Survey of Software Fault Localization

Nimit Singhania
IBM India Research Laboratory
New Delhi, 110070, India
nimising@in.ibm.com

Abstract

I present a survey of automated techniques for software fault localization. Fault localization or localizing the root cause of failure is one of the most difficult processes in software debugging. Hence, many automated techniques have emerged to help in this process. Most of these techniques are based on the principles used in real life for fault diagnosis. I have used these principles to classify fault localization techniques

1 Introduction

Software debugging is one of the most time consuming process in software development and finding root cause of a failure is one of the most difficult process in debugging. So, many techniques have emerged to automatically localize fault in software. In this survey, I cover the major techniques explored for automatic software fault localization.

Software fault localization is not much different from fault localization in any other domain. Instead, we apply the same principles to diagnose software as in any other system. Some of these principles can be outlined as follows:

1. **Anomalous Behavior**: Faulty systems show many anomalies apart from the actual failing output. They tend to show behavior much different from the standard correct behavior. Thus, identifying such anomalies in the system behavior can help us gain better insight into the fault.

2. **Experimentation**: Experimenting with a system can help gain information about the fault.

3. **Dependence Analysis**: Systems have cause-effect chains. These cause effect chains can be traversed to find regions containing fault. For example, the fault has to be in the cause chain of a failing output and hence, we can restrict search for fault in the cause chain of the statement.

4. **Logical Deduction**: Given some information about a fault, many logical inferences can be derived about it, which may be useful to detect fault.

5. **Knowledge Reuse**: Previous knowledge about a system can be useful to locate the fault. This knowledge could be in various forms, like earlier experience with resolving faults or learning system specific properties etc.

Further, during any fault localization process, we perform the following steps. First, we look for clues or facts available about the fault in the system. Then based on these facts, we make some inferences about the fault. If the inferences lead to localization of fault, we are done. Otherwise, based on the inferences from previous steps, we try to gain new information from the system. This process repeats till we localize the fault.

I use the principles described above to organize the techniques explored for software fault localization.

2 Problem

Given a software that contains one or more faults, the objective of software fault localization is to localize code region that is most likely to contain fault. Here, some information about the bug may be initially present like a failing execution of the software, source code of the software, feedback from user about type of fault that occurs etc. Different techniques use different information about the fault. Given such information, the techniques pinpoint code regions that contain or are likely to contain the fault.

I would like to clarify some terminologies used here. $Failure$ represents a condition where the software either crashes or produces incorrect output in an execution of the software. A *fault / bug* represents code in the software that is the source of failure and thus needs to be modified. *Failing output* represents the location in the source code where failure is finally observed by the user. Hence, the aim of software fault localization is to locate code region that is likely to contain fault, given a failure or failing output. An $execution\ trace$ represents the sequence of statements executed in the corresponding execution of the software. A $failing\ trace$ corresponds to execution trace in an execution with failure and *correct / passing trace* represents an execution that is correct and does not show failure.

3 Techniques

I have divided the techniques for software fault localization into following categories, based on the principles described earlier.

3.1 Anomalous behavior

The principle used here is that a system shows anomalous or unusual behavior when it has a fault. Hence, apart from the actual failing output, the system also has other anomalies which can be used to detect the source of failure.

This is useful in software fault localization too. Hence, many automated techniques try to locate anomalies in the failing executions. How do we identify such anomalies? An anomaly in a failing execution is a feature of the failing execution that is rarely or never found in correct executions. Hence, the failing execution is compared with the correct executions to find such features and these features are reported back to the user.

Different techniques take into account different aspects of these executions, like set of statements executed, or the paths taken etc and use these to compare the executions. Hence, given an execution, this information is abstracted from the executions and is known as *program spectrum* [19].

In [14] and [15], authors give statistical techniques to find features that correlate highly with bugs or faults. [14] focuses mainly on fault localization in software with single faults. [15] extends [14] to handle large software containing multiple faults. Both the techniques focus on locating bugs in deployed software. Here, the code is first instrumented with numerous assertions or predicates and the values returned for these assertions in an execution form the program spectrum. The assertions include values of the branch predicates, sign of return values of functions, boundary checks on scalar variables etc and thus represent features of program execution. They are introduced into the software code either manually or automatically by compiler transformations. These assertions are then sampled during the executions of deployed software by users. However, sampling all assertions from an execution adds a lot of overhead and may impact performance of the software. So, as numerous executions of code occur by its users, the load of sampling assertions is distributed among users and only a small number of assertions are sampled in any execution. The sampling needs to be statistically fair and each assertion should have fair chance to be sampled in an execution. Hence, source code is further augmented to ensure this.

In [14], for deterministic bugs (bugs for which some assertions exist that are always true when the failure occurs), assertions predicting the bugs are found by eliminating the false assertions. For non-deterministic bugs (bugs that are not deterministic with respect to any assertion), it is a statistical problem of finding features of program spectra that best predict failing executions. This is equivalent to a machine learning problem of learning a binary classifier using as few input features as possible and is solved by statistical logistic regression.

In [15], the above technique does not work because, logistic regression tries to find least number of assertions that best describe *all* the failing executions, and not just ones that occur due to specific bugs. Hence, assertions that predict multiple bugs (super bug predictors) or ones that predict some specific cases of failure of a bug (sub predictors) get dominance. This problem is solved as follows. The authors define a probabilistic measure $Increase(P)$[1], for each assertion P, which determines the failure predicting power of the assertion. It is found that assertions with negative values of $Increase()$ do not predict failure and thus are eliminated. Now, the assertions that can expose bugs need to be found. Assertions that have high $Increase()$ values, correlate very well with failure. However, they may predict only a small set of failing executions (sub bug predictors) and hence, may not lead us to the bug. But, assertions that predict large number of failing executions (super bug predictors) are not specific to any bug and are not useful. We need a balance between failure predictive power ($Increase()$) and number of failing executions predicted. Hence, the assertions that have a good balance of both of these properties are found to predict bugs and are reported to the user.

Similar to above, given a set of passing and failing executions, [11] uses statistical techniques to present fault information visually in the source code. Here, set of statements in the source code executed in a program represent the program spectrum. The statements are colored by a color in the range red to green according to their probability of being executed in failing executions as compared to that in successful executions. Thus red statements are more likely to contain faults as they are frequently found in failing executions but rarely in successful executions and vice versa. Also, brightness of the statements is in proportion with the percentage of failing or passing test cases in which they are executed, whichever is higher. This highlights statements that are executed in majority of failing or passing executions.

In above techniques, all failing and passing executions are compared directly. However, many of these failures occur due to different bugs and thus, when software has multiple bugs, problems similar to [15] occurs. Even though [15] reduces these problems by using an appropriate ranking scheme, they are not completely eliminated. Hence, we now discuss a few techniques that take one failing execution at a time and compare it with correct executions to find anomalies. Here, first a model of correct behavior of the executions is built from the passing executions. Then, the failing executions are compared with this model and anomalies are reported to the user.

[23] uses time spectra to represent program executions and tries to catch regions of code that take unusual amount of time in the failing executions. By time spectrum, we mean information about time spent by an execution in different regions of the source code. This information is collected at the granularity of functions. As stated above, it first builds a $correct$ behavior model for the program using the time spectra of passing executions. It builds a behavior model for each function in the program. For each function in the program, it collects the distribution of time spent in the body and functions called within, from the passing executions. Then, it builds a statistical model on this data to get the required behavior model. Now, given a failing execution, for each method invocation, the time spectrum is computed and mapped to the corresponding function model. The invocations whose spectra are least likely to be found in correct program executions according to the model are reported back to the user.

[9] tries to find anomalies in the values of expressions in a program (values of expressions in a program represent the program spectra here). It maintains dynamic invariants ($correct$ model) for expressions at different program points. An invariant captures the set of values that are allowed for an expression in a correct program execution. All the expression values are first reduced to integers. Then, invariant is represented by two bits for each bit position in an integer. One bit stores the initial value of this bit and other stores if different values are allowed for this bit. In this way, it maintains precise information in the invariant when it accepts a small set of values, and only approximate information for larger sets. Given a set of correct and failing executions, these invariants are generated from the correct executions and then for each failing execution, the expressions whose invariants are violated are reported to the user.

[1]Increase(P) = Failure(P) - Context(P) where
Failure(P) = Prob(program fails | P is observed to be true)
Context(P) = Prob(program fails | P is observed)
Increase(P) basically represents the increase in probability of failure when P is observed true.

[19] uses comparison of failing executions with successful executions that are very $similar$ to the failing ones, rather than comparing them with arbitrary successful executions and hence executions that are irrelevant to the error. This may provide better results if a close enough successful execution is found for the failing execution. This technique defines $dissimilarity$ or $distance$ between executions using size of the difference in sets of statements executed by the executions. Using this distance measure, it finds a successful execution that has $minimum$ distance from given failing execution and reports the difference between them. [7] uses a different distance metric based on control flow. It first $aligns$ the statements in two different executions, so that every statement instance in one execution has a corresponding equivalent statement in the other execution. The distance metric is given by the set of branch statement instances which are aligned and have different outcomes in the executions.

[20] provides a technique to find the year 2000 bugs in program. This was one of the first techniques to use comparison of executions to point out regions containing fault. Here, set of intra-procedural loop free paths executed in the program execution are used as spectra. Given an input for which the program fails for a post 2000 execution, [20] returns paths that are executed in post 2000 execution but not in pre 2000 execution. The shortest distinguishing prefixes of such paths potentially contain the date dependent computation and hence the fault.

A few model checking techniques, use the idea of comparing executions to find source of failures in counterexamples produced by these tools. Model checking tools are used to verify certain properties of systems and whether these properties hold for all possible executions of the program. If not, they produce counterexamples which represents a possible execution that violates the given property. These counterexamples are generally lengthy and difficult to understand. Hence, additional information is needed to understand the cause of failure. An essential feature of these techniques is that information about all the executions of the program, is present in the model checking framework and hence, they have access to all possible executions of the program.

[2] reports a set of failing traces, each of which correspond to an error *cause*. An error *cause* is defined as a control flow transition that is found in a failing trace, but in none of the correct traces. Given a program, [2] first finds a failing trace in the program. If one is found, it finds control flow transitions in this trace that never occurs in the correct traces (This information is obtained from the model checking framework). It reports them along with the trace. Now, it searches for failing traces due to other cause transitions. For this, already found failure causing transitions are first blocked. Then, new failing traces are generated by model checking tool and their causes are reported. This process repeats till no new failing traces are found. [5] extends the idea of [19] to also report a successful trace closest to the failing trace (counterexample). Similarly, [6] produces multiple positive and negative traces that are related to the same error cause and then apply above techniques of comparing failing and successful executions to find faulty code regions.

The techniques in this domain have some limitations. The results of a technique in this domain are highly dependent on the type of spectra used, the type of comparison made between passing and failing executions : statistical ([15], [14], [11], [23], [9]) or distance based ([19]), and the type of fault present. There exist no general technique that works best with all types of faults. Instead, a fault, that can be caught by a technique in this domain, is best captured by a specific spectra and comparison type. If a coarser spectra is used, it may not show up as an anomaly. Similarly if a finer spectra is used, many false anomalies may show up along with the actual anomaly exposing the fault. Hence, given a failure, it is not possible to know whether a program spectra technique can catch the fault or what type of spectra and comparison of spectra should be used to catch it, unless we know some details about the type of fault present in the software.

3.2 Experimentation/Mutation based techniques

These techniques experiment with the system in order to find faulty regions. An example used in real life can explain the idea better. Consider a flashlight that is not working and we wish to locate the fault. One of causes of failure could be zero charge in batteries. To test this, we try replacing batteries with new ones or ones containing charge. If the flashlight starts working, it implies that batteries must have discharged. If not, we now know that the fault is in other parts of the flashlight. Now, either the bulb might have blown out or the problem is in the circuitry of the flashlight. To test this, we further try replacing the bulb and the process goes on till we reach the source of error. In this way, by replacing parts of the flashlight with correct components, we can locate the source of failure.

The basic idea here is that *removing the source of failure from the system shall remove failure as well*. However, at all times, changes to the system should be valid, and should not introduce new errors into the system.

Delta debugging[25] was one of the first few techniques to exploit this idea. It uses this idea to locate faults in GCC(GNU C Compiler). Given a failure producing input, it experiments changes in the input, so as to isolate the error into smallest possible region of the input. It does so by experimentally removing components of input and check if that removes failure. If yes, then the corresponding removed components contain the fault otherwise the remaining components of the input contain the fault. It recursively applies this technique to reach the smallest input component that induces failure. Also, it gives a failing and passing input (let them be f' and p' respectively), one containing failure inducing component and other not. It uses divide and conquer process to ensure that the number of experiments is at most quadratic in size of input program.

Once, a minimal input is found, it experiments changes to program states in GCC execution on f' and p'. Given a program point where the program states of passing and failing executions are similar, it tries replacing components of failing execution state with corresponding components from passing execution state to find components in the failing execution state that lead to failure. This is done at different program points and failure inducing program state components at different program points are reported to the user.

[17] improves upon [25] by taking hierarchal structure of input into account, while finding minimal failure inducing input. It applies technique in [25] first at the highest level in the hierarchy to induce failure inducing components at this level. Then, it moves to the next level of hierarchy of these failure inducing components and applies this technique. This goes on till we reach the lowest level. This technique ensures that all intermediate inputs are syntactically valid. Further, it also leads to fewer experiments as compared to [25] and hence is scalable to larger inputs.

Now, we look at a technique that experiments with the control flow path executed in a program execution, so as to obtain a path that removes failure. Given a failing execution of a program, [26] modifies the execution by switching predicate outcomes at branches and forcing the execution along alternate branch. It switches only one predicate outcome at a time and hence ensures that number of new executions produced are linear in number of branches in the failing execution. Last executed predicates before the failing output, are switched first. Further, priority is given to predicates that have dependences with the failing output and these are further prioritized according to their dependence distance from the failing output. This ensures that the number of switchings are least possible before we get a successful execution. Now, predicate whose outcome was switched to create a successful execution or led to removal of failure, is reported to the user. Further, statements that could effect the value of this predicate are also reported.

[3] experiments with the values of expressions in the program executions and tries to find expressions that are likely to contain fault. First, it tries to find expressions, for which some alternate values in failing executions leads them to pass. These are called repair candidates. After this, it eliminates repair candidates that are inflexible, or do not allow change in their value in some passing execution as it would break otherwise. This is because changes to an expression in a program lead to changes in value of the expression in most of its executions. Hence, after a faulty expression is fixed, passing executions are likely to get a different value for the expression. If a passing execution does not allow alternate value for an expression, then such an expression cannot be changed and hence cannot be a repair candidate. Thus, such expressions are eliminated. It finally reports all the remaining repair candidates

to the user.

In both these processes, it tries to find expressions that allow an alternate value in the execution. For this, it repeats the following process for different expressions and executions. Given an expression and a program execution, the expression is replaced by a symbolic variable, and then symbolic execution is used to find values that lead to successful program execution and are different from the original value of the expression.

This is a very useful technique that can give us a lot of information about the fault. However, experiments need to be conducted intelligently otherwise, it may happen that very little or no information is gained about the fault, even after numerous experiments.

3.3 Dependence Analysis

Systems, that have a sequence of events, have cause effect chains. Hence, the cause effect chains can be transversed to find more information about the error. For example, if we know that execution of a statement always leads to error, then the error is either in the cause or the effect chain of the statement and hence only these statements should be looked at to locate the fault. This idea is used by static and dynamic analysis techniques to find faulty code. We discuss here a few techniques in this domain .

$Static$ and *Dynamic Slicing*[1] are first few algorithms in this domain. A $backward\ static$ slice of a variable is the set of statements that can effect the value of variable at this program point in some potential run of program. Similarly a *forward static* slice of a variable is the set of statements that can be affected by the value of variable at this program point. Dynamic Slicing takes into account a specific run of program while computing slices. A $backward$ $dynamic$ slice of a variable at a point in execution is the set of statements in the execution that effect the value of variable at this point. Similarly a *forward dynamic* slice is defined. In general dynamic and static slices contain large number of statements and are not always useful to locate faulty code.

[8] integrates dynamic slicing and delta debugging [25] to find code regions containing fault. First, it finds the minimal failure inducing input using delta debugging. Now, the error is bound to be in the forward dynamic slice of this minimal input in the corresponding failing run. Similarly, error has to be in the backward slice of failing output variable. Hence, this technique computes the above slices and finds the intersecting region. This code region should contain the error and is output to the user.

Dynamic slice cannot capture errors due to omission of execution of some correct statements which happens because of incorrect outcome of the enclosing branch predicate. In such cases, the branch predicate of the omitted statements is not included in the dynamic slice and hence, the cause of failure which is the cause of incorrect branch outcome is not included in the dynamic slice. [27] addresses this by adding $implicit$ dependence edges to the dependence graph (A dependence graph describes the direct dependences among statements in a program trace). A statement u is implicitly dependent on a branch predicate p , if on switching the predict outcome, p affects the output at statement u i.e. in the new program run with outcome of p switched, either u is not executed, or p is in the dynamic slice of u.

[16] gives a static analysis technique to handle $typestate$ errors like NULL dereference etc. A typestate of a variable or object is given its current state in a typestate automaton. As different operations are performed on this object, its typestate changes according to the transitions of the automaton. For example, the following figure 1 represents the typestate automaton for NULL dereference errors. Initially, all pointers are in $uninit$ or uninitialized state. When an assignment occurs, their typestate transitions to $unsafe$ state. Then, a pointer dereference takes the object to $error$ state and all variables or objects in $error$ state are likely to have NULL dereference error.

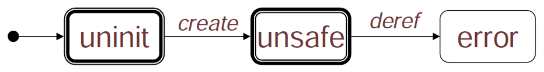

Figure 1: TypeState Specification for NULL Dereference [16]

Given a program point where the failure occurs and the type of error, [16] tries to locate the possible faulty statements that could lead to failure. It tracks the typestate information in backward direction. Given an object is in $error$ state, it tracks the typestate information in the backwards direction to find statements, where the actual error might have initiated. For example, in the case of NULL dereference, it tries to locate statements, where the initialization of NULL value to a pointer may have occurred. It tracks the typestates backwards, and tries to find statements, where transition from $uninit$ to $unsafe$ could have occurred. For this, it first reverses the typestate automaton. Then, it traverses the control flow graph from the error location to find possible locations where the error may have began.

[10] applies static analysis to a specific path in the program. Given a control flow path in the program, it tries to locate a subsequence of the path that is necessary for the execution of the path i.e. the path can be executed only if the subsequence is feasible. Such a subsequence is known as path slice. Formally, a path slice is a subsequence of the path such that if the slice is infeasible, path cannot be executed in any execution of the program, whereas, if it is feasible, the exact path may or may not be feasible, but the target location of the path is reachable. It is obtained by backward data-flow analysis, that takes only those statements that can control the path to the target location. Details can be found in [10].
This is helpful in analyzing counterexamples produced by many model checking and static analysis tools. It differs from dynamic slice in the fact that the given path may not be a feasible path and hence, not reproducible in any dynamic execution. Further, it is more specific than static slice, as it takes into account only statements in the given control flow path.

The strength of these techniques is that, in most cases, the fault is guaranteed to be in the slice reported. However, the slice produced is generally quite large and hence, may not be very useful.

3.4 Logical Deduction

The idea here is that many logical inferences about a system can be made, given some information. This is a process that is very important in any kind of problem solving and hence software debugging as well. An example (from [22]) will make the idea more clear. Consider the following program :

```
1.  func(x) {
2.       if(x==1)
3.           return 1;
4.       else
5.           return 0;
6.  }
```

Suppose we know some correct set of (input, output) possible for the above function are (0, 1) and (1, 0). However, the program fails on both of these test cases. Let us analyze the situation and try to understand possible causes for failure. From the first input output set where 0 is input and 1 is output, we can conclude that statement 2 or 5 must contain the fault. This is same as the set of statements in the backward dynamic slice of output. Similarly from the second test case, we can conclude that statement 2 or 3 must contain the error. However, from these two conclusions, we can logically infer that, either statement 2 is erroneous or statement 3 and 5 both are erroneous. This is because, if statement 2 is faulty, both the failures are explained. However, if statement 2 is not faulty, then 3 and 5 both have to be erroneous in order to explain the failures. Thus using logical inference we have arrived at two possible conclusions about error, which may not have been possible with other methods of fault localization.

This idea of logical inference is used by [22] to report sets of statements that can be erroneous. Given a program showing failure, it first builds a logical model of the dependencies in the program. For details about the model, please refer [22]. Now, given a set of correct input output pairs for which the execution fails (as in the above example), it reports back set of diagnoses. A diagnosis is a minimal set of statements, which if assumed to be incorrect can explain failure in all failing executions and represents a likely error. To find such diagnoses, it uses Reiter's theory([18]). First, it computes all possible conflict sets for the given input output pairs. A conflict set is a set of statements such that at least one of these has to be faulty in-order to explain the failure. For example, the set of statements executed in a failing execution has to be a conflict set. In the above example, {2,3} and {2,5} form conflict sets. Minimal hitting sets of these conflict sets are the required diagnosis and are reported to user. A *hitting* set of the conflict sets is set which has a non-empty intersection with each conflict set(in above example, {2}, {2,3}, {2,5}, {3,5} and {2,3,5} are possible hitting sets).

A *minimal hitting* set is a hitting set such that none of its subsets are hitting sets ($\{2\}$ and $\{3,5\}$ are only such sets in the above example). These are computed using techniques described in [18].

It has been proved that finding minimal hitting sets is an NP-Complete problem. Hence, techniques in this domain do not have a polynomial time algorithm and are quite expensive to use.

3.5 Knowledge Reuse

Here, the idea is that history of software maintainance and development contains a lot of information that can be used to debug software. We discuss a few techniques in this domain.

Bugs repeat in software. So, many of the bugs are known bugs and have been previously resolved in software. Hence, we can leverage previous bug fix information, and save effort in trying to debug the same bug again. The difficult task here is to identify previous bugs that match the current bug.

[12][2] uses software change history for this. It first builds a bug database, from the software changes in history that correspond to bugs. In any bug fix change, the deleted code chunk (bug chunk) is the one that contains bug and we can extract information about the bug from it. To extract this information, the bug chunk is parsed to get abstract syntax tree of the chunk which is then broken into components. Then, these components are processed to remove unnecessary and local information in the code. Language constructs like keywords and basic data types are removed. Local variables are replaced by their data types. To prevent name collisions, data-type is appended to the name of the variables. Field names are prepended with structure types. After this, the components that contain no useful information are discarded. These are then populated into a database. Now, given a source code containing bugs, it is first processed to obtain components as above, and then bug chunks with matching components are reported back to the user along with corresponding fix code chunks.

In [24], system call sequences made during an execution containing bug are used to identify it and match it with known bugs. First sequence of system calls made during a failing execution are traced. Then, this sequence is compared with traces from known bugs to see if it matches with any of them and the corresponding bugs are reported as possible causes.

Previously written code can be used to learn implicit programming rules (for example, a lock should be followed by unlock, and many such software specific rules not documented anywhere) and hence detect bugs due to violation of such rules. [13][2] uses this idea and provides a data mining technique to extract such implicit rules and the corresponding violations in the program. The functions in a software are converted into a set of items, each item representing unique elements in the functions like function calls, object references. The items are processed, similar to [12], so as to retain only items that contain unique information. Then, frequent item-set mining is applied on these function sets to obtain programming rules. Given sets of items, frequent item-set mining, tries to find subsets of items that are contained in many of these sets. The intuition is items that are found together frequently are likely to contain a programming rule. Thus, these item-sets are likely to contain rules. These rules (ex $a \Rightarrow b$ if b should occur whenever a occurs) are extracted from frequent item-sets by finding association among items and code regions that violate these rules are reported to the user

[21][2] uses a similar idea to learn rules for accessing objects in programs. Here, first the object access pattern in each function is extracted from their source code as object usage models (transition graphs representing possible sequence of instructions using the object). Then, these models are converted into item-sets and frequent item-set mining is applied similar to [13].

4 Conclusion

We have seen different principles for fault localization and their application in various software techniques used for automatic fault localization. These principles are orthogonal to each other and cover different aspects of fault lo-

[2]These techniques are more focussed towards finding bugs rather than locating root cause of failure. However, these could be helpful in fault localization as well. Further they elaborate the idea of knowledge reuse.

calization. Hence, we can explore combining these principles to build techniques that provide better fault diagnosis for software.

References

[1] Hiralal Agrawal. *Towards automatic debugging of computer programs*. PhD thesis, West Lafayette, IN, USA, 1992.

[2] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 97–105, New York, NY, USA, 2003. ACM.

[3] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 121–130, New York, NY, USA, 2011. ACM.

[4] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 342–351, New York, NY, USA, 2005. ACM.

[5] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error explanation with distance metrics. *Int. J. Softw. Tools Technol. Transf.*, 8:229–247, June 2006.

[6] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In Thomas Ball and Sriram Rajamani, editors, *Model Checking Software*, volume 2648 of *Lecture Notes in Computer Science*, pages 121–136. Springer Berlin / Heidelberg, 2003.

[7] Liang Guo, Abhik Roychoudhury, and Tao Wang. Accurately choosing execution runs for software fault localization. In *In CC*, pages 80–95, 2006.

[8] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 263–272, New York, NY, USA, 2005. ACM.

[9] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. *Software Engineering, International Conference on*, 0:291, 2002.

[10] Ranjit Jhala and Rupak Majumdar. Path slicing. *SIGPLAN Not.*, 40(6):38–47, 2005.

[11] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 467–477, New York, NY, USA, 2002. ACM.

[12] Sunghun Kim, Kai Pan, and E. E. James Whitehead, Jr. Memories of bug fixes. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 35–45, New York, NY, USA, 2006. ACM.

[13] Zhenmin Li and Yuanyuan Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 306–315, New York, NY, USA, 2005. ACM.

[14] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 141–154. ACM Press, 2003.

[15] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation.

[16] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. Pse: explaining program failures via postmortem static analysis. *SIGSOFT Softw. Eng. Notes*, 29(6):63–72, 2004.

[17] Ghassan Misherghi and Zhendong Su. Hdd: hierarchical delta debugging. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 142–151, New York, NY, USA, 2006. ACM.

[18] R Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987.

[19] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries, 2003.

[20] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *ESEC '97/FSE-5: Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 432–449, New York, NY, USA, 1997. Springer-Verlag New York, Inc.

[21] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 35–44, New York, NY, USA, 2007. ACM.

[22] Franz Wotawa, Markus Stumptner, and Wolfgang Mayer. Model-based debugging or how to diagnose programs automatically. In *IEA/AIE '02: Proceedings of the 15th international conference on Industrial and engineering applications of artificial intelligence and expert systems*, pages 746–757, London, UK, 2002. Springer-Verlag.

[23] Cemal Yilmaz, Amit Paradkar, and Clay Williams. Time will tell: fault localization using time spectra. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 81–90, New York, NY, USA, 2008. ACM.

[24] Chun Yuan, Ni Lao, Ji-Rong Wen, Jiwei Li, Zheng Zhang, Yi-Min Wang, and Wei-Ying Ma. Automated known problem diagnosis with event traces. *SIGOPS Oper. Syst. Rev.*, 40(4):375–388, 2006.

[25] Andreas Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10, New York, NY, USA, 2002. ACM.

[26] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faults through automated predicate switching. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 272–281, New York, NY, USA, 2006. ACM.

[27] Xiangyu Zhang, Sriraman Tallam, Neelam Gupta, and Rajiv Gupta. Towards locating execution omission errors. *SIGPLAN Not.*, 42(6):415–424, 2007.