Model Checking and Delta Debugging for Automated Fault Localization; A Tutorial

Mohammad Amin Alipour
Oregon State University
Corvallis, OR, USA
alipour@eecs.oregonstate.edu

ABSTRACT

Debugging programs is a complicated and cumbersome task. It is an iterative process which includes hypothesizing the cause of a failure and validating the hypothesis. Since programs comprise of many components with complex interactions this chain often involves several trials.

Fault localization (FL) is the task of identifying components that are likely to cause the failure. Automated fault localization techniques aim to accelerate the debugging process by guiding developers to a small portion of the program that is likely to be culpable for the program failures. In essence, they reduce the search space programmers need to inspect manually to locate the fault.

There are several different approaches to fault localization. In this paper, we study two approaches: delta debugging, and model checking.

Delta debugging framework offers techniques for simplification of test cases and isolation of failure-inducing parts from test cases, which help in debugging and can be adapted for fault localization. Delta debugging requires minimal knowledge about the structure of the program and specification. In this paper, we explain delta debugging algorithms and how it could be adapted for fault localization. Model checking is the problem of checking whether a model (e.g. program) satisfies a property. Model checking needs to know the internal structure of the programs that it works on. Since it can answer different queries about the execution path of the program, it can be employed for fault localization. In this paper, we study a couple of techniques that have been implemented in model checkers for fault localization.

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.

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 [36]. 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 [41, 1, 22], spectra based fault localization [40,39,26], statistical inference [37,3], delta debugging [43] and model checking [12]. 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

[1]Understanding the process of software debugging is still an active field of research, e.g. See [15,35,29] for more.

part of them. Section 2 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 3, we provide a brief background on model checking. Then, it outlines some techniques that analyze counter-examples for fault localization. Section 4 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.

2. DELTA DEBUGGING

The starting point of debugging is a failing test case. How we can make most of it?

Failing test cases should be able to guide the debugging process. In debugging, a small, simple failing test case is more desirable than a large, complicated test case that represents the same failure. Suppose a bug in a browser which causes the browser to crash, developers hardly can inspect a large HTML file with thousands of lines that triggers the failure, developers in a reasonable amount of time. Therefore, there is a desire to *simplify* the input by minimizing the size of the input in such cases. Moreover, a successful test case which is *very similar* to a failing one helps developers to contrast two test cases and may help to further isolating parts of test case that lead to the failure.

Delta debugging framework facilitates comprehension of a failure of a program by simplification of test cases (Section 2.1) and the isolation of failure-inducing parts of test cases (Section 2.2) [43]. It assumes that test cases are composed of disjoint set of components that can be added to or removed from a test case. These components can be used to contrast test cases which in the form of deltas (δs).

2.1 Simplification of test cases

Minimization of a failing test case requires finding the minimum subset of the componenets of the test case that retain the failure. This problem is NP-hard, in general. The simplification algorithm in delta debugging, *ddmin*, takes a failing test case and produces a *1-minimal* failing test case. A failing test case is 1-minimal if removal any of its constituent components masks the failure, i.e. causes it to disappear. In this case, we can say that all present features in the 1-minimal test case are relevant to the failure.

Algorithm 1 depicts the details of *ddmin*. Assume c_f induces a failure in a program P. To simplify c_f, *ddmin* recursively splits c_f into n subset $c_1 ... c_n$ of components. *ddmin* first checks if there is $c_i, 1 \geq i \geq n$ which retains the failure, in such cases, it returns the result of simplification of c_i. Otherwise, it checks whether the elimination of each of this subsets can retain failure, i.e. $c_f - c_i$. If so, it simplifies $c_f - c_i$. If none of the previous cases happened, it increases the granularity of partitioning. Initially, this is invoked by $ddmin(P, c_f, 2)$.

Figure 1 (taken from [43]) shows a portion of an HTML file that causes the printing feature of an early version of Mozilla Firefox to crash. The original file contains 896 lines of HTML code. Given this test case, understanding the cause of failure is not a trivial task for a developer. *ddmin* simplifies this test case to a failing test case which is only a single HTML tag: `<SELECT>`. This eight-character test case is of course more understandable for debugging. *ddmin* has proved to be useful in simplifying test cases for programs which can take variable size inputs such as browsers, compilers, lexers and etc. The complexity of *ddmin* algorithm is quadratic of size of c_f.

Algorithm 1 *ddmin* algorithm for simplification of failing test case, $ddmin(P, c_f, n)$

```
Input: P, Program, $c_f$, Failure-inducing Input, and n, number
  of partitions.
  Split $c_f$ to n partitions to build $c_1, ..., c_n$
  if $\exists c_i$ such that makes P fail then
    $ddmin(P, c_i, 2)$
  else if $\exists c_i$ such that $c_f - c_i$ makes P fail then
    $ddmin(P, c_f - c_i, max(n-1, 2))$
  else if $n < size(c_f)$ then
    $ddmin(P, c_f, min(2n, size(c_f)))$
  else
    $c_f$
  end if
```

2.2 Isolating the failure-inducing parts of test cases

Test case simplification reduces the size of a test case, but the simplified failing test case may still include elements which are not necessary for the failure. For example, a minimized test case for C compiler may need to preserve some punctuation symbols, like semicolons and parenthesis, to keep the test case a valid C program, while those symbols are not related to the failure. One way to tackle this problem is to contrast the failing test case with a very similar passing test case. With that, we can *isolate* the parts in the failing test case that are not in the passing one, as the failure-inducing portion of the test case.

Delta debugging framework proposes an algorithm called *dd2*, that given a pair of passing test case and failing test case (c_p, c_f), it attempts to produce a new pair of passing test case and failing test case (c_p', c_p') with minimal differences.

Suppose a set of disjoint differences between c_p and c_f be $\Delta = \{\delta_1, \delta_2, ..., \delta_k\}$. That is, if Δ is applied to c_p , it results in c_f. Let "$\cup$" to denote application of a change in a passing test case, and "$-$" to denote removal of a change in the failing test case, e.g.

- $c_p = c_f \cup \delta_1 \cup ... \delta_k$, or
- $c_f = c_p - (\delta_1 \cup ... \delta_k)$.

We want to find a subset of changes to the (c_p,c_f) in order to reach a pair (c_p', c_f') with minimal distance that c_p' passes and c_f' fails. In general, it requires trying all subsets of changes to find minimum differences between c_p' and c_f', which has exponential time complexity. To be practical, *dd2* produces a pair of a passing test case and a failing test cases with 1-minimal distance. That is, for any $\delta_i \in c_f' - c_p'$, $c_p' \cup \delta_i$ would not be a passing test case, and similarly $c_f' - \delta_i$ is not a failing test case. In other words, the difference between two test cases cannot be reduced further.

Algorithm 2 outlines the *dd2* algorithm. In this algorithm, n represents the granularity of changes that partitions changes into n partitions. For example, if the granularity is 2, it tries

```
<td align=left valign=top><SELECT NAME="op sys" MULTIPLE SIZE=7><OPTION VALUE="All">All<OPTION VALUE="Windows 3.1">
Windows 3.1<OPTIONVALUE="Windows 95">Windows 95<OPTION VALUE="Windows 98">Windows 98<OPTION VALUE="Windows ME">
Windows ME<OPTION VALUE="Windows 2000">Windows2000<OPTION VALUE="Windows NT">Windows NT<OPTION VALUE="Mac System 7">
Mac System 7<OPTION VALUE="Mac System 7.5">Mac System 7.5<OPTION VALUE="MacSystem 7.6.1">Mac System 7.6.1<OPTION VAL
UE="Mac System 8.0">Mac System8.0<OPTION VALUE="Mac System 8.5">Mac System 8.5<OPTION VALUE="Mac System8.6">Mac Syst
em 8.6<OPTION VALUE="Mac System 9.x">Mac System 9.x<OPTIONVALUE="MacOS X">MacOS X<OPTION VALUE="Linux">Linux<OPTION
VALUE="BSDI">BSDI<OPTION VALUE="FreeBSD">FreeBSD<OPTIONVALUE="NetBSD">NetBSD<OPTION VALUE="OpenBSD">OpenBSD<OPTION
VALUE="AIX">AIX<OPTION VALUE="BeOS">BeOS <OPTION VALUE="HP-UX">HP-UX<OPTIONVALUE="IRIX">IRIX<OPTION VALUE="Neutrino
">Neutrino<OPTION VALUE="OpenVMS">OpenVMS<OPTION VALUE="OS/2">OS/2<OPTION VALUE="OSF/1">OSF/1<OPTION VALUE="Solaris"
>Solaris<OPTIONVALUE="SunOS">SunOS<OPTION VALUE="other">other</SELECT></td><td align=left valign=top><SELECT NAME="p
riority" MULTIPLE SIZE=7><OPTION VALUE="--">--<OPTION VALUE="P1">P1<OPTION VALUE="P2">P2<OPTIONVALUE="P3">P3<OPTION
VALUE="P4">P4<OPTION VALUE="P5">P5</SELECT></td><td align=left valign=top><SELECT NAME="bug severity" MULTIPLE SIZE=
7><OPTION VALUE="blocker">blocker<OPTION VALUE="critical">critical<OPTIONVALUE="major">major<OPTION VALUE="normal">
normal<OPTIONVALUE="minor">minor<OPTION VALUE="trivial">trivial<OPTION VALUE="enhancement">enhancement</SELECT></tr>
</table>
```

Figure 1: A snippet of a HTML file that caused a crash in Mozilla printing.

to see if application/removal of half of changes can make a passing test case to either to fail or vice versa. It recursively narrows the difference between c_p' and c_f'. Moreover, if a coarse granularity does not help to narrow down the gap between the passing and failing test cases, it increases the granularity. In this algorithm function `test(t)` tries input `t` in the program. It can have three outputs, program passes (V), program fails with similar failure to c_f[2] (X), or unknown (?). Initially, this algorithm is invoked with $dd2(c_p, c_f, 2)$.

Algorithm 2 *dd2* algorithm to isolate failing causes, dd2(c_p',c_f',n)

```
Input: $c_p'$ current passing test case, $c_f'$ current failing test
  case, and $n$, granularity of remaining $\delta$'s between $c_p'$ and
  $c_f'$ ($\Delta_1 .. \Delta_n$).
  if $\exists \Delta_i$ such that test($c_p' \cup \Delta_i$) = $X$ then
    $dd2(c_p', c_p' \cup \Delta_i, 2)$
  else if $\exists \Delta_i$ such that test($c_f' - \Delta_i$) = $V$ then
    $dd2(c_f' - \Delta_i, c_f', 2)$
  else if $\exists \Delta_i$ such that test($c_p' \cup \Delta_i$) = $V$ then
    $dd2(c_p' \cup \Delta_i, c_f', max(n-1, 2))$
  else if $\exists \Delta_i$ such that test($c_f' \Delta_i$) = $X$ then
    $dd2(c_p', c_f' - \Delta_i, max(n-1, 2))$
  else if $n < \Delta$ then
    $dd2(c_p', c_f', min(2n, |\Delta|))$
  else
    return ($c_p'$, $c_f'$)
  end if
```

In case of the bug in the Mozilla Firefox if we assume the c_f=`<SELECT NAME="priority" MULTIPLE SIZE=7>` and c_p is an empty string, the result of isolation is:

- c_f= `<SELECT NA=ty" MULTIPLE SIZE=7>`
- c_p= ` SELECT NA=ty" MULTIPLE SIZE=7>`

which the difference of the passing and failing test cases is "<" in the beginning of c_f. The computational complexity of *dd2* is $O(|c_f|^2)$.

ddmin and *dd2* algorithms are powerful tools for understanding failures hence improving the debugging process. These algorithms also have been used for fault localization. In the Section 2.3, we explain how *dd2* can be used for comparing two states of a program and identifying the variables which contribute to a failure of the program as *causes* of the failure. In Section 2.4, we outline a method that exploits the notion of *cause* for fault localization.

[2]We assume two failures are similar, if they happen at similar location with similar stack of function call.

2.3 Search in Space

A program failure is the result of a chain of infection in the program states that is manifested in the output, or observed by the oracle. A program state is infected if the value of at least one variable in the state deviates from the expected value. Finding what variables go wrong in a failing execution of a program can be used in debugging. The program can be sliced upon the infected variables and the cause of failure (i.e. fault) can be identified in the logic that processes the variable. In the rest of this section, we look at a method that exploits the *dd2* algorithm to find the set of infected variables.

Zeller has devised HOWCOME [42], a technique to isolate variables that infect the program state, i.e. produce error that leads to failure. Suppose there are two runs r_f and r_p of a program *P*, which r_f fails and r_p succeeds. Furthermore, let assume states of r_f and r_p at a location *L* is captured by *memory graphs* [44] G_f and G_p. HOWCOME uses *dd2* algorithm, presented in Section 2.2, to G_f and G_p to find minimal changes to G_p that makes r_p to fail. A memory graph is a directed graph in which its vertices contain values and types of program variables and arrows are labeled to denote operations such as de-referencing, and variable access.

HOWCOME assumes two vertices are matching in G_p and G_f, if either they have same types and values, or they are pointers of similar types which both contain NULL or both contains non-NULL values. Furthermore, two edges are matching when their labels are similar and their source and destination vertices are matching. Two graphs can be matched by adding and removing nodes. Figure 2 shows sample states of two executions of a program and the changes (δs) between them. HOWCOME uses the (approximate) largest common subgraph algorithm to match G_p and G_f graphs. It results a set of atomic changes that transform G_p to G_f. HOWCOME uses each of these atomic changes as a δ in the *dd2* algorithm.

When *dd2* is run on G_p and G_f, it shows (1-)minimal changes to the value of variables of $G_p$that *cause* the resumption of r_p to fail. In other words, they infect the state of the program that leads to the failure. Using this information, developers could focus only on these values and inspect their impacts in the rest of the program, instead of spreading their attention

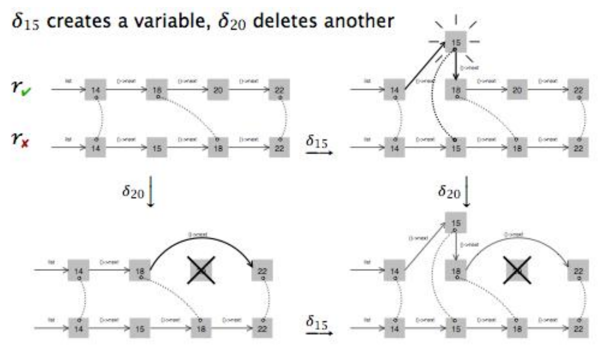

Figure 2: Memory graphs for two execution of program, possible changes. (Courtesy of [42].)

to all of variables and values.

Merits and Caveats. This approach does not require any information about the program, or developer intervention. Applying HOWCOME at different locations in the program may lead to different *causes*. Thus, one question is where the HOWCOME can be applied. Another issue is that, at a location, there are potentially different sets of variables that can be identified as causes.

Running example. We use an implementation of the Shell Sort in Figure 3 (taken from [14]) to illustrate the technique. There is a bug in this program that resides at Line 30 where the second argument to pass to `shell_sort` must be `argc-1` not `argc`. Suppose two executions of this program, F and S. F invokes the program with "`14 11`" values and S with "`8 7 9`". Let Table 1 depict the contents of memory at Line 29 for the executions of F and S. In the execution of F, since the value of a[2] is less than a[0], they are swapped and the executions ends of with a wrong output "`0 11`". In S, the value of uninitialized memory location at a[3] is greater than other elements of a, thus it is not swapped. If HOWCOME is run at Line 29, it would identify a[2] as the cause of failure. The reason is if we replace the content of a[2] in S with the content of a[2] is F, the execution of S would fail.

2.4 Searching in Time

Faults in a program are places that infect the state of the program. HOWCOME, described in the previous section, is a technique that tries to isolate (1-minimal) set of variables as suspects for the failure at a location. Based on that, Cleve and Zeller propose a heuristic to locate the faults [14].

HOWCOME compares states of failing and passing executions of a program at different locations determines *causes* at those locations. Wherever, the cause changes from adjacent statements st_{i-1} to st_i, a "cause transition" has happened. Cleve and Zeller conjecture that st_i's that make a cause transitions are suspects to embody faults. For example, in the Shell sort example, in the previous section, HOWCOME identifies `argc` as the cause for the failure at the beginning of `shell_sort` function, thus there is a cause transition at Line 30. It differs from the cause at Line 29 (`a[2]`), therefore Line 30 is a suspicious statement for the failure.

Merits and Shortcomings. This technique has been tested on Siemens test subjects [24], and could identify some of the seeded faults in them, and failed on some. The authors have employed the program dependence graph (PDG) to evaluate the accuracy of their technique. That is, starting from nodes

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  static void shell_sort(int a[], int size)
4  {
5   int i, j;
6   int h = 1;
7   do {
8     h = h * 3 + 1;
9   } while (h <= size);
10  do {
11   h /= 3;
12   for (i = h; i < size; i++)
13      {
14      int v = a[i];
15      for (j = i; j >= h && a[j - h] > v; j -= h)
16        a[j] = a[j - h];
17      if (i != j)
18         a[j] = v;
19   }
20  } while (h != 1);
21 }
22 int main(int argc, char *argv[])
23 {
24  int i = 0;
25  int *a = NULL;
26  a = (int *)malloc((argc - 1) * sizeof(int));
27  for (i = 0; i < argc - 1; i++)
28    a[i] = atoi(argv[i + 1]);
29
30  shell_sort(a, argc);
31  for (i = 0; i < argc - 1; i++)
32    printf("%d ", a[i]);
33  printf("\n");
34  free(a);
35   return 0;
36 }
```

Figure 3: A buggy implementation of the Shell Sort.

that represent cause transitions in PDG, it measures the percent of nodes in PDG which need to be examined to reach the actual fault. The experiments, show that their technique is more accurate than previous technique called the nearest neighborhood [39]. To make sense of the accuracy of this technique, in almost half of cases, 40% or more of nodes in the programs should be examined by developers before reaching to the node that represents the real fault. The authors hypothesize that some of the faults are hard to locate, like variable initialization faults.

3. MODEL CHECKING FOR FAULT LOCALIZATION

In this section, first, we briefly introduce the concept of model checking (Section 3.1). Then, in Section 3.2, we study some techniques that locate faults using the results of model checking procedures.

3.1 Model Checking

Model checking is the problem of verifying a property ϕ in a transition system R which models a system/program [12]. Intuitively, R represents states of a system, and the transitions

Table 1: Memory contents for the execution of S and F at Line 29 of the Shell Sort.

	argc	argv[0]	argv[1]	argv[2]	i	a[0]	a[1]	a[2]	a[3]
F	3	14	11	NULL	2	14	11	0	1425
S	4	8	7	9	3	8	7	9	4215

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) [9] or its variants for concise encoding of formulae [13], or they may reduce the problem of model checking to a SAT/SMT problem and utilize SAT/SMT solvers for the verification of properties [7]. Discussion about model checking and related topics is out of the scope of this paper; we refer the interested readers to [25] for a concise treatment of these topics.

3.2 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 3.2.1, 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 3.2.2, 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 3.2.3, describes a reduction of fault localization problem to Max-SAT problem.

3.2.1 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 [5]. 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 [4] 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 [21].

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 4 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)` = { }

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)`

```
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  }
```

Figure 4: Example

```
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 }
```

Figure 5: Buggy Minmax program, courtesy of [18].

```
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
```

Figure 6: counterexample for minmax in Figure 5

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.

3.2.2 Distance Metrics

Given a counter-example, Groce proposed a technique to find the *closest* passing trace to the counter-example [17, 20, 19, 18]. *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 [20]. `explain` uses the bounded model checker CBMC[3]. Figure 5 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 6 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 5 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.

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:
$v_1 = v_1^a$
$v_2 = v_2^a$
...
$v_n = v_n^a$
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 7 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.

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

[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.

```
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
```

Figure 7: Result of execution of `explain` on the minmax

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

Merits and Shortcoming. `explain` has been experimented on some versions of `Tcas`[4] program from Siemens test subjects [24] 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 [34, 4] for further details.)

In [11], `explain` has been extended to predicate-based abstract counter-examples. The new technique has been implemented on top of MAGIC [10] 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.

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

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

[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.

```
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
------------------------
```

Figure 8: distance of a passing trace and counter-example in abstract terms for minmax program.(Courtesy [11])

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 8 is the result of using this technique on the bug in the minmax program.

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).

3.2.3 Reduction to Max-SAT

Propositional representation of program traces has facilitated reduction of various program analysis and verification to propositional logic problems (See [8, 6, 33]). 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, 27].

Max-SAT problem is the problem of finding a valuation for a SAT formula that satisfies the *maximum* number of clauses in

```
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];
}
```

Figure 9: sample program

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 9. 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.

4. 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.

4.1 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 [31]) 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.

4.1.1 *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 negatives. 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 [26,2]. 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, 14]. It is worth mentioning that a recent study [38] 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 2.4, the hypothesis is backed with the fact that a cause transition has happened at a location. Techniques using contrasting, Section 3.2.1 justify selection based on absence of a transition in a passing trace. Likewise, BugAssist in Section 3.2.3, 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 3.2.2, 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 [32,30] 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.

4.1.2 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.

Scalability

As we have seen in this paper, the techniques vary in their scalability. The methods based on model checking and symbolic execution, Section 3, 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.

4.2 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 [16]. Majority of proposed techniques attempt to find the wrong algorithmic component faults. Thus, there is a need to devise techniques to address these faults.

5. REFERENCES

[1] 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.

[2] 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.

[3] Baah, G. K., Podgurski, A., and Harrold, M. J. Mitigating the confounding effects of program dependences for effective fault localization. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (New York, NY, USA, 2011), ESEC/FSE '11, ACM, pp. 146–156.

[4] Ball, T., Levin, V., and Rajamani, S. A decade of software model checking with slam. *Communications of the ACM 54*, 7 (2011), 68–76.

[5] 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.

[6] Biere, A. *Handbook of satisfiability*, vol. 185. Ios PressInc, 2009.

[7] Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. Symbolic model checking without bdds. *Tools and Algorithms for the Construction and Analysis of Systems* (1999), 193–207.

[8] Bradley, A., and Manna, Z. *The calculus of computation: decision procedures with applications to verification*, vol. 374. Springer, 2007.

[9] Bryant, R. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on 100*, 8 (1986), 677–691.

[10] Chaki, S., Clarke, E., Groce, A., Jha, S., and Veith, H. Modular verification of software components in c. *Software Engineering, IEEE Transactions on 30*, 6 (2004), 388–402.

[11] Chaki, S., Groce, A., and Strichman, O. Explaining abstract counterexamples. *ACM Sigsoft Software Engineering Notes 29* (2004), 73–82.

[12] Clarke, E., Grumberg, O., and Peled, D. *Model checking*. MIT press, 2000.

[13] Clarke, E., McMillan, K., Campos, S., and Hartonas-Garmhausen, V. Symbolic model checking. In *Computer Aided Verification* (1996), Springer, pp. 419–422.

[14] 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.

[15] DeMillo, R. A., Pan, H., and Spafford, E. H. Critical slicing for software fault localization. *ACM Sigsoft Software Engineering Notes 21* (1996), 121–134.

[16] 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.

[17] Groce, A. Error explanation with distance metrics. *Tools and Algorithms for the Construction and Analysis of Systems* (2004), 108–122.

[18] Groce, A. *Error Explanation and Fault Localization with Distance Metrics*. PhD thesis, Carnegie Mellon University, 2005.

[19] Groce, A., Chaki, S., Kroening, D., and Strichman, O. Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer (STTT) 8*, 3 (2006), 229–247.

[20] Groce, A., Kroening, D., and Lerda, F. Understanding counterexamples with explain. In *Computer Aided Verification* (2004), Springer, pp. 318–321.

[21] 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.

[22] 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.

[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] Hutchins, M., Foster, H., Goradia, T., and Ostrand, T. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on Software engineering* (Los Alamitos, CA, USA, 1994), ICSE '94, IEEE Computer Society Press, pp. 191–200.

[25] Jhala, R., and Majumdar, R. Software model checking. *ACM Comput. Surv. 41*, 4 (Oct. 2009), 21:1–21:54.

[26] 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.

[27] Jose, M., and Majumdar, R. Bug-assist: assisting fault localization in ansi-c programs. In *Computer Aided Verification* (2011), Springer, pp. 504–509.

[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] Ko, A., and Myers, B. Debugging reinvented. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on* (may 2008), pp. 301 –310.

[31] Ko, A., Myers, B., Coblenz, M., and Aung, H. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *Software Engineering, IEEE Transactions on 32*, 12 (2006), 971–987.

[32] Ko, A. J., and Myers, B. A. Finding causes of program output with the java whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2009), CHI '09, ACM, pp. 1569–1578.

[33] Kroening, D., and Strichman, O. *Decision procedures: an algorithmic point of view*. Springer, 2008.

[34] Kurshan, R. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton university press, 1995.

[35] 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.

[36] 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.

[37] 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.

[38] Parnin, C., and Orso, A. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2011), ISSTA '11, ACM, pp. 199–209.

[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] 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.

[42] 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.

[43] ZELLER, A., AND HILDEBRANDT, R. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng. 28* (February 2002), 183–200.

[44] 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.