Inferred Dependence Coverage to Support Fault Contextualization

Fang Deng
Department of Informatics
University of California, Irvine
Irvine, California 92617-3440
Email: fdeng@ics.uci.edu

James A. Jones
Department of Informatics
University of California, Irvine
Irvine, California 92617-3440
Email: jajones@ics.uci.edu

***Abstract*—This paper provides techniques for aiding developers' task of familiarizing themselves with the context of a fault. Many fault-localization techniques present the software developer with a subset of the program to inspect in order to aid in the search for faults that cause failures. However, typically, these techniques do not describe how the components of the subset relate to each other in a way that enables the developer to understand how these components interact to cause failures. These techniques also do not describe how the subset relates to the rest of the program in a way that enables the developer to understand the context of the subset. In this paper, we present techniques for providing static and dynamic relations among program elements that can be used as the basis for the exploration of a program when attempting to understand the nature of faults.**

I. Introduction

Recent years have seen a plethora of techniques proposed by researchers for identifying likely locations of faults in software. These fault-localization techniques use information about how the software was executed, both when it behaved as expected and when it behaved erroneously to suggest locations in the program whose execution correlate with software failure. These techniques suggest locations or sequences of locations that are most likely to comprise the fault. A typical scenario is that a programmer starts from one of the results of the fault-localization technique (whether a location, such as a statement or branch, or a sequence of statements, such as a acyclic path or disjoint path). From there, the developer must build an understanding and context of that result to help determine if this was the correct result of the localization or if this result was a false positive and the next should be utilized. Programmers rely on their experience and intuition to guide their continued exploration.

The result of fault-localization techniques is a (sometimes ranked) list of locations or sequences of locations within a program that are identified as potentially faulty. Several techniques consider these locations to be "*suspicious* of" or "*blamed* for" causing test-case failures. Often, these suspicious locations are scattered throughout the program.

Results from techniques such as TARANTULA [8], [9] or SBI [11] label source-code components belonging to multiple functions, classes, and files as suspicious. The source-code entities at these potentially disparate locations may or may not be related. Often the relationship among these locations in the code is not immediately obvious — it is the developers' job to manually inspect and determine their relations and the functionality that it describes. Upon inspection, the developer may determine that the identified source-code entities are, indeed, related. Some ways in which we have witnessed these relations are through control relationships (e.g., a suspicious method call to a suspicious method) or through data relationships (e.g., a global variable is defined at a suspicious location and referenced at another suspicious location). In our experience with these techniques, we have found that the suspicious locations often constitute a logical coherence, performing some particular functionality involved in the failures despite the code being scattered.

More recently researchers have proposed failure-correlated paths (e.g., [2], [3], [6], [7], [12]), which can provide some context in aiding the understanding of faults. In these works, sets of sequences or paths of execution are presented to the developer as a set of potential fault diagnoses. These techniques perform fault localization while giving context within the diagnoses. In this spirit, we wish to extend and further inform these forms of context by providing more explicit relationships among the locations within the diagnoses, guidance to understand the relationships and context around the diagnoses.

It is this problem that we are addressing in this work: contextualizing the fault diagnosis from any automated fault-localization technique and supporting exploration, both within the diagnosis and around it — a process that we are referring to as *fault contextualization*. To support this task, we propose a technique to augment static dependency with dynamic coverage information in an effort to provide guidance to the developer (or automated client analyses) as to which static relationships are most related in actual execution. The coverage inference is realized by utilizing lightweight information gathered from test cases. We envision that the hybrid static/dynamic model could serve as an augment to the traditional debugging interfaces, enabling developers to determine the context of the faults and the relations among the failure-correlated code.

II. Background

To provide the necessary background to motivate and explain our approach, we overview some existing static analyses (Section II-A), an existing fault localization technique (Section II-B), and the existing models for exploring a program given a fault-localization result (Section II-C).

A. Program and System Dependence Graphs

The model of the program by which we determine which instructions in a program are related to the other instructions is the *system dependence graph*. To describe a system dependence graph, we first describe the *program dependence graph* (or, *PDG*) on which it is built. The program dependence graph [15] is an intraprocedural model of a procedure. It captures both control and data dependencies among instructions within the procedure. Control dependencies capture the relationship between two instructions when one conditionally guards execution from another. In other words, if instruction A may not be executed because of a predicate in instruction B, A is said to be control dependent on B. Data dependence captures the relationship between two instructions when a value may propagate from one to another. For example, if instruction C has a variable assignment (e.g. `x=5`) that over some path can reach, without redefinition, instruction D that has a reference of that variable (e.g. `print(x)`), D is said to be data dependent on C for the variable `x`. The PDG for a procedure is composed of all instructions, represented as nodes, and all control and data dependencies, represented as edges.

The system dependence graph (or *SDG*) [5] enables efficient analysis and slicing of dependencies across procedural boundaries. The SDG is an interprocedural model that is built on the basis of the PDGs of individual procedures. In this graph, nodes are the instructions of the program, and edges are the control and data dependence relationships among the instructions. In addition, additional edges, called *summary* edges, are introduced to enable accurate graph walking and slicing to account for procedure calling context. In this discussion, we will focus primarily on the control and data dependence edges.

B. Statistical, Coverage-based Fault Localization

Our contextualization approach is applicable to the results of any fault-localization technique (including manual ones). However, we briefly describe one such technique upon which our implementation is built. A plethora of statistical fault localization techniques have been proposed by researchers. One such technique, TARANTULA [9], utilizes whole test suites (or any subset thereof) to infer likely locations for faults based upon the relative participation of the passing and failing test cases and the events that occurred during execution. This work originally was presented to target instruction-level coverage, and that is the level at which we will apply our localization.

The main insight of each of this (and many other) coverage-based techniques is that, when running a test suite, execution events that correlate with failures are more likely to be the cause (i.e. fault or bug) of those failures. Said differently, events that occur mostly in failing test cases, but rarely in passing test cases, are more *suspicious* of being the fault. This inferencing examines the event similarities among the failing test cases and differentiates those similarities from the events occurring in the passing test cases.

This original approach has been improved by the replacement of the original suspiciousness metric with a similarity coefficient borrowed from the molecular biology field. Abreu and colleagues [1] proposed the use of the Ochiai coefficient. The equation for Ochiai can be represented as

$$suspiciousness(i) = \frac{failed(i)}{\sqrt{totalfailed * (failed(i) + passed(i))}} \quad (1)$$

where *passed*(i) is the number of passed test cases in which instruction i is executed, *failed*(i) is the number of failed test cases in which i is executed, and *totalfailed* is the number of failed test cases in the test suite.

By assessing the *suspiciousness* of each instruction of the program, we can identify those instructions that are the most suspicious of being faulty to begin our search for the true cause of the testing failures (i.e., fault).

C. Traditional Models of Exploration

Multiple techniques have been proposed by researchers to extend the search from the predicted location of the fault (or most suspicious) to the actual fault. Jones and colleagues [8] proposed a process of examining the instructions in the program in the order of decreasing suspiciousness: that is, first examine those instructions that are most suspicious, then examine the next most suspicious instruction, and so on, until the fault is found and understood. In contrast, Renieris and Reiss [16] and Cleve and Zeller [4] proposed a process of examining first those instructions that are predicted to be the location of the fault, and then extending that search through the use of the system-dependence graph. This SDG exploration proceeds in a breath-first fashion from the predicted fault locations across all dependency edges in concentric rings. Figure 1 depicts this process, where the developer starts his search for the fault at the predicted fault location in the center node, and then examines all nodes in the first concentric ring (all at a distance of 1 from the predicted fault), and then to the next concentric ring where the fault is found. Color is used in this figure to represent potential relevance to the propagation of the fault's effects and to demonstrate how irrelevant nodes may be examined in this process.

Both of these exploration models (in the order of decreasing suspiciousness and in the breadth-first order of the SDG) are simplified models of actual human behavior given the results of the fault-localization techniques. Neither were meant to be prescriptive or even as a guideline or advice — their purpose is only to facilitate a quantitative evaluation of fault-localization techniques.

In each of these exploration models, we can identify some key ways in which they are not likely to be an accurate

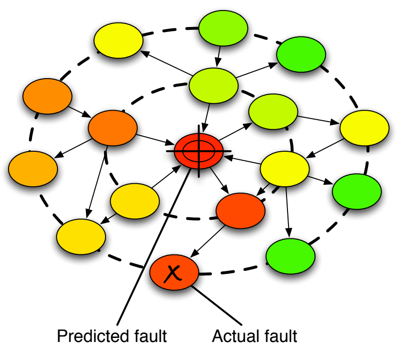

Fig. 1: Order of search as defined by Cleve and Zeller from the predicted location of the fault and proceeding outward in a breadth-first search. Figure is adapted from Reference [4].

model of human behavior. In fact, the authors of each of the articles that introduced of these models admitted as much as a threat to validity. With the decreasing-suspiciousness model, the structure of the code is ignored. This inattention to the structure of the code may cause developers to examine instructions in isolation and to jump their search from one location in the program to another, perhaps disparate location in the program. The order prescribed by a strict most-to-least suspiciousness sequence is likely to be unrealistic as it ignores the need for the developer to understand the surrounding code and the relationships both with the immediate surrounding code in the code listing and those that exist across modules.

Conversely, with the breadth-first SDG model, the structure of the code is the only contributing factor to the search. The developer is more likely to explore in a fashion that follows similar code features. Remember, that the SDG is a static model of the program, which means that the model over-approximates the possible dependencies that occur in practice. In practice, many of the dependencies are never or rarely fulfilled.

III. Approach

To address some of the key limitations of each of the exploration models described in Section II-C, we developed a new model that bridges the static representation of the SDG with dynamic information. The goal of this new hybrid model is to serve as a foundation for interfaces that can help guide developers in their search for faults and in their exploration to *understand* those faults: a process that we are calling *fault contextualization*.

Our approach leverages both the static information from an SDG and the dynamic information about the program and its execution. The general idea of our approach is to inform the SDG of a faulty program with dynamic coverage information by assigning each dependency (i.e., an edge in the SDG) with a variable weight. The weight implies the degree to which a dependency is actually covered in executions. The weight provides additional information by conveying which path may be more likely to have propagated either the effects of a fault or the necessary preconditions to trigger the fault. Unlike dynamic slicing, the weight of the dependency is inferred by considering the coverage of all test cases rather than a single execution path. Thus, the SDG is able to convey the execution information related to an entire test suite through a single model.

To calculate the weight, the traversal or realization of control and data dependencies need to be gathered from executions, either by direct instrumentation (i.e., modifying the program to keep track of which were executed) or by inference from the instruction-level coverage that is currently being gathered for the fault localization. Direct instrumentation of control and data dependencies can be done, however, this type of instrumentation is not commonplace in practice. One reason is that data-flow instrumentation is known to impose a higher runtime overhead on the subject program [13], [17], and is subject to complications such as aliasing [10], [14]. Instruction-coverage instrumentation is much more common in software-development practice, as it is used to determine white-box testing adequacy. It imposes a low runtime overhead, and many tools currently exist to gather this information about a program. As such, in our approach, we infer the traversal of a dependency from instruction coverage.

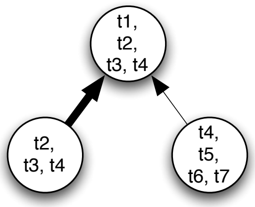

Fig. 2: Three instruction nodes labeled by the test cases that covered them. A dependency edge whose incident nodes have a higher set similarity of the test cases that executed them is treated as more likely to have been executed.

To infer dependency traversal from instruction-level coverage, each node can be labeled with the test cases that executed it (i.e., the instruction that it represents). A set similarity metric is utilized to determine a relative likelihood that that dependency was executed. For example, in Figure 2, each SDG node is labeled with the test cases that executed it. The dependency from the node labeled “t2,t3,t4” to the node labeled “t1,t2,t3,t4” can be thought to be more likely to have been executed than the other because the set of test cases on each node incident on that dependency edge is more similar. In our approach, we use Jaccard similarity coefficient to compute the set similarity, presented in Equation 2. In Equation 2, A and B represent the sets of test cases that executed the two nodes of an edge respectively.

$$J(A,B) = \frac{|A \bigcap B|}{|A \bigcup B|} \tag{2}$$

We then model the weight of an edge using the set similarity as the weight. As a result, some dependencies are assigned a greater weight than others. A greater weight of dependence can be interpreted as a shorter distance between the two nodes that the dependence connects. Users can use the distance to determine which node should be visited first before others when familiarizing themselves with the context.

IV. Experience

We evaluated the effectiveness of our approach in terms of its ability of clustering failure-correlated effects. We extracted SDGs of multiple faulty versions of two C programs — each program was over 7KLOC, and each version contained one fault. We executed test suites on the programs and captured instruction coverage. We then used our approach described in Section III to infer the weight of each dependency in the SDGs. We also calculated suspiciousness of all instructions using the approach described in Section II-B. As a result, we observed in both programs that nodes (i.e., instructions) with similar suspiciousness formed clusters in terms of distance (the inverse of weight). And, we found that the clusters were not obvious if we assigned each dependence with a same value (i.e., if no dynamic coverage information was used). This phenomenon provides initial evidence that our approach is effective on clustering failure-correlated effects which are scattered across various locations in the source code and thus informing and guiding developer attention in their debugging investigation.

V. Conclusions

In this paper, we presented a technique that augments a static-analysis model of the program—the system-dependence graph—with dynamic information gathered from the execution of the program under test. The dynamic information is gathered from commonplace, lightweight instruction-level coverage, and that is used to infer coverage of program dependences. The result of the technique is a static/dynamic hybrid model of the program. Some dependencies are assigned a greater weight than others, which may be used to inform and guide developer attention when investigating and understanding faulty behavior to ultimately debug the program.

Our experience shows that the technique and model successfully highlight and associate relevant and failure-related program context, even when that context is scattered across various locations in the source code. Our hybrid model performed more effectively to cluster the failure-related context than the static model, and has thus demonstrated that it is a suitable model on which to build contextualization client applications.

In future work, we will extend our studies to include more programs and faults. We will also investigate the factors that enable only some faults to be clustered independently. In addition, we will develop client analyses and applications that support program exploration, contextualization, querying, and identification of failure-related cross-cutting concerns in a way that can leverage dynamic information from the entire test suite.

VI. Acknowledgments

We thank the anonymous reviewers for their valuable and thorough comments. This material is based upon work supported by the National Science Foundation under award CCF-1116943, and by a Google Faculty Research Award.

References

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques*, pages 89–98, Washington, DC, USA, 2007. IEEE Computer Society.

[2] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan. Identifying bug signatures using discriminative graph mining. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 141–152, New York, NY, USA, 2009. ACM.

[3] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 34–44, Washington, DC, USA, 2009. IEEE Computer Society.

[4] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 342–351, New York, NY, USA, 2005. ACM.

[5] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12:26–60, January 1990.

[6] H.-Y. Hsu, J. A. Jones, and A. Orso. Rapid: Identifying bug signatures to support debugging activities. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 439–442, Washington, DC, USA, 2008. IEEE Computer Society.

[7] L. Jiang and Z. Su. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 184–193, New York, NY, USA, 2007. ACM.

[8] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM.

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

[10] W. A. Landi. *Interprocedural aliasing in the presence of pointers*. PhD thesis, Rutgers University, New Brunswick, NJ, USA, 1992. UMI Order No. GAX92-19944.

[11] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 15–26, New York, NY, USA, 2005. ACM.

[12] W. Masri. Fault localization based on information flow coverage. *Softw. Test. Verif. Reliab.*, 20:121–147, June 2010.

[13] J. Misurda, J. A. Clause, J. L. Reed, B. R. Childers, and M. L. Soffa. Demand-driven structural testing with dynamic instrumentation. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 156–165, New York, NY, USA, 2005. ACM.

[14] A. Orso, S. Sinha, and M. J. Harrold. Effects of pointers on data dependences. *International Conference on Program Comprehension*, 0:0039, 2001.

[15] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Not.*, 19(5):177–184, 1984.

[16] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, ASE '03, pages 30–39, Montreal, Quebec, October 2003.

[17] R. Santelices and M. J. Harrold. Efficiently monitoring data-flow test coverage. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 343–352, New York, NY, USA, 2007. ACM.