Automated Fault Localization Techniques; A Survey

Mohammad Amin Alipour

School of Electrical Engineering and Computer science
Oregon State University
alipour@eecs.oregonstate.edu

Abstract. Fault localization is a task in software debugging to identify the set of statements in a program that cause the program to fail. As size and complexity of software grows,and the more developers participate in debugging, fault localization becomes harder than before. Automated fault localization techniques aim to facilitate this task by guiding the developers to a small portion of program that are likely to be culpable for the program failures.
In this paper, we look at important techniques for automated fault localization and categorize them based on their common features. We also briefly discuss some of challenges and shortcomings in this field.

1 Introduction

Software systems fail. Bug reports are filed. Software developers request more time to investigate the problems. customers call in to complain about bugs. Project managers face such situations everyday.

Software is a complex system. Usually many pieces of code interact through interfaces and manipulate different data to accomplish a task, e.g. processing a purchase query or, rendering a LaTeX file. It requires good understanding of the specification (i.e. expected behaviors), moreover, proper abstraction and design and, proper implementation of algorithms, interfaces and data structures. An error can occur in each of this stages and it can remain unknown until it manifests in a failure, (hopefully) during testing. A software fault is the embodiment of errors in the source code, e.g. wrong initialization or operator, or missing statements or logic, etc. To fix failures, faults should be located and corrected.

Studies on software debugging[1] suggest that debugging is an iterative process that includes hypothesizing the location of faults, proposing a fix and finally validating the hypothesis by testing. Fault localization (FL) is a crucial part of software debugging. Obviously, in this process, imprecise fault localization can mislead the whole chain of locate-fix-validate and entails excessive time and effort burden on software development. On the other hand, precise and efficient fault localization can reduce the number of trial-error cycles, hence accelerate the debugging.

[1] Understanding the process of software debugging is still an active field of research, e.g. See [17, 31, 29] for more.

Fault localization is essentially a search over the space of program components (e.g. statements, variables, values, predicates) to find suspicious entities that might have participated in a program failure. It often involves inspection of numerous components and their interactions with the rest of system. In practice, there are many suspects in a program for a failure, i.e. faults.

Automated fault localization techniques attempt to assist developers by refining and reducing the search space for faults. Thus, developers need to focus on smaller number of entities. This would save time in debugging; hence it may reduce maintenance costs. Moreover, since most companies are under market pressure, they tend to release software with known and unknown bugs [32]. Automated fault localization techniques can help companies debug more known bugs before shipping their products. It would improve the dependability of software products by facilitating timely debugging.

Fault localization techniques can be classified based on approaches that they are based on. The major approaches are program slicing [46, 4, 21], spectra based fault localization [40,39,27], statistical inference [34,**?**], delta debugging [49] and model checking [**?**]. In this paper, we study techniques based on the two last approaches: delta debugging, and model checking.

Delta debugging framework is a collection of techniques for minimizing failing test cases and isolating the failure-inducing part of them. Section 6 introduces these techniques. Then it looks at a technique based on them for fault localization. Model checking is the process of searching state space of a program for violations of a given property. Such violations are returned in form of a counter-example trace. Since model checking can provide various information about program states, transitions and execution paths in the program, it can be used in fault localization. In Section **??**, we provide a brief background on model checking. Then, it outlines some techniques that analyze counter-examples for fault localization. Section 9 concludes the paper with laying out the opportunities in fault localization.

As we will see, these two approaches differ in many ways, but most significantly by the amount of information that they need/extract from the program. In this paper, each technique is followed by a discussion of its merits and shortcomings.

The rest of the paper is organized as follows. After establishing common taxonomy for the paper in Section 2, we explain some efforts which utilize program slice for fault localization in Section 3. Then, in Section 4 we describe *spectrum*-based approaches that attempt to quantify the different behaviors of program in failure and success runs. Presence of high speed networks can justify collecting information from many execution over the Internet; Section 5 outlines series of efforts to utilize this information for cooperative bug isolation. Section 6 summarizes major efforts that exploit *delta debugging* [49] to find cause of failures. In Section 7, we look at some techniques that dynamically modify a failure execution of program to spot the statements which their modifications makes the execution succeed. In 8, we point out some techniques for fault localization in model checking. Section 9 discusses the current state of the art.

```
1: BinarySearch[l_List, k_Integer,
2: low_Integer, high_Integer, f_] :=
3: Module[{mid = Floor[ (low + high)/2 ]},
4: If [low > high, Return[low - 1/2]];
5: If [f[ l[[mid]] ] == k, Return[mid]];
6: If [f[ l[[mid]] ] > k,
7: BinarySearch[l, k, 1, mid-1, f], // FAULT
8: BinarySearch[l, k, mid+1, high, f] ] ]
```

Fig. 1: A snippet of a fault Mathematica package

2 Preliminaries

In this Section, we introduce some terms that we use throughout this paper. In this paper we use the term software and program interchangeably.

Definition 1 (Error) *An error is an internal state of a program that deviates from the expected behavior of the program.*

Definition 2 (Fault/Bug/Flaw) *Fault is set of statements which their execution may perturb the state of program to error state. In other words, it may infects the state of program.*

Definition 3 (Failure) *Failure is manifestation of error to an external entity (i.e. oracle) which discern the program failure. In this case we say the program fails.*

Definition 4 (Testable Program) *A program is testable, if some error states can be manifested.*

Program failures are often observed at output of program when the the program crashes or produces wrong output. Many of error states do not manifest at output, because they may be ignored later during execution of the program, e.g. data race of similar values on a shared variable. Another reason that makes the program *un-testable* is lack of specification. For example, Figure 1 illustrates a snippet of of a flaw in Mathematica software that was not detected for five years, because there was no performance specification [1]. Line 7 in this program must be `BinarySearch[l, k, low, mid-1, f]`. This fault which turns the logarithmic algorithm to linear.

3 Program Slicing for Fault Localization

In this section we explain the fault localization techniques that are based on program slicing. A slice of program with respect to a criterion is a subset of statements in the program which faithfully reflect that criterion [46]. Throughout this sectopm, we use two program snippets (taken from [51]) depicted in Figure 2 to illustrate some of techniques.

```
 1:  read (a);
 2:  read (n);
 3:  i=0;
 4:  while (i<n) {
 5:    read (x);
 6:    read (y);
 7:    a=a/x;
 8:    b=x;
 9:    if (a>1)
10:      b=a-4;
11:    if (b>0)
12:      z=x+y;
       else
13:      z=x-y;
14:    output (z);
15:    i=i+1;}
```

Fig. 2: Running Example

For example the slice of program in Figure 2 for criterion "the statements that may influence the value variable `b` in Line 10" is: <1,2,3,4,5,7,8,15>.

Forward static slice fs of a program p regarding to a variable v in statement s is the set statements in p that *may* be influenced by the value v at s [46]. Conversely, a backward static slice of bs regarding to a variable v in statement s is set of statements that *may* affect the value of v in s.

Backward dynamic slice of a program with respect to a variable v at statement s in the execution trace $hist$ for some test-case is the set of statements in $hist$ that affect the value of v [4, 3]. Agrawal et al. argue that the majority of faults exist in the dynamic slice of failing test cases [4]. For example in Figure 2, given the input values { `a = 8, n = 1, x = 2, y = 2`}, the backward dynamic slice for `z` in Line 14 is {1,2,3,4,5,6,7,9,10,11,12}. Pan and Spaffor provide several heuristics based on dynamic slices of programs for debugging [38].

Later in [5], Agrawal et al. propose χ-slicing to narrow down the search space for faults to χ-slice of programs. A χ-slice of a program regarding to a failing test case and a passing test case contains the statements that are in failing trace but not in passing trace. χ-slicing fails whenever faults appear in both failure and successful traces.

Gyimóthy et al. propose the notion of *relevant slice* and propose a method for computation of it [21]. A relevant slice of a program regarding to a variable in a location and a test input comprises of the dynamic slice of the program, set of predicates in the dynamic slice which their different evaluation may affect the value of the variable, and statements that affect those predicates. Relevant slicing is more inclusive than subsumes dynamic slicing, thus it can include the faults that are not in dynamic slice. For example, suppose we modify Line 7 in Figure 2 to the fault `a = a/2x - 1;`. In this case, the dynamic slice of the

program regarding to variable `z` at Line 14 for input values { `a = 8, n = 1, x = 2, y = 2`} would be: {1, 2, 3, 4, 5, 6, 8, 11, 12}, which misses Line 7 where the fault resides, but relevant slice is: {1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12} and includes the fault.

Zhang et al. have evaluated the use of relevent slicing for fault localization on Siemens benchmark. They have shown that it captures all faults in single fault programs and part of faults in multiple fault programs [51]. In [52], they propose a dynamic approach to find faults that are not captured by dynamic slices.

4 Spectrum-based Fault Localization Techniques

In this section we review some of fault localization techniques that try to find the faults based on quantifying their different behaviors in passing and failing traces. These aspects of behaviors are called *spectra*.

Reps et al. have used (loop-free) path profiles (spectra) to analyze programs with potential Y2K errors[2] [40]. They compare path profiles of executions of programs with pre-2000 and post-2000 date values. Their method tries to find new paths that appears in the processing of post-2000 which are not in the pre-2000 date values and conversely paths that are missed in the execution of post-2000 values. The shortest prefixes of such paths summarize abnormal behavior of programs. For the path profiling they use the Ball and Larus's path profiling [10]. In the experiments they used few Unix utilities including `cal` and `ncftp` which have shown the approach is viable. They conjecture that there is a correlation between the path spectra and faulty programs.

In [22], Harrold et al. validate above conjecture on test. For the experimentation they used seven Siemens original subject programs and their variants (totaling 88 programs). They tested these programs under several input values and extracted different spectra information out of the executions. The experiment shows a correlation between the difference in spectra and faulty programs.

Renieris and Reis propose *nearest neighborhood* which contrasts basic block coverage of successful runs and failures to rank the suspicious blocks that may contain faults [39]. To build the profile of traces, covered basic blocks are sorted by number of times that they are executed in the trace. Then closest pair of passing trace and failing trace is selected based on the minimum difference of trace profiles. They use Ulam's distance to measure this difference. The basic blocks that are in the failing trace but not in the successful trace are reported as potential fault locations.

Jones et al. employ statement spectra to build the Tarantula system for visualizing the potential location of faults [27]. In Tarantula, a program is instrumented to compute statement coverage for several failing and succeeding test cases. Then, based on number of occurrences of a statement in successful and

[2] Y2K bug was the problem of two-digit encoding of date values in some legacy program. This encoding would have caused the dates reset to `00` in the beginning of year 2000, which in turn could cause miscalculations in those legacy systems.

failing executions, a *suspiciousness factor* (SF) for the statement is computed. The suspiciousness factor of a statement denotes the likelihood of statement to be fault. Thus, the SF's are used to rank the statements for inspection by developers. The Tarantula assigns higher SF to statements that show up more in failure traces than successful traces using Formula (1).

$$SF(s) = \frac{\%Failed(s)}{\%Failed(s) + \%Passed(s)} \tag{1}$$

In the above Formula, $\%Failed(s)$ is the ratio of number of failing traces which include statement s to total number of failing traces. Likewise. $\%Passed(s)$ is the ratio of number of successful traces to the total number of successful traces. The evaluation of Tarantula with 20 mutants of `Space` program shows it gives high suspiciousness factor to the faulty statements in majority of cases. However, in some cases that failure is due to incorrect variable initialization, it fails to identify it because all executions run these statements.

In [2], Abreu et al. propose Ochiai metric (Formula 2) to calculate the suspiciousness factor in the Tarantula framework.

$$SF(s) = \frac{Failed(s)}{\sqrt{TotalFailed * (Failed(s) + Success(s))}}, \tag{2}$$

where $Failed(s)$ is the number of failing test cases which execute s, $Success(s)$ is the number of successful test cases that execute s, and $TotalFailed$ is the number of failing test cases in the test suit. The Ochiai formula improves the accuracy of Tarantula framework on Siemens programs.

Santelices et al. compare the effectiveness of three coverage metrics: statement, branch and Def-Use(DU) pairs as spectra in Tarantula framework [41]. They introduce a mapping between branch and DU-pairs suspiciousness factors to suspiciousness of individual statements. They define the cost of fault localization as the percent of statements needs to be inspected before the fault is detected. In their subject programs, DU-paths works better than branch and branch coverage works better than statement coverage. Moreover, they show that the average of the (statement-branch-DU-pairs) entails better results on average, than each individually.

Yilmaz et al. use time spectra for fault locaization [47]. To this end, they instrument the programs to calculate the time of function calls. Then, they try to derive a model for time per caller-callee in failing and passing executions using Gaussian Mixture Models (GMMs). Afterwards, they rank the caller-callee model of failing executions based on their difference with the corresponding model in the successful run. They noticed that the number of passing test cases is important for proper modeling of time behavior of successful programs. That is , in cases with small ratio of passing test cases to failing test cases, the results were inferior to the cases with higher ratio.

Another spectra that has been used for fault localization is edge spectra that has been proposed by [53]. In this work the suspiciousness factor of each edge based is calculated and then these factors are used to calculate the suspiciousness factor of basic blocks.

5 Cooperative Bug Isolation

With prevalence of the Internet and given the fact that shipped software products often have bugs, users' running programs can cooperate in producing a large variety of information about the behavior of programs. Moreover, these techniques must keep overhead of profiling (i.e. instrumentation) as low as possible to incur acceptable performance overhead on the execution of programs on users' machines. This section surveys the techniques that tap into such information for fault localization.

Cooperative bug isolation (CBI) was introduced in [32]. CBI instruments program to extract the following predicates:

- For each assignment to variable `a`, it introduces three relational predicates (<,==,>) to all in-scope variables.
- Predicates for the return value of function calls at call-site, as of $x > 0$, $x == 0$ and $x < 0$ where x is the return value of the function call.
- Predicates for relations between same type pointers: $p == q$, $p \neq q$, $p == null$ and $p! = null$.

CBI samples predicates, to alleviate the performance overhead of predicate instrumentation, It utilizes sampling rate of $\frac{1}{a^{1000}}$, meaning that each predicate can be sampled with probability of $\frac{1}{a^{1000}}$. in the experiments which incurs only 5% performance overhead to the execution.

CBI tries to find predicates that are good predictors for program. To this end, authors classify bugs into two categories: deterministic bugs and non-deterministic bugs. A bug is deterministic with respect to a predicate P, when $P = true$ implies that program will eventually fail. If no such P exists, the bug is non-deterministic.

Deterministic bugs are determined by contrasting the predicates that only appear in failure traces. For non-deterministic bugs, CBI utilizes *logistic regression learning* method [44] for learning binary classifiers. This method assigns a value between 0 and 1 to for each predicate. This value represents the correlation of predicates with a failure. Top score predicates are reported as potential causes of the failure.

CBI framework further extended in [33] to include a richer set of predicates:

- **Branch predicates:**Two predicate for each branch which show if the branch has been taken in the execution of the program or not.
- **Return predicates:**Five predicates for each return values to monitor whether the returned value is ever < 0, ≤ 0,$= 0$, ≥ 0 and > 0.
- **Scalar-pair predicates:**For each assignment to variable `a`, six predicates (<,≤,=,≥,>, ≠) to all same-type in-scope variables and constants.

The new framework tries to find the cause of failures in presence of several bugs. Given information of failure traces, the high-level of the proposed algorithm is:

1. Find predictors of the most important bug in the failure traces.
2. Remove all failure runs that belong to the bug in Step 1 and go to Step 1.

In the new framework, the conditional probability of failure of program with respect to predicates P is computed ($Pr(Crash|P)$). The conditional probability of observing the predicate is calculated ($P(Crash|Pobserved)$), too. The value *Increase* is calculated using Formula 3. (3).

$$Increase(P) = Pr(Crash|P) - Pr(Crash|Pobserved) \tag{3}$$

For reduction of noise in the statistical reasoning, those predicates which the confidence interval of their $Increase$ is less than 95% are eliminated. Afterwards the *Importance* factor for each predicate is estimated by Formula 4:

$$Importance(P) = \frac{2}{\frac{1}{Increase(F)} + \frac{1}{log(F(P))/log(NumF)}}, \tag{4}$$

where $F(P)$ is the number of failures when predicate P is evaluated to true and $NumF$ is the total number of failures.

Importance factor of a predicate approximates how much the predicate is correlated to failure. In each iteration of the above algorithm, the predicate with highest value of *Importance* is given to programmer for debugging. Then, all traces that contain that predicate are removed and the above fault localization scheme is repeated on the remaining traces.

Thakur et al. extended the CBI infrastructure to isolate the concurrency bugs [43]. In their framework, they have added two additional predicates for each shared variable that capture the thread access patterns to shared data.

Sometimes a single predicate is an inadequate predictor for a bug. Thus, Arumuga et al. extend CBI to include compound predicates [8]. In the new framework, they only consider conjunction and disjunction between pairs of predicates. A peculiar aspect of their contribution is that instead of extra instrumentation for extracting such predicates, they approximate them from the available information of individual predicates.

Chilimbi et al. have adapted CBI framework to develop HOLMES which uses paths instead of predicates [15]. HOLMES attempts to catch acyclic, intra-procedural paths that are strong predictor of failures. The motivation is that paths provide context for failures which programmers can trace for debugging. Moreover, paths usually are available with failures for postmortem analysis, which in this case it adds no monitoring overhead on software execution. Authors also propose an adaptive process to reduce the cost of path monitoring of software which restricts the monitoring to parts of the program that are more likely to include fault. Since the path monitoring happens in a small portion of programs,HOLMES does not need to sample all paths, instead it monitors *all* paths in a small portion of program.

Adaptive HOLMES does not start monitoring until a failure occurs. It uses the formula 5 to select the functions to monitor.

$$\theta(f) = \Sigma_{i=1}^{n} \frac{1}{\Delta_i(f)} \tag{5}$$

In this formula, $\theta(f)$ is the effective score of function f which ranks functions according to the possibility of including fault. n is the number of bug reports.

$\Delta_i(f)$ is the number of functions between occurrence of f in stack trace and the location of failure.

In HOLMES, users can define a threshold for strong predictors. If the current monitored functions does not produce a sufficiently strong predictor higher than the threshold, either path monitoring is expanded to include more functions that interact with current monitored functions [7, 14] or the predictors are strengthened by coupling predictors with branches, or by coupling paths together [8].

HOLMES inspired ArumugaNainar and Liblit to reduce the cost of instrumentation with heuristics for selection of instrumentation sites in CBI [9]. In their technique when a branch predicate is found as a best predicator, its *vicinity* can potentially be good predictors, thus they are added for monitoring to reach a strong fault predictor.

Liu et al. argue that the fraction of time that a predicate at a location that evaluates to true can boost the accuracy of CBI, thus they develop SOBER [34]. SOBER utilizes the fraction of "true" observations of a predicate P within each program execution, as in formula 6, in its calculations.

$$\pi(P) = \frac{n_t}{n_f + n_t} \tag{6}$$

where n_t and n_f are the number of times the predicate P evaluates to true or false, respectively.

In SOBER, sets of failing and passing test cases are treated as statistical samples. If the density function of a predicate in failing traces differs from its density functions in passing traces, the predicate is correlated to a bug. $f(\pi(P)|\theta)$ denotes the density function of a predicate P in a sample θ.

To measure the difference of density functions, SOBER employs null test hypothesis for $f(\pi(P)|\theta_f)$ and $f(\pi(P)|\theta_p)$, where θ_f and θ_p are failing and passing samples respectively. If the likelihood of validity of this hypothesis was low, then it concludes that they are not identical. The less likelihood of the hypothesis, shows the greater difference between the density functions. Therefore, SOBER uses likelihood of validity of the hypothesis for predicates to rank them for correlation to failure.

Jiang and Su argue that giving a scenario of failure and affording the user the correlated predicates in addition to most suspicious predicates would facilitate understanding of the cause of failure, hence debugging it [25]. Thus, they use combination of feature selection and clustering to find the correlated predicates and error trace. They use SVM and Random forest techniques to find the most important fault-inducing predicates, and k-means algorithm to partition set of predicates in clusters of correlated predicates. They also devise an algorithm for creating the error trace by choosing the outcomes of branches such that they contribute to failures and assembling them. If the branch condition tends to be more in the failing trace, the branch is taken and if the negation of it appears more in the failure traces it will not be taken in the assembled trace.

6 Delta Debugging

In this section we explain two techniques that exploit *delta debugging* [49,48] for fault localization. Delta debugging (DD) contains two algorithms: simplification, and isolation.

Given a failing test input, simplification algorithm systematically reduces the size of the input by eliminating parts of it to reach the minimal input such that it cannot further reduced. The isolation algorithm constructs pair of passing and failing inputs in which difference is minimal.

Gupta et. al combine delta debugging with dynamic slicing to locate faults [20]. Their approach, first, finds the minimal distance passing and failing inputs. Then, the forward dynamic slices with respect to the difference of failing input and passing input is computed. Then the backward dynamic slice with respect to outputs are computed. The intersection of the forward and backward slices are called "failure-inducing chop" which potentially have contains fault.

Cleve and Zeller [16] use delta debugging to isolate a chain of state transitions in a failing pass that leads to failure. Given a failing and a passing trace, first their method uses memory graphs [54] to find common variables in the two runs. The common variables are used to represent the state of the program. The states of program in both executions are contrasted to candidate a cause variable. They check the cause variable in every step. When it changes in a step, is called the *cause transition*. Programmer can utilize chain of cause transitions for debugging. Usually, the locations that cause transitions take place are potential places to fix the program.

7 Dynamic Approaches

Fault are places in a program where they need to be modified to fix the program. Hence, if a modification to the program at a location changes the output of a failure trace to success, such location would be potential fault in the original program. In this section, we describe two of these techniques which are based on this idea:predicate switching, and value replacement.

Predicate switching starts with an input that entails a failure for th program, it runs the program with one predicate switched at a time [50]. If switching a predicate makes the program succeed, the predicate is identified as the source of failure. Since there are many predicates in the program they employ two heuristics to accelerate reaching such predicates. the predicate switching:LEFS, and PRIOR. Last Executed First Switched (LEFS) ordering, switches the last executed predicate. Prioritization-based ordering (PRIOR) which partition predicates into two category: high and low priorities. PRIOR gives high priority to predicates that can be influenced by the faulty run. Then, PRIOR starts with predicates in high priority group. Predicate switching approach fails to find bugs data part of program. Its developers justify the data part of program constitutes a small portion of program comparing to control part.

In a failure trace, if the incorrect outcome of a faulty statement is replaced with the correct value, a failing execution changes to a successful one. Based

on this intuition, Jeffrey et al. propose *value replacement* technique [24]. In this technique, the value of variable in statements, are systematically changed to a value from value profile of the statement. Value profile of a statement in a test suit is all values of variables at the statement in all (failing/successful) traces. For predicates the profile also includes the result of branching (taken/not taken). The authors observe that the growth of size of value profiles is logarithmic in size of test suit. Given a failing trace and value profiles, the value replacement method executes the program until a statement, then it replaces a value in the statement with another one from the value profile of the statement, and executes the rest of program. The authors propose some heuristics to reduce the number of statements subject to value replacement and the number of values to be replaced.

8 Formal Techniques

In this section, first, we briefly introduce the concept of model checking (Section 8.1). Then, in Section 8.2, we study some techniques that locate faults using the results of model checking procedures.

8.1 Model Checking

Model checking is the problem of verifying a property ϕ in a transition system R which models a system/program [?]. Intuitively, R represents states of a system, and the transitions between them. ϕ denotes a property of interest that we want to investigate. Model checking procedures include sets of algorithms to validate if model R holds the ϕ property. If R violates ϕ, the model checking procedure returns a *counter-example* that details an example (i.e. a failure trace) in R that leads to the violation of ϕ. Transition system R can represent many software and hardware systems and property ϕ is usually a temporal property that needs to be held by R.

Algorithmically, model checking techniques can be divided into two categories: explicit state and symbolic. Explicit state model checking enumerates the reachable states from initial states in R explicitly. That is, it keeps the concrete value of variables for each state. Symbolic model checkers encode states as boolean formulae. For this, symbolic model checkers may use Binary Decision Diagram (BDD) [**?**] or its variants for concise encoding of formulae [**?**], or they may reduce the problem of model checking to a SAT/SMT problem and utilize SAT/SMT solvers for the verification of properties [**?**]. Discussion about model checking and related topics is out of the scope of this paper; we refer the interested readers to [**?**] for a concise treatment of these topics.

8.2 Fault Localization Techniques

When a model checking procedure finds a violation of a desired property in the transition system of a program, it returns a counter-example, which is a failing

trace of the program. In this section, we study several techniques that use a counter-example in a model which try to locate the faults in the model/program. In Section 8.2, we look at one of the first techniques in the area that contrasts the transitions in one or multiple counterexamples with passing traces. In Section 8.2, we glance at techniques that given a counter-example, attempt to build the *closest* passing trace to the counter-example. These methods are in a sense similar to delta debugging, but they are trying to build similar traces. Contrasting failing and passing traces can help to find suspicious transitions. Finally, Section 8.2, describes a reduction of fault localization problem to Max-SAT problem.

Contrasting Counterexamples with correct traces Using a Single Counter-example

Ball et al. devised a technique for fault localization relying on the premise that execution of the fault is necessary for the software failure [11]. In their technique, whenever a counter-example was found, it finds all traces that conform to the property (passing traces). Then, by contrasting transitions in the passing traces and the counter-example, the faults that led to the deviation from the property are found and reported as a fault. Furthermore, if the program contains several faults, those faults that already have been found are replaced by the `halt` statement. `Halt` statement semantically stops the execution of the program, thus if a trace leads to a fault, it would not be resumed to avoid the error.

Merits and Shortcomings. This technique has been implemented in the SLAM [**?**] model checker. It has been experimented on some Windows device drivers. Its major caveat is that it cannot handle incidental correctness. That is, if a fault also appears in the passing trace, this technique fails to find it. Another issue is using SLAM for implementation of this technique. SLAM implements a model checking technique based on predicate abstraction.The states of program with predicate abstraction do not store concrete values of program variables, instead they contain predicates on the values. This may add some infeasible traces to the set of correct traces, because predicate abstraction does not check the validity of paths for the non-counter-examples. In this case, it is possible that fault transitions be added to the set of correct transitions and this approach fails to identify them.

Using Multiple Counter-examples

The above technique relies on one counter-example that violates the desired properties at a particular location. However, it is possible that multiple counter-examples exist that violate the properties at the same location. For that, Groce and Visser introduce the notion of *negative* and *positive* traces with respect to a counter-example and propose transition analysis on the traces [19].

Set of negative traces (`neg`) with respect to a counter-example t that ends up with an error state s_e contains all traces that start from initial states and end at s_e such that the control location immediately before the s_e is similar to the control location in t. In other words, `neg(t)` contains all counter-examples that violate same property, similarly. Likewise, a set of positive traces (`pos`) w.r.t. t is

defined, except that they take the control location immediately before the s_e but they proceed. Moreover, they are not prefix to any of traces in neg.

Given pos and neg, this technique analyzes the transitions. The analysis forms two groups all and only for each set. all(pos) denotes transitions that exist in in *all* traces in pos. Similarly all(neg) contains transitions that appear in all traces in neg. only(pos) denotes transitions in all(pos) but not in any trace in neg. Likewise, only(neg) is the set of transitions in all(neg) but not in any of traces in pos.

To illustrate this technique, Figure 3 depicts a snippet of a program that, iteratively, takes a lock randomly, and releases it. A desired property in this program is: "lock can be released, if it has been acquired." The fault in this program resides on the Line 9 where it should be inside the scope of the second if-statement. A counter-example for this program is $1 \rightarrow 2 \rightarrow 3 \rightarrow 7 \rightarrow 9 \rightarrow 10 \rightarrow 3 \rightarrow 7 \rightarrow 8 \rightarrow Error$.

- all(neg) = {1, 2, ¡3,F¿, 7, 8, 9 }
- all(pos)= { 1,2, ¡3,T¿,4,5,7,8 }
- only(neg) = {¡3,F¿, 9}
- only(pos) = { }

```
1  int got_lock = 0;
2  while(true){
3     if(random(2) == 0){
4       lock();
5       got_lock++;
6      }
7      if(got_lock !=0)
8        unlock();
9      got_lock --;
10  }
```

Fig. 3: Example

The technique uses the definition of all and only to define the cause of failure as the transitions in both all(neg) and only(neg). That is, transitions that all traces of neg(t) and did not appear in any of passing traces. Given the above values for all's and only's, the cause of failure is determined as the set of transitions: cause(neg) = {<3,F>, 9}. cause(neg) identifies the cause of the failure as a combination of not taking the first if-statement and execution of statement Line 9.

Merits and Shortcoming. This method has been implemented in Java PathFinder. It could find the cause of an error in a buggy version of DEOS real-time operating system. This techique also suffers from incidental correctness problem. That is, if the fault can appear in passing traces and failing traces, this technique is not helpful.

Distance Metrics Given a counter-example, Groce proposed a technique to find the *closest* passing trace to the counter-example [**?**, **?**, **?**, **?**]. *Distance* between a failing trace (i.e. counterexample) a and a passing trace b is defined as the number of differences between a and b. This algorithm has been implemented in `explain` tool [?]. `explain` uses the bounded model checker CBMC[3]. Figure 4 shows a version of minmax program that intended to find the minimum and maximum of three input variables. It includes an assertion that states the value of minimum must be less than or equal to the maximum value. The fault is at Line 10, where the correct assignment is `least = input2`. Figure 5 depicts a counter-example for the minmax program.

Given a counterexample a for a program P that violates assertion p, `explain` transforms P to SSA format such that each variable is defined (i.e. assigned) only once. The comments in the Figure 4 show variables to be added for SSA format. If P includes loops, P are unwound to the length of a plus a small constant integer. To find a passing trace that does not violate the assertion, it is sufficient to conjunct a propositional formula for P, say $symb(P)$ with the assertion p, i.e. $symb(P) \wedge p$. Any satisfiable valuation to this formula would be a passing trace for P.

```
1 int main(){
2   int input1, input2, input3;
3   int least = input1;   //least#0
4   int most  = input1;   //most#0
5   if(most < input2)     //guard#1
6     most = input2;      //most#1,2
7   if(most < input3)     //guard#2
8     most = input3;      //most#3,4
9   if(least > input2)    //guard#3
10    most = input2;      //most#4,5
11  if(least > input3)    //guard#4
12    least = input3;     //least#1,2
13  assert (least <= most);
14 }
```

Fig. 4: Buggy Minmax program, courtesy of [?].

Then, `explain` adds a set of constraints that represent the difference between a and the passing traces. If the variables of the SSA format of P have values: $v_1 = v_1^a$

[3] CBMC verifies safety properties in C programs. It constructs a propositional formula (say M) that represents the transition system of the program. If a program contains loops, CBMC unwinds loops for a particular number of iterations. To verify assertion p, the negation of p is conjuncted to M. If $M \wedge \neg p$ is satisfiable, it means that there is a counterexample that violates the p, otherwise the program satisfies the property.

$v_2 = v_2^a$
...
$v_n = v_n^a$
The related constraints would look like:
$v_1\Delta = (v_1! = v_1^a)$
$v_2\Delta = (v_2! = v_2^a)$
...
$v_n\Delta = (v_n! = v_n^a)$.
For example, `input1#0`Δ `==(input1#0` $!=$ `1)`. Obviously, the above constraints would not affect the result of the satisfiablity problem. Afterwards, `explain` uses a Pseudo-Boolean Solver (PBS) to solve the resulting formula. PBS is similar to SAT/SMT solvers except it accepts formulas in form $\Sigma w_i.b_i \bowtie k$ where $\bowtie$ is either of $\{<, >, \leq, \geq, =\}$ and tries to solve it. b_i is a boolean variable and, w_i and k are constant values. `explain` chooses $b_i = v_i\Delta$ and $w_i = 1$. Then, starting from $k = 1$, it incrementally increases the k and invokes the PBS until it finds a valuation that satisfies $symb(P) \wedge p$ and the constraints on Δ's. The result is a passing trace, say b that its distance to a is k. The difference of a and b is reported to the programmer. For example, Figure 6 shows resutls of application of this technique to the minmax program. It shows that in the nearest passing trace, the result of `guard#3` is not true, meaning that branch at Line 9 is not taken and respectively Line 10 is not executed. It can lead developers to Line 10.

```
input1 = 1
input2 = 0
input3 = 1
least#0= 1
most#0 = 0
\guard#1 = false
most#1 = 0
most#2 = 1
\guard#2 = false
most#3 = 1
most#4 = 1
\guard#3 = true
most#5 = 0
most#6 = 0
\guard#4 = false
least#1 = 1
least#2 = 1
```

Fig. 5: counterexample for minmax in Figure 4

A potential problem in using SSA format of program in definition of Δ's is that it may count variables that are not executed in the program. To address

this, explain introduces Δ-slicing which slices the parts of programs that are not executed or are not related to assertion p.

```
Value changed: input2 from 0 to 1
Value changed: most#1 from 0 to 1
guard changed: least#0 > input2#0 (\guard#3) was TRUE
Value changed: most#5 from 0 to 1
Value changed: most#6 from 0 to 1
```

Fig. 6: Result of execution of explain on the minmax

Merits and Shortcoming. explain has been experimented on some versions of Tcas[4] program from Siemens test subjects [**?**] and also on a micro-kernel. It was compared to nearest neighborhood method [39] that is a spectra-based fault localization technique. The results show that in some cases it outperforms the nearest neighborhood technique. The basic underlying caveat of this techniques is that it relies on symbolic model checking and PBS that hardly scale to large programs.

Abstract Counterexamples Predicate abstraction model checking relies on the fact that not all components of a program are involved in a specific property. Therefore, instead of considering values of all variables in the program, it stores a set of predicates on the state of the program that actually affect the property under investigation. It helps scaling up model checking for larger programs.(See [**?**, ?] for further details.)

In [**?**], explain has been extended to predicate-based abstract counter-examples. The new technique has been implemented on top of MAGIC [**?**] model checker.The new algorithm is the same as explain, except for comparison of failing and passing traces it uses predicates. But this comparison is not straightforward like Δ's in SSA, because at each control location, MAGIC may keep different predicates. Thus, the new technique attempts to align the states of passing and failing counterexamples such that they can be comparable. States in predicate abstraction are like $\{(s_1, \alpha_1), (s_2, \alpha_2), ..., (s_i, \alpha_i), ...\}$, where s_i is composed of predicates at state i and α_i represents transition in state i. $c(s_i)$ denotes the control location on state s_i, furthermore $p_j(s_i)$ denotes the jth predicate in s_i.

The alignment is defined as follows and assures the states are aligned if they are comparable. The alignment is one-to-one and it preserves the order of states in the transitions system.

[4] Tcas is a program for traffic collision avoidance in avionic system. An implementation of it that is available at Siemens test subject is a popular test subject.

$$align(i,j) = \begin{cases} 1 & \text{if } (c(s_i) = c(s_j) \wedge \\ & align(i,k) = 0 \text{ for } k \neq j \wedge k \leq |b| \wedge \\ & align(k,j) = 0 \text{ for } k \neq i \wedge k \leq |a| \wedge \\ & align(m,n) = 0 \text{ for } m < i \wedge n > j \wedge \\ & align(m,n) = 0 \text{ for } m > i \wedge n < j \\ 0 & \text{otherwise} \end{cases}$$

The unalignment is defined as not being aligned.
$unaligend_a(i) = \neg \bigvee align(i,j)$
$unaligend_b(j) = \neg \bigvee align(i,j)$

Then, the difference between two traces a and b is defined as $d(a,b) = W_p.\Delta p(a,b) + W_\alpha.\Delta\alpha(a,b)) + W_c.\Delta c(a,b)$, where $\Delta p(a,b)$ denotes the total number of predicate differences in all aligned states and $\Delta\alpha(a,b)$ is the total number of differences in actions in the aligned states. $\Delta c(a,b)$ denotes the total number of states that are *not* aligned. W_p, W_α and W_c are weights for each of those terms. For the sake of minimizing the difference of states, the weights have been chosen as $W_p = W_\alpha = 1$ and $W_c = max(|p(s^a)|) + 2$. Afterwards, $d(a,b)$ is added to the transition relation as in `explain` and a PBS is used to minimize the difference of a and b. Figure 7 is the result of using this technique on the bug in the minmax program.

```
Control location deleted (step #5):
10: most = input2
{most = [ $0 == input2 ]}
------------------------
Predicate changed (step #5):
was: most < least
now: least <= most
Predicate changed (step #5):
was: most < input3
now: input3 <= most
------------------------
Predicate changed (step #6):
was: most < least
now: least <= most
Action changed (step #6):
was: assertion failure
------------------------
```

Fig. 7: distance of a passing trace and counter-example in abstract terms for minmax program.(Courtesy [?])

Shortcomings and Strengths. Good predicates provide a higher level explanation of the failure and there are more intuitive. Essentially, it can summarize the conditions under which a program fails. Moreover, model checking with predicate abstraction usually scales better than model checking without abstraction. On the other hand, predicate abstraction may need numerous iterations adding numerous predicates to states in order to produce a passing traces. It may result a conjunction of several predicates to explain the counter-example which it is not necessarily helpful. This technique has been used for faulty versions of some small programs (upto 350 LOC) and some moderate size of code (3 KLOC).

Reduction to Max-SAT Propositional representation of program traces has facilitated reduction of various program analysis and verification to propositional logic problems (See [**?**, **?**, **?**]). In this section, we look at reduction of fault localization to Max-SAT problem.

The counter-example includes an input data that leads to the violation and the trace. Using the program specification (say p), a failing output (I) and the corresponding symbolic encoding of trace of the failing input($SP(I)$), Jose and Majumdar has built BugAssist that reduces the problem of fault localization to the Max-SAT problem [28, **?**].

Max-SAT problem is the problem of finding a valuation for a SAT formula that satisfies the *maximum* number of clauses in the formula. Max-SAT problem, like 3-SAT problem, is an NP-hard problem and like SAT-Solvers, several Max-SAT solvers exist. As a result, Max-SAT solvers also show the clauses that are not satisfied by the proposed valuation. The set of clauses that are not satisfied by the valuation is called minimal "UNSAT-core" of the formula. An un-satisfiable formula can have several UNSAT-cores.

An interesting feature of modern Max-SAT solvers is the ability to allow users to divide the constraints (i.e. clauses) in the formula into two categories: soft and hard. Hard constraints denote the constraints that must be satisfied by the valuation and soft constraints are those constraints that can be left unsatisfied. It should be recalled that in SAT problems *all* constraints must be satisfied. Furthermore, some Max-SAT solvers let users to associate weights to clauses and they attempt to maximize the total weight of satisfied clauses.

Given a program and a failing input, BugAssist generates the symbolic representation of the trace of the failure(say P), then it builds the propositional formula $I \wedge P \wedge p$. where I is a formula that denotes the input of program and p is the specification of program. BugAssist makes constraints I and p hard constraints and P constraints are soft constraints. That is, the Max-SAT solver must find the minimal UNSAT-CORE within P. UNSAT-cores are the *cause* of unsatisfiability of the formula, hence we can assume that there are reasons for program failure. For example, consider the program in Figure 8. It fails for input `index = 1`. Thus, in this program $I = (index == 1)$, the specification is $p = (i \geq 0 \wedge i < 3)$, and $P = (index1 == 1 \wedge index2 == index1 + 2 \wedge i == index2)$, where I and p are hard constraints. The UNSAT-core of this example is $index2 == index1 + 2$.

The inspection of the corresponding statement and related statements can lead us to the fault.

An UNSAT-core might be incomplete or insufficient to explain the program failure. Thus, BugAssist employs an iterative process that uses the feedback from the user. For this, when the Max-SAT procedure in BugAssist finds an UNSAT-core u of the formula, it reports it to the user. The user inspects u as clue for debugging, if she found it useless for debugging, she asks for another UNSAT-core. BugAssist adds u to the hard constraints and then reruns the Max-SAT procedure.

Merits and shortcoming. Perhaps the most significant contribution of BugAssist is the reduction of fault localization to Max-SAT problem and using off-the-shelf solvers to solve it. Another contribution is that it needs only a failing trace and the corresponding input to form the corresponding Max-SAT formulation. In other word, it does not need passing traces for fault localization. On the downside, it suffers from the problem of scalability that all of symbolic techniques suffer. However, it suggests using dynamic symbolic techniques or slicing, delta debugging to reduce the size of propositional formulae or failing traces. BugAssist has been experimented on Tcas program of the Siemens test subject and on avarage it could reduce the search space for the fault to 8% of the program.

```
int Array[3];
int testme(int index)
{
  if ( index != 1) /* Potential Bug 2 */
     index = 2;
  else
     index = index + 2; /* Potential Bug 1 */
  i = index;
  assert(i >= 0 && i < 3);
  return Array[i];
}
```

Fig. 8: sample program

9 Discussion

In this paper, we have looked at two different approaches to fault localization. In this section, we discuss the viability of these approaches. First, we revisit the goal of fault localization and relate them with the techniques that we studied in this paper. Finally, we discuss about a type of fault that is not addressed by the techniques presented in this paper, missing component fault.

9.1 Goal of Automated Fault Localization

Automation of fault localization techniques intends to facilitate and accelerate debugging by finding the suspicious components of programs. A large portion of software development (35% according to [**?**]) is spent on navigating through programs for maintenance. Thus, there is an opportunity for the automated fault localization techniques to reduce that. Their success in this mission depends on their effectiveness and efficiency.

Effectiveness An effective fault localization technique should point to the places that actual faults reside and provide some clues for fixing the fault. In short, it should be *precise*, and *informative*.

Precision Precision demands low false positives. The output of an FL technique should not confuse developers by pointing to too many different, irrelevant components of a program as suspects of a failure. Current automated FL techniques usually either produce a set of suspicious statements without any particular ranking, or they devise a suspiciousness factor and then rank all statements according it. The techniques that we described in this paper fit in the first category; they just give a set of suspicious statement and do not rank them.

There are two major metrics that researchers use to evaluate the FL techniques. One metrics assumes that statements are ranked by suspiciousness. The precision of the technique is the percent of statements in the program that must be examined to reach the fault, starting from the most suspicious statement [27, 6]. The other metric uses program dependence graph (PDG) [23] to evaluate FL techniques. That is, given a suspicious statement, the percentage of nodes in PDG that needs to be examined to reach the fault node determines the accuracy of a method [39, 16]. It is worth mentioning that a recent study [**?**] has shown that developers, on average, only pay attention to the first 10 suggestions of FL techniques and would dismiss the rest of results. In other words, an automated FL technique is useful if the actual fault is in its 10 first suggestions.

.

Informativeness Although identifying some statements as suspects for a failure reduces the search space for debugging, it barely helps developers to design a fix for the program. In other words, each suspicious statement is a hypothesis about the location of fault and it would be better to accompany with the justifications about why the FL technique has inferred a component as the potential bug. Given such information, developers can add their own knowledge about the program to decide if a particular location is the cause of the failure and if so, what would be the best fix for that?

In the cause transition technique, Section **??**, the hypothesis is backed with the fact that a cause transition has happened at a location. Techniques using contrasting, Section 8.2 justify selection based on absence of a transition in a passing trace. Likewise, BugAssist in Section 8.2, just relies on the fact that the

suspicious statement appear in the UNSAT-core of the trace formula. Although all of these justifications seem appealing, they are easy to refute; e.g. it is possible if we try different inputs we get different cause transitions, or a faulty transition can appear in some passing traces. We wish to have a better explanation of the participation of a potential fault in a failure. The technique for explaining abstract counter-examples, described in Section 8.2, has the advantage of summarizing the failure in higher abstraction (predicates on program variables) than other techniques. We do no know the most suitable level of abstraction for explaining failures or justifying a fault at a particular location; e.g. if a chain or series of why questions [**?**, **?**] the best. But it seems reasonable to expect fault localization techniques to augment suspicious program components with the information about how they could have contributed to the error.

Efficiency Software debugging should be performed timely, within the timing and budgeting constraints. This constraint also applies to fault localization as well. It should be efficient. We identify two metrics for efficiency: scalability, and information usage.

Performance Time and budget for debugging is limited. This limitation also applies to the fault localization. thus, a fault localization technique must terminate in the timely manner. Depending to the size of program and the property under investigation, model checking techniques can take long time to produce the result. Delta debugging may also take a long time, because it uses debuggers extensively.

Scalability As we have seen in this paper, the techniques vary in their scalability. The methods based on model checking and symbolic execution, Section **??**, suffer from scalability. That is they cannot be used for large programs. On the other hand, delta debugging techniques are able to scale better.

Information Usage Each fault localization technique needs some information from the program for its processing. It consumes this information to infer locations of faults. There are two dimensions on the information consumption: what it needs to start with, and what it can exploit to boost its precision or performance.

Techniques based on delta debugging, require little knowledge about the program and specification; they can started from a failing input and passing input, which are usually available. On the other hand, techniques based on model checking need formal specification of programs to be able to work, which is unlikely to exist for all programs.

In addition to a program itself, there are many other sources of information that can be used; e.g. change history, developers, and found errors. Each of these can be exploited to enhance the fault localization. For example, we may be able to enhance fault localization by identifying components with complex

behaviors; these components may be identified by their developers or comments in the program. We have not seen usage of such information in any of techniques that described in this paper.

9.2 Missing Component

Software faults can be partitioned into two broad categories: wrong algorithmic component and missing component. While the former has to do with statements that are faulty (e.g. using wrong operator), the latter characterizes situations when a piece of program logic is missing (e.g. missing an assignment statement, or a conditional statement). A study of real faults in some open source project has shown that the missing component faults constitutes 64% of real faults [18]. Majority of proposed techniques attempt to find the wrong algorithmic component faults. Thus, there is a need to devise techniques to address these faults.

References

1. ABBOTT, P. Tricks of the trade. *The Mathematica Journal 5*, 4 (1995), 27–34.
2. ABREU, R., ZOETEWEIJ, P., AND GEMUND, A. J. C. V. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)* (2007), pp. 89–98. Generated by Odysci - http://www.odysci.com/article/1010112995982027.
3. AGRAWAL, H., DE MILLO, R., AND SPAFFORD, E. An execution-backtracking approach to debugging. *Software, IEEE 8*, 3 (may 1991), 21 –26.
4. AGRAWAL, H., AND HORGAN, J. R. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation* (New York, NY, USA, 1990), PLDI '90, ACM, pp. 246–256.
5. AGRAWAL, H., HORGAN, J. R., LONDON, S., AND WONG, W. E. Fault localization using execution slices and dataflow tests. In *International Symposium on Software Reliability Engineering* (1995).
6. ALI, S., ANDREWS, J., DHANDAPANI, T., AND WANG, W. Evaluating the accuracy of fault localization techniques. In *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on* (nov. 2009), pp. 76 –87.
7. ALLEN, E. B., KHOSHGOFTAAR, T. M., AND CHEN, Y. Measuring coupling and cohesion of software modules: An information-theory approach. In *Proceedings of the 7th International Symposium on Software Metrics* (Washington, DC, USA, 2001), METRICS '01, IEEE Computer Society, pp. 124–.
8. ARUMUGA NAINAR, P., CHEN, T., ROSIN, J., AND LIBLIT, B. Statistical debugging using compound boolean predicates. In *Proceedings of the 2007 international symposium on Software testing and analysis* (New York, NY, USA, 2007), ISSTA '07, ACM, pp. 5–15.
9. ARUMUGA NAINAR, P., AND LIBLIT, B. Adaptive bug isolation. In *32nd International Conference on Software Engineering (ICSE 2010)* (Cape Town, South Africa, May 2010), P. Devanbu and S. Uchitel, Eds., ACM SIGSOFT and IEEE.
10. BALL, T., AND LARUS, J. R. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture* (Washington, DC, USA, 1996), MICRO 29, IEEE Computer Society, pp. 46–57.

11. BALL, T., NAIK, M., AND RAJAMANI, S. K. From symptom to cause: localizing errors in counterexample traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2003), POPL '03, ACM, pp. 97–105.
12. BELL, R. M., OSTRAND, T. J., AND WEYUKER, E. J. Does measuring code change improve fault prediction? In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering* (New York, NY, USA, 2011), Promise '11, ACM, pp. 2:1–2:8.
13. BELL, R. M., WEYUKER, E. J., AND OSTRAND, T. J. Assessing the impact of using fault prediction in industry. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops* (Washington, DC, USA, 2011), ICSTW '11, IEEE Computer Society, pp. 561–565.
14. BESZEDES, A., GERGELY, T., FARAGO, S., GYIMOTHY, T., AND FISCHER, F. The dynamic function coupling metric and its use in software evolution. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 103–112.
15. CHILIMBI, T. M., LIBLIT, B., MEHRA, K., NORI, A. V., AND VASWANI, K. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering* (Washington, DC, USA, 2009), ICSE '09, IEEE Computer Society, pp. 34–44.
16. CLEVE, H., AND ZELLER, A. Locating causes of program failures. In *Proceedings of the 27th international conference on Software engineering* (New York, NY, USA, 2005), ICSE '05, ACM, pp. 342–351.
17. DEMILLO, R. A., PAN, H., AND SPAFFORD, E. H. Critical slicing for software fault localization. *ACM Sigsoft Software Engineering Notes 21* (1996), 121–134.
18. DURAES, J., AND MADEIRA, H. Emulation of software faults: A field data study and a practical approach. *Software Engineering, IEEE Transactions on 32*, 11 (nov. 2006), 849 –867.
19. GROCE, A., AND VISSER, W. What went wrong: explaining counterexamples. In *Proceedings of the 10th international conference on Model checking software* (Berlin, Heidelberg, 2003), SPIN'03, Springer-Verlag, pp. 121–136.
20. GUPTA, N., HE, H., ZHANG, X., AND GUPTA, R. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering* (New York, NY, USA, 2005), ASE '05, ACM, pp. 263–272.
21. GYIMÓTHY, T., BESZÉDES, A., AND FORGÁCS, I. An efficient relevant slicing method for debugging. In *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering* (London, UK, 1999), ESEC/FSE-7, Springer-Verlag, pp. 303–321.
22. HARROLD, M. J., ROTHERMEL, G., WU, R., AND YI, L. An empirical investigation of program spectra. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (New York, NY, USA, 1998), PASTE '98, ACM, pp. 83–90.
23. HORWITZ, S., AND REPS, T. The use of program dependence graphs in software engineering. In *IN PROCEEDINGS OF THE FOURTEENTH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING* (1992), pp. 392–411.
24. JEFFREY, D., GUPTA, N., AND GUPTA, R. Fault localization using value replacement. In *Proceedings of the 2008 international symposium on Software testing and analysis* (New York, NY, USA, 2008), ISSTA '08, ACM, pp. 167–178.

25. JIANG, L., AND SU, Z. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering* (New York, NY, USA, 2007), ASE '07, ACM, pp. 184–193.
26. JONES, J. A., AND HARROLD, M. J. Empirical evaluation of the tarantula automatic fault-localization technique. In *Automated Software Engineering* (2005), pp. 273–282.
27. JONES, J. A., HARROLD, M. J., AND STASKO, J. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering* (New York, NY, USA, 2002), ICSE '02, ACM, pp. 467–477.
28. JOSE, M., AND MAJUMDAR, R. Cause clue clauses: error localization using maximum satisfiability. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2011), PLDI '11, ACM, pp. 437–446.
29. KATZ, I. R., AND ANDERSON, J. R. Debugging: an analysis of bug-location strategies. *Hum.-Comput. Interact. 3*, 4 (Dec. 1987), 351–399.
30. KIM, S., ZIMMERMANN, T., WHITEHEAD JR., E. J., AND ZELLER, A. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering* (Washington, DC, USA, 2007), ICSE '07, IEEE Computer Society, pp. 489–498.
31. LAWRANCE, J., BOGART, C., BURNETT, M., BELLAMY, R., RECTOR, K., AND FLEMING, S. How programmers debug, revisited: An information foraging theory perspective. *Software Engineering, IEEE Transactions on PP*, 99 (2010), 1.
32. LIBLIT, B., AIKEN, A., ZHENG, A. X., AND JORDAN, M. I. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation* (New York, NY, USA, 2003), PLDI '03, ACM, pp. 141–154.
33. LIBLIT, B., NAIK, M., ZHENG, A. X., AIKEN, A., AND JORDAN, M. I. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 15–26.
34. LIU, C., YAN, X., FEI, L., HAN, J., AND MIDKIFF, S. P. Sober: statistical model-based bug localization. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2005), ESEC/FSE-13, ACM, pp. 286–295.
35. MCCONNELL, S. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004.
36. NAGAPPAN, N., ZELLER, A., ZIMMERMANN, T., HERZIG, K., AND MURPHY, B. Change bursts as defect predictors. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on* (nov. 2010), pp. 309 –318.
37. OSTRAND, T. J., AND WEYUKER, E. J. Software fault prediction tool. In *Proceedings of the 19th international symposium on Software testing and analysis* (New York, NY, USA, 2010), ISSTA '10, ACM, pp. 275–278.
38. PAN, H., AND SPAFFORD, E. H. Heuristics for automatic localization of software faults. Tech. rep., Purdue University, 1992.
39. RENIERES, M., AND REISS, S. Fault localization with nearest neighbor queries. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on* (oct. 2003), pp. 30 – 39.
40. REPS, T., BALL, T., DAS, M., AND LARUS, J. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of*

the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering (New York, NY, USA, 1997), ESEC '97/FSE-5, Springer-Verlag New York, Inc., pp. 432–449.

41. SANTELICES, R., JONES, J. A., YU, Y., AND HARROLD, M. J. Lightweight fault-localization using multiple coverage types. In *Proceedings of the 31st International Conference on Software Engineering* (Washington, DC, USA, 2009), ICSE '09, IEEE Computer Society, pp. 56–66.
42. TAN, L., ZHOU, Y., AND PADIOLEAU, Y. acomment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *Software Engineering (ICSE), 2011 33rd International Conference on* (may 2011), pp. 11 –20.
43. THAKUR, A., SEN, R., LIBLIT, B., AND LU, S. Cooperative crug isolation. In *Proceedings of the Seventh International Workshop on Dynamic Analysis* (New York, NY, USA, 2009), WODA '09, ACM, pp. 35–41.
44. TREVOR HASTIE, R. T., AND FRIEDMAN, J. *The Elements of Statistical Learning Data Mining, Inference, and Prediction*. Springer, 2001.
45. WANG, X., CHEUNG, S., CHAN, W., AND ZHANG, Z. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on* (may 2009), pp. 45 –55.
46. WEISER, M. Program slicing. In *Proceedings of the 5th international conference on Software engineering* (Piscataway, NJ, USA, 1981), ICSE '81, IEEE Press, pp. 439–449.
47. YILMAZ, C., PARADKAR, A., AND WILLIAMS, C. Time will tell: fault localization using time spectra. In *Proceedings of the 30th international conference on Software engineering* (New York, NY, USA, 2008), ICSE '08, ACM, pp. 81–90.
48. ZELLER, A. Isolating cause-effect chains from computer programs. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering* (New York, NY, USA, 2002), ACM, pp. 1–10.
49. ZELLER, A., AND HILDEBRANDT, R. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng. 28* (February 2002), 183–200.
50. ZHANG, X., GUPTA, N., AND GUPTA, R. Locating faults through automated predicate switching. In *Proceedings of the 28th international conference on Software engineering* (New York, NY, USA, 2006), ICSE '06, ACM, pp. 272–281.
51. ZHANG, X., HE, H., GUPTA, N., AND GUPTA, R. Experimental evaluation of using dynamic slices for fault location. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging* (New York, NY, USA, 2005), AADEBUG'05, ACM, pp. 33–42.
52. ZHANG, X., TALLAM, S., GUPTA, N., AND GUPTA, R. Towards locating execution omission errors. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2007), PLDI '07, ACM, pp. 415–424.
53. ZHANG, Z., CHAN, W. K., TSE, T. H., JIANG, B., AND WANG, X. Capturing propagation of infected program states. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (New York, NY, USA, 2009), ESEC/FSE '09, ACM, pp. 43–52.
54. ZIMMERMANN, T., AND ZELLER, A. Visualizing memory graphs. In *Software Visualization*, S. Diehl, Ed., vol. 2269 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2002, pp. 533–537. 10.1007/3-540-45875-1.