decided by the threshold given by users to select low influence frequency.

Appendix C: Programs Tested

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 [Gel78], which was analyzed by Budd [Bud80], 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 [Hoa61] 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 [BEL75], DeMillo–Lipton–Sayward [DLS78], and Frankl–Weiss [FW91]. 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 [Bra70], which was also analyzed by Budd [Bud80], 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 [GG75] analyzed an erroneous text formatting program (originally by Naur [Nau69]). 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 [McN71], which was adopted for experiment by Frankl and Weiss [FW91], generates the transpose of a sparse matrix whose density does not exceed 66%. Two faults were identified in the original FORTRAN program. [Gus78] 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.[RHC76] 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 $>$).

Appendix D: Figures of Experimental Results Presented by the Order of Programs Tested

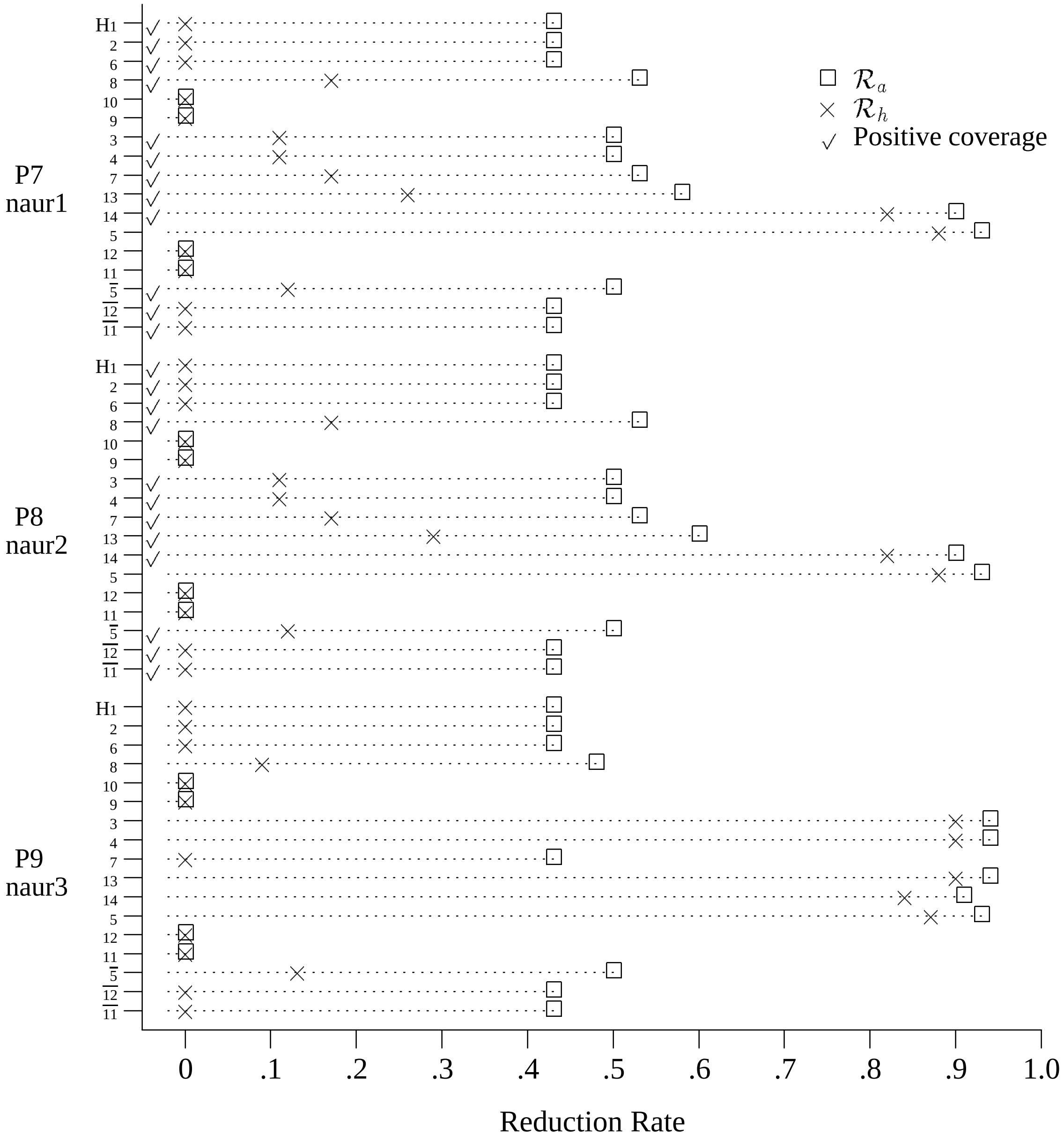


Figure D.1 Effectiveness comparison of heuristics in the order of tested programs

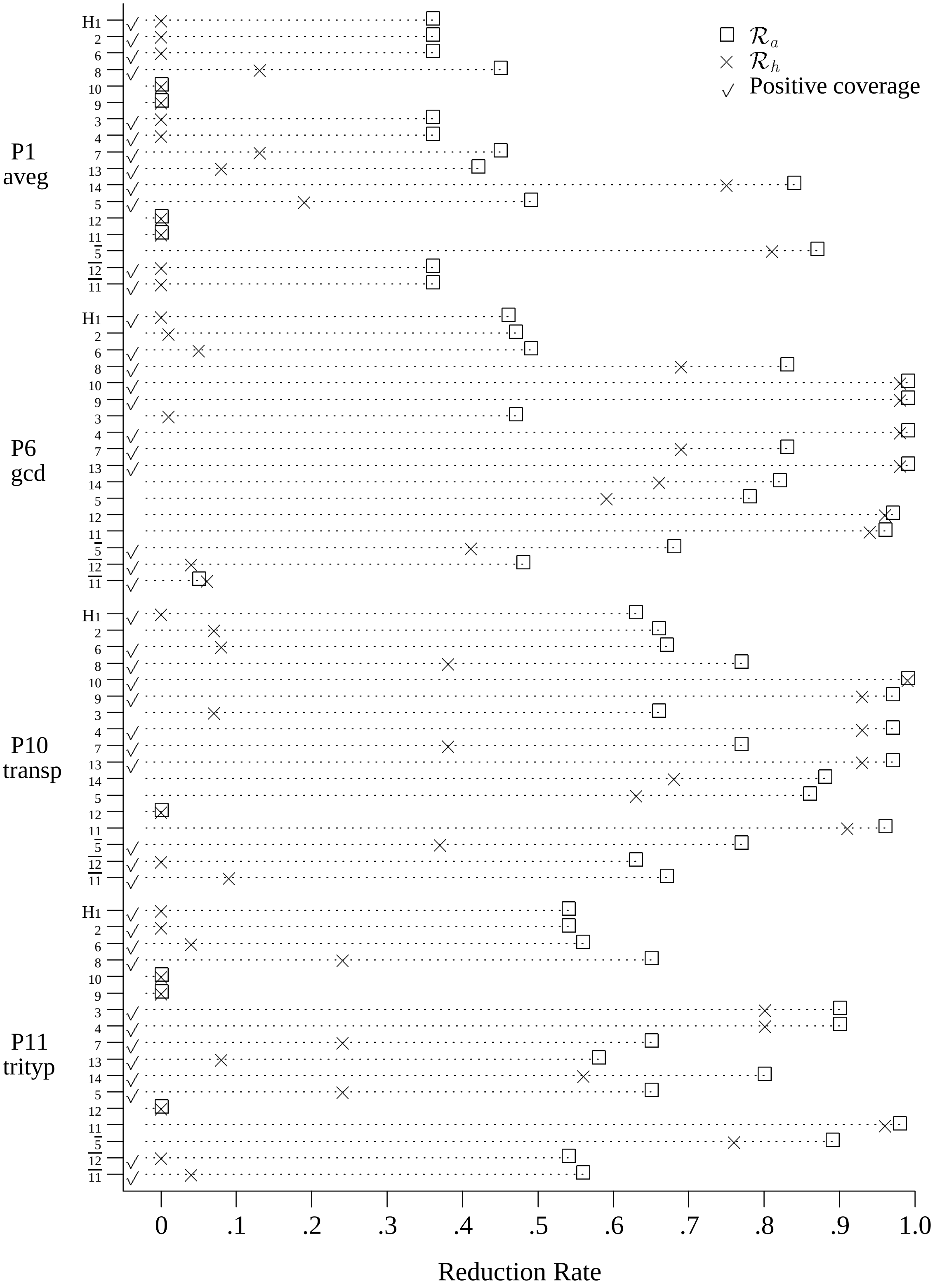


Figure D.1 Continued.

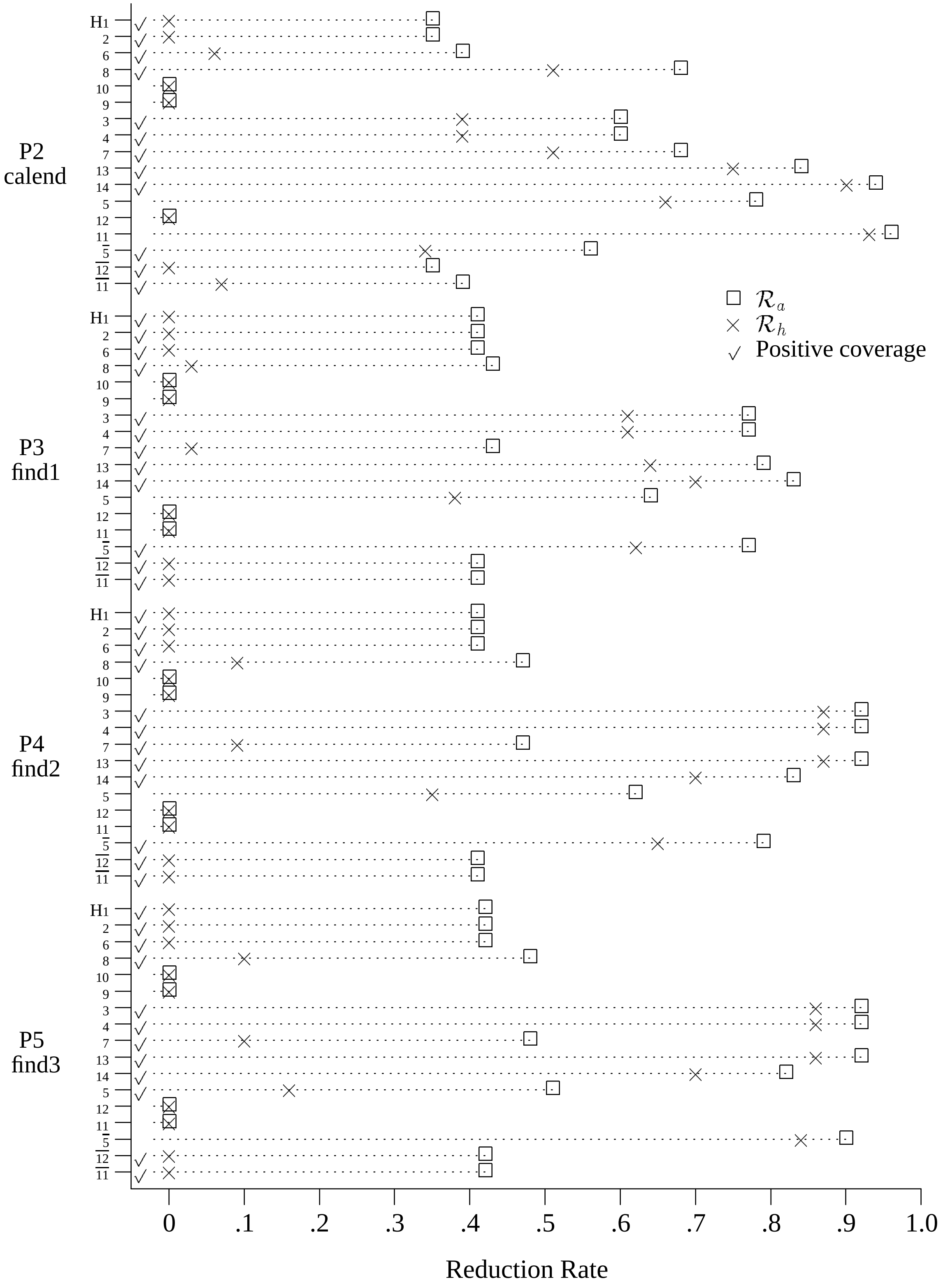


Figure D.1 Continued.

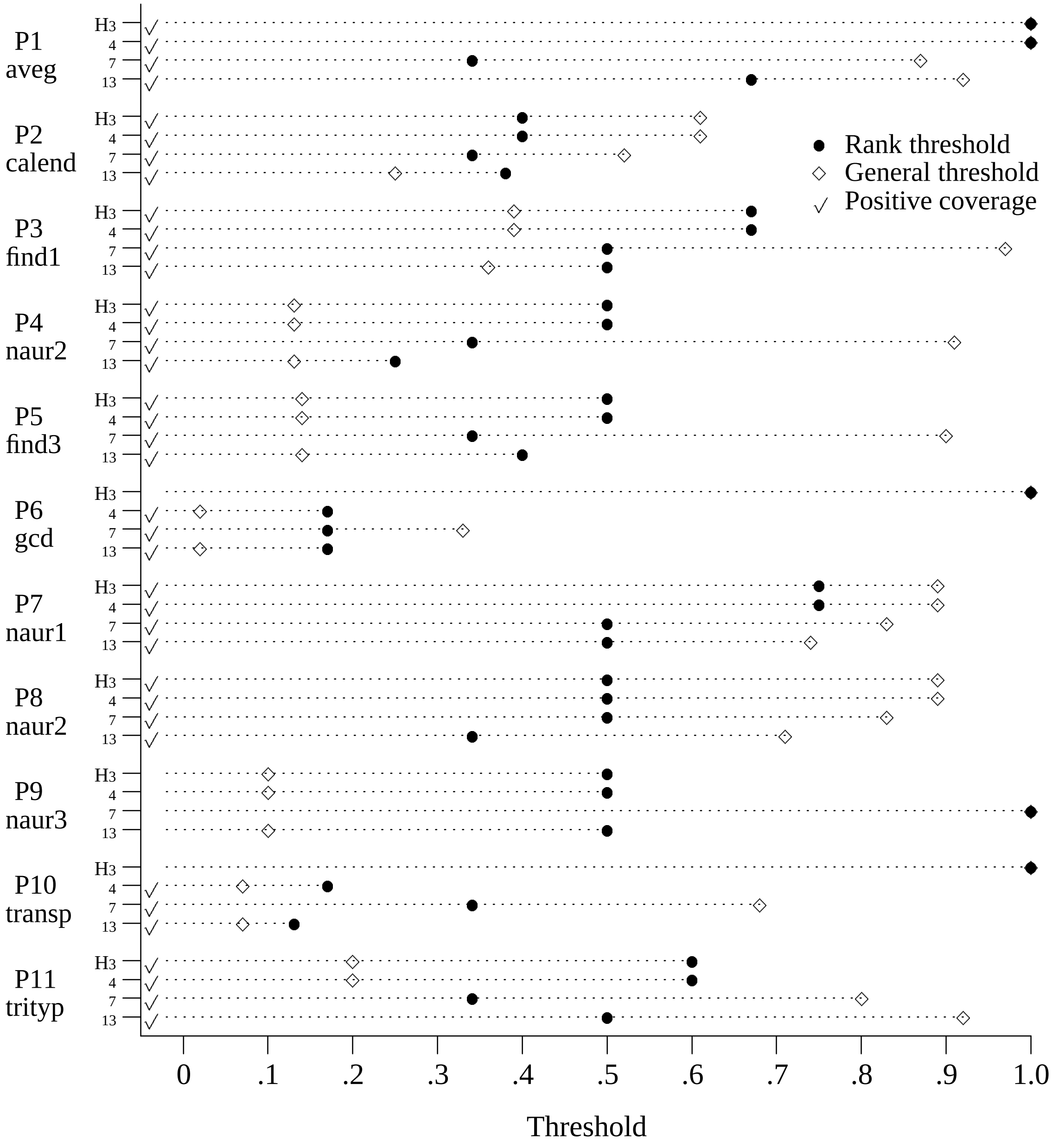


Figure D.2 Effectiveness comparison between rank and general threshold in Figure 6.7 presented by the order of tested programs.

Appendix E: Algorithms for Applying the Proposed Heuristics

```
Main    /* main procedure of the debugging mode */
 Detect program failures after thorough test.
 Analyze the failures and designate non–error–revealing test cases related to the
  failures based on the analysis in Chapter 3 for debugging purposes.
 loop
     if (the search domain is empty or a new search domain is needed)  then
         if (mutation–based testing information is needed)  then
             Quit this debugging mode temporarily, switch to the testing mode
               to conduct further testing as needed, and switch back to the
               debugging mode later.
         else
             Enter fault_localization to get a reduced search domain.
         endif
     endif
     Predict the fault type and location.
     Verify the prediction.
 until (the fault is identified or users give up).
 Fix the fault.
end Main.
```

```
Procedure fault_localization
 if (using mutation–based testing information)  then
     Enter apply_heu_ch5 to get a suggested search domain.
     if (choosing the heuristics proposed in Chapter 4 for more help)  then
         Enter apply_heu_ch4 to get another suggested search domain.
         Analyze these two search domains to obtain a new one.
     endif
 else
     Enter apply_heu_ch4 to get a suggested search domain.
 endif
 return  the current search domain.
end fault_localization.
```

```
Procedure apply_heu_ch4
Select dynamic slicing criteria such as variable, test case, location, and dynamic
  analysis technique (exact dynamic slicing or expanded dynamic slicing).
if (for global analysis) then
    if (have sufficient computing resources) then
        Invoke all heuristics, then apply the meta–heuristic that intersects
          regions suggested by Heuristics 1–4, 6–10, and 13.
        if (the suggested domain is satisfactory) then return the domain.
    endif
    Invoke H9 and H10 to provide a very small region.
    if (the suggested domain is satisfactory) then return the domain.
    switch (number of error–revealing and non–error–revealing test cases):
        case1: no non–error–revealing test cases
            Apply heuristics in the following group order: a) H8 (same as H10
              in this case); and b) H7.
        case2: few non–error–revealing but many error–revealing test cases
            Apply heuristics in the following group order: a) H8 and H13;
              b) H7; and c) H4 and H3.
        case3: similar # of non–error–revealing and error–revealing test cases
            Apply heuristics in the following group order: a) H13; b) H8;
              c) H7; and d) H4 and H3.
        case4: many non–error–revealing but few error–revealing test cases
            Apply heuristics in the following group order: a) H8 and H13;
              b) H4 and H7; and c) H3.
    endswitch
    if (the suggested domain is satisfactory) then return the domain.
    Apply the supplemental heuristic H14 based on the information from previous
      heuristics for possible faulty predicate statements.
    if (the suggested domain is satisfactory) then return the domain.
    Apply other heuristics and extend the search domain to the bottom line (H1).
    if (the suggested domain is satisfactory) then return the domain.
else /* for local analysis */
    Apply H15 or H16 for a selected test case (H15 is preferred).
    if (the suggested domain is satisfactory) then return the domain.
return an empty set.
```

```
end apply_heu_ch4.


Procedure apply_heu_ch5
 if (the information of Critical Slicing is needed) then
     Get critical slices with respect to selected error–revealing test cases.
     return the critical slices.
 else
     Follow the categories $\mathcal{MT}1$ to $\mathcal{MT}5$ to find the best fit case and get the
       suggested region.
     return the region.
end apply_heu_ch5.
```

VITA

VITA

Hsin Pan was born on September 1, 1959 in Taiwan, Republic of China. He graduated from National Taiwan University (NTU) with a Bachelor of Science degree in Computer Science in June 1982. After graduating from NTU, he worked at the Department of Computer Science and Information Engineering at NTU as a system administrator and a full time teaching assistant for two years. In September 1984, he began graduate studies at Michigan State University in East Lansing, Michigan, and received the Master of Science in Computer Science in June 1986. Then, he matriculated at Purdue University in West Lafayette, Indiana, and was awarded the degree of Doctor of Philosophy in Computer Science in August 1993.

Mr. Pan is a member of the Software Engineering Research Center at Purdue University, the IEEE Computer Society, and the ACM.