2017 IEEE/ACM 39th IEEE International Conference on Software Engineering Companion

Debugging with Probabilistic Event Structures

Ezekiel O. Soremekun
Saarland University, Germany.
soremekun@cs.uni-saarland.de

***Abstract*—Debugging is a search process to find, understand and fix the root cause of software defects. Can debugging benefit from probabilistic information? We hypothesize that debugging activities can benefit from probabilistic information that capture the statistical dependence of program features and the minor variations of program behavior. This probabilistic information helps to guide the search for the root cause of the bug and provides detailed diagnostic information (such as failure-inducing inputs and method calls leading to the fault). To realize our hypothesis, we propose to improve debugging activities by guiding bug diagnosis using both probabilistic reasoning and program analysis. The main idea is to mine probabilistic information from program executions, then apply these information to construct probabilistic event structures (e.g. probabilistic call graphs) that guides debugging activities such as fault localization and comprehension. The resulting probabilistic model will guide bug diagnosis towards the most likely paths to the root cause of bugs and provide contextual diagnostic information.**

I. Research Problem

Most human activities in software engineering are essentially *search processes* that inspect software artifacts, in order to find specific program features and properties. These activities include tasks such as testing and debugging. In a typical software development scenario, testing is the search process that exposes the bug (wrong behavior), while debugging is the process that diagnoses the root cause of the bug and searches for fix candidates to correct the wrong behavior.

In the last three decades, researchers have developed numerous techniques to support developers when debugging [1]. These techniques fall into two major classes: *program analysis* and *probabilistic reasoning* techniques. *Program analysis* techniques such as tainting [2] and slicing [3] are popular debugging aids that employ either static or dynamic analysis to support debugging activities. Meanwhile, *probabilistic reasoning* provides a means to capture and analyze uncertainties in program behaviors. Probabilistic reasoning is currently used in debugging for *statistical fault localization*, this is the process of finding the defective statement in the source code that caused the bug (e.g. Tarantula [4]). A common characteristic of both approaches is that they are *specification mining* approaches, meaning they derive a specification of the program in terms of the relationship between program features (e.g. executed program statements) and program failure.

However, developers hardly use existing debugging aids [5]: Both statistical tools and program analysis tools are infrequently used in practice. In particular, our survey of 180 professional software practitioners showed that the majority of developers rely on traces and interactive debuggers, but never use slicing, algorithmic debugging, or statistical debugging [5].

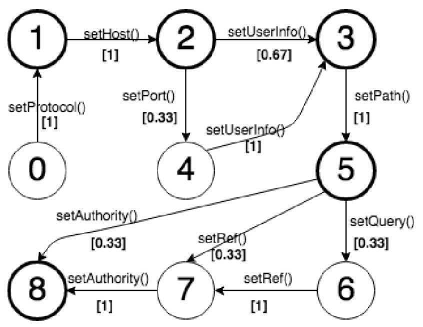

Fig. 1. Probabilistic Call Graph derived from `java.net.url` processing the inputs in Figure 2. Probabilities are enclosed in [...], Vertices are program states and Edges are method calls. Thick vertices (i.e., $\{1,2,3,5,8\}$) are mandatory program states.

```
http://user:password@www.google.com:80/command?
    foo=bar&lorem=ipsum#fragment
http://usr:pswd@www.cnn.com/worldcup#result
ftp://bob:12345@ftp.example.com/oss/debian7.iso
```

Fig. 2. Sample URL inputs (adapted from [8])

We also observed that developers desire more *sophisticated bug diagnosis* tools, they desire tools that can provide a high-level diagnosis of the pertinent sequence of events leading to the error (as cause-effect chains or a narrative).

Overall, we observed that developers seek advanced bug diagnosis techniques beyond fault localization; they desire both probabilistic (e.g. suspicousness rank) and contextual information (e.g. sequence of method calls) [5]. A major limitation of statistical debugging is that they lack the necessary contextual information (e.g. method calls) needed to understand program behavior. Meanwhile, program analysis techniques do not account for uncertainties (e.g. inconsistencies) in program behavior [6]. However, uncertainties are relevant for bug diagnosis, because debugging is inherently probabilistic [7]. For instance, a common source of uncertainty when debugging is incomplete oracles; where developers are equipped with partial information about program correctness.

The key insight of this proposal is that program analysis is powerful in capturing contextual information, whereas probabilistic reasoning can handle uncertainties. Hence, the main idea is to *apply probabilistic reasoning to guide the search for faults when debugging, and employ program analysis to provide contextual information required to understand the program behavior*. Specifically, we leverage the strengths of both program analysis and probabilistic reasoning, by applying *probabilistic specification mining* to diagnose software failures. We aim to construct *probabilistic event structures* (e.g. probabilistic call graph in Figure 1) that assign probabilities

978-1-5386-1589-8/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE-C.2017.34

to program events (such as method calls), in order to capture uncertainties in program behavior. The goal of this work is to develop new debugging aids that meet developers' need for more detailed diagnosis. Our proposed debugging aids are expected to provide both program features (e.g. input fragments) that are statistically dependent on a failure and the program behavior (e.g. method calls) that lead to such failures.

To illustrate our hypothesis, consider the probabilistic call graph (cf. Figure 1) mined from the execution of `java.net.url` processing the URL inputs in Figure 2. In this graph, the probabilistic information encodes the *probabilistic occurrence* (frequency) of each method call and the observed *probabilistic order* (sequences) of method calls. Hence, if `java.net.url` successfully parsed all URL inputs except for the first URL, then our approach would provide the features that are statistically dependent on the failure (such as the input fragment `port=80`), and the program behavior leading to the fault (such as the program state transitions $\{2 \rightarrow 4, 4 \rightarrow 3\}$ and method call sequence {`setPort()`→`setUserInfo()`}) to the developer. This diagnostic information is derived from the observation that these program features are statistically different from the features observed for most inputs.

II. Background

A. Debugging

In the last three decades, there have been over 400 research papers proposing different debugging aids to support programmers in locating and fixing software faults [1]. Most of these publications fall into the category of *statistical debugging* — an approach that identifies faulty program elements by correlating program executions with failures [4], [9], [10]. However, most statistical fault localization tools do not satisfy the debugging needs of developers. A recent study [11] examined the crucial factors (e.g. trustworthiness, scalability and efficiency) that determine practitioners' adoption of a particular statistical fault localization tool. The study revealed that the debugging aids proposed in the last 5 years do not satisfy at least 75% of developers' requirements. Meanwhile, program analysis tools for debugging are criticized for not ranking the statements presented to the developer for inspection [6]. For instance, program slices can become very large [12], which makes fault localization inefficient when the fault is located relatively far away from the bug symptoms.

In a recent human study [5], we found that contextual information (such as control dependence) are important for code comprehension when debugging. Besides, we observed that one in five developers are interested in tools that help at program understanding and that better program understanding leads to more correct patches. In this study, we identified two major limitations of existing debugging aids in practice:

- Neither statistical debugging nor program analysis based debugging tools provides *sufficient support for developers when debugging*;
- Developers need both *suspiciousness (probabilistic) information* and *contextual information* (e.g. program dependence and software specification) when debugging.

Hence, we propose to *synergistically combine probabilistic reasoning and program analysis* to develop debugging aids that not only identify faulty program features, but also provide contextual information (e.g. method calls leading to failure).

B. Specification Mining

Specification Mining [13] provides a means to abstract the behaviors of the software through *dynamic analysis* of program executions. Generally, specification mining involves *trace generation* and *specification construction.* In the trace generation phase, ordered traces of events that represent possible execution of the program are collected. Then, in the specification construction phase, the set of event traces are analyzed to obtain a single *specification* that models the traces.

Typically, *mined specifications do not capture uncertainties — the minor behavioral variations of the program — that is required for activities such as debugging* [6]. For instance, this approach is employed to dynamically mine program invariants from program traces [14]. While program invariants can identify properties that hold and must be preserved for observed traces, they do not capture uncertainties (e.g. inconsistent behavior). Thus we propose to apply *probabilistic specification mining* [13] to dynamically extract minor behavioral variations, in order to capture the likelihood of events occurring in program runs. For instance, the grammar specification obtained from `java.net.url` processing the URL inputs in Figure 2 would only characterize the grammar of the `protocol` parameter as ('`http`' | '`ftp`'). Meanwhile, a probabilistic specification could also provide the likelihood of occurrence of the `protocol` in program runs — ('`http`' {0.67} | '`ftp`' {0.33}) (cf. Figure 3): This is useful for diagnosing wrong behaviors involving minor (least occurring) input fragments like '`ftp`'.

III. Proposed Approach

In this section, we first introduce our preliminary work on *hybrid fault diagnosis* [15], then we present the application of *probabilistic grammar models* in debugging. In both approaches, we propose to collect probabilistic information (such as feature probabilities) from program runs in order to improve the program specification obtained from dynamic analysis.

A. Hybrid Fault Diagnosis

In this work [15], we plan to evaluate the two major fault localization approaches, namely statistical fault localization and dynamic slicing. Both approaches are essentially specification mining tools for debugging: Statistical debugging mines a specification of the program using multiple runs, in order to characterize the mapping between program features (e.g. statements) and failures. Meanwhile, slicing provides a specification of the program for a specific failing execution. However, we observed that both techniques are limited in one way or another: Statistical debugging lacks contextual information and dynamic slicing does not capture uncertainties.

Thus we propose a *hybrid technique* which serves as a probabilistic specification mining approach that harnesses the power of both statistical correlation and dynamic analysis.

```
URL ::= PROTOCOL {1.0} '://' AUTHORITY PATH {1.0} ['?'
        QUERY {0.33}] ['#' REF {0.67}]
AUTHORITY ::= [USERINFO {1.0}  '@'] HOST {1.0}
              [':' PORT {0.33}]
PROTOCOL ::= 'http' {0.67} | 'ftp' {0.33}
USERINFO ::= /[a-z]+/ {1.0}  ':'  /[a-z0-9]+/ {1.0}
HOST ::= /[a-z.]+/ {1.0}
PORT ::= '80' {1.0}
PATH ::= /\/[a-z0-9.]*/ {1.0}
QUERY ::= 'foo=bar&lorem=ipsum' {1.0}
REF ::= /[a-z]+/ {1.0}
```

Fig. 3. Probabilistic Grammar Model derived from `java.net.URL` processing the inputs in Figure 2 using AUTOGRAM (adapted from [8]). Probabilities are enclosed in {...}; Optional parts are enclosed in brackets [...] ; regular expression shorthands are enclosed in /.../.

The hybrid approach proceeds in two phases. It first reports the N most suspicious statements, obtained from the ordinal ranking[1] of a statistical fault localization technique. Then, if the fault is not found, it reports the symptom's dynamic backward dependencies. This approach overcomes the weakness of dynamic slicing by first giving probabilistic reasoning a chance at determining the faulty statements. On the other hand, it enables a developer to be more efficient by inspecting only a few suspicious statements, then proceeding to more contextual information obtained from program dependence analysis.

B. Probabilistic Grammar Models

Main Idea: In this work, we leverage the power of program analysis and probabilistic reasoning to improve the state-of-the-practice in debugging. Specifically, given a failing execution, we aim to accomplish the following tasks: First, we want to provide high-level contextual information (such as the sequence of method calls) that were responsible for the failure (cf. Section III-B1). Secondly, we want to highlight the input fragments that were statistically dependent on the failure (cf. Section III-B2). Lastly, we want to expose patches that mask incorrect behaviors, especially when developers are equipped with an incomplete test oracle (cf. Section III-B3).

Approach: Our approach works in three major steps (cf. Figure 4). First, given the program and a set of inputs, a grammar miner (e.g. AUTOGRAM [8]) produces a *context free grammar* describing the input structure of the program. Secondly, our *probabilistic miner* extracts the statistical information of program features from the program executions to produce *feature probabilities* of the observed runs. Then, our *probabilistic grammar constructor* produces a *probabilistic grammar model* [16], derived from both the feature probabilities and the input grammar. The resulting probabilistic model identifies the failure-inducing input fragment and method calls.

Example: Given the input URLs provided in Figure 2 and a program that parses a Uniform Resource Locator (URL) such as `java.net.URL`, the grammar miner (e.g. AUTOGRAM) mines the program's input specification by aggregating the input fragments handled by the same function into lexical and syntactical entities as shown in Figure 3 (except the probabilities encoded in braces {..}). Then, the probabilistic grammar constructor would include probabilistic information (e.g. frequency of features) collected from the program runs

[1]In ordinal ranking, lines with the same score are ranked by line number

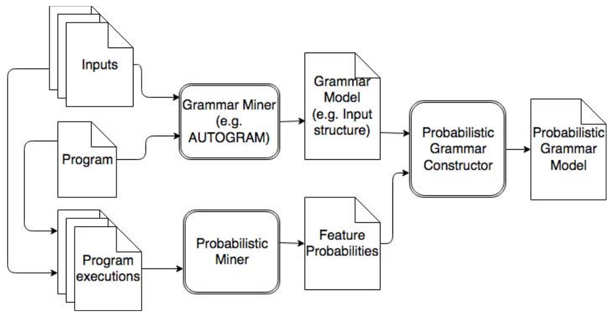
Inputs
Program
Program executions
Grammar Miner (e.g. AUTOGRAM)
Probabilistic Miner
Grammar Model (e.g. Input structure)
Feature Probabilities
Probabilistic Grammar Constructor
Probabilistic Grammar Model

Fig. 4. Workflow of the Probabilistic Grammar Mining Approach

in the grammar model, this is encoded in braces ({...}) in Figure 3 and square brackets ([...]) in Figure 1. In this example, the probabilities reflect the likelihood of the program features in a program run and can be easily applied to detect minor behavioral variants common to program failure.

In the following, we illustrate the application of the resulting probabilistic models (cf. Figure 1 and Figure 3) for the automated guidance of the following debugging tasks:

1) Grammar-based Debugging: Non-probabilistic grammar models can not be directly applied to find inconsistent behaviors, because they do not capture the minor variations in input processing that lead to failures. Thus, in our work, we propose to adapt input grammar for debugging purposes by augmenting it with the strengths of probabilistic reasoning. We intend to achieve this by using a grammar miner to identify input fragments and program features that correlates with failure. For instance, given the probabilistic model in Figure 3, if the execution of the first URL input in Figure 2 leads to a failure and we observed from our input grammar that all other URLs without a `query` or `port` are correctly parsed; then, we can easily provide sophisticated debugging information containing the inconsistent call sequences in the program's execution (i.e. state transitions {2,4,3} and {5,6,7}) and the likelihood that the `setQuery()` and the `setPort()` methods are variant (uncommon) calls. This debugging information is derived from the observation that these method calls statistically depend on the failure. We expect this diagnostic information to improve code comprehension and debugging effectiveness.

2) Input Analysis: We intend to use the probabilities assigned to input fragments and their feature dependencies to determine the parts of inputs that result in failure and how to correct such inputs if they are (syntactically) wrong, or to determine the program features that incorrectly handle the input fragments of correct inputs. This is inspired by our observation that developers perform *input manipulation* to understand program behavior and to detect the root cause of bugs [5]. For instance, in a run-time monitoring scenario with a representative test suite, a URL manipulation attack such as: `http://guardian.co.uk/login.asp?userid=admn%27%3b%20update%20logintable%20set%20passwd` should be flagged as suspicious, since we observed that the `query` input fragment (i.e. url fragment after terminal "?") is statistically unlikely in our grammar. Besides, input analysis can be handled for external resources such as configuration options, files and network sockets. This is particularly important because one in seven developers are interested in determining the most general conditions that lead to program failure when debugging [5]. In addition, we intend to develop an *input rectification* approach that leverages our probabilistic grammar models to determine wrong input fragments from failed executions and recommend their corrections based on the grammar of observed correct input fragments.

3) Patch Evaluation: In debugging, passing all test cases is insufficient to ensure patch correctness, because some patches mask incorrect behavior. In a recent human study [5], we observed that one in three human-generated patches are incorrect but *plausible* (passes all tests). However, such plausible patches exhibit inconsistent behavior that can be captured by our probabilistic grammar model. Hence, we intend to develop a *probabilistic test oracle* that quantifies the *degree of correctness* of programs (e.g as a deviation from mandatory program states) and provides *witnesses of inconsistent behavior* by capturing the uncertainties (e.g. inconsistent call sequences) in the program behavior during testing. This oracle would collect uncertainties in program runs and map them to program features. For instance, consider a scenario in which `setPort()` method (in Figure 1) was buggy: it failed for the first URL input (in Figure 2) and a developer provides a plausible but incorrect patch that fixes the symptom of the bug. If the patch stems a new state transition ($\{2, 4, 5\}$) with call sequence (`setPort(), setUserInfo()`), then our probabilistic oracle would provide the developer with transition $\{2, 4, 5\}$ as the witness of the inconsistent behavior, and the absolute deviation from the mandatory program states ($= 1 - \frac{1}{5} = 0.8$) as the patch's degree of correctness.

IV. Evaluation

In our evaluation of the hybrid fault diagnosis approach [15], we implemented dynamic slicing [3] and statistical debugging [10] for both CoREBench [17] and the Siemens benchmark[2]: We found that the best results are obtained by our hybrid approach. Besides, in a recent human study [5], we have provided DBGBENCH; this is a benchmark that provides fault locations, fault explanations, patches and time spent by practitioners while debugging real bugs from CoREBench.

In the future, we intend to evaluate our proposed debugging aids by comparing them to the state of the art debugging tools [2], [3], [4], [6], [7], using benchmarks of both real and artificial faults (e.g. DBGBENCH [5] and the Siemens benchmark). Using DBGBENCH, we plan to evaluate the effectiveness of our approach in providing useful contextual information, by measuring its ability to find the pertinent function calls and inputs mentioned in the aggregated human-generated bug diagnosis. In addition, we plan to assess the performance of our approach by measuring the time taken by our tools to diagnose each error in DBGBENCH in comparison to the time spent by participants and other debugging tools. Finally, we intend to measure the ability of our probabilistic test oracle to determine plausible but incorrect patches, using developer provided (plausible) patches in DBGBENCH.

[2]http://www-static.cc.gatech.edu/aristotle/Tools/subjects

All information about our current and future debugging research can be found at the following address:

https://www.st.cs.uni-saarland.de/debugging/

V. Expected Contributions

The expected contributions of this work include:

- a *hybrid fault diagnosis* technique that harnesses the power of both statistical debugging and program slicing;
- The definition and construction of *probabilistic grammar models* that capture the probabilistic input structure and feature execution statistics of programs;
- A *debugging framework* to *automatically mine probabilistic grammar models* for bug diagnosis and support *input (fragment) manipulation and rectification*;
- A *probabilistic test oracle* that determines the *degree of correctness* of patches during debugging.

References

[1] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, Aug. 2016.

[2] J. A. Clause and A. Orso, "Penumbra: automatically identifying failure-relevant inputs using dynamic tainting," in *ISSTA*, 2009, pp. 249–260.

[3] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE '81, 1981, pp. 439–449.

[4] J. A. Jones, M. J. Harrold, and J. T. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE 2002, 2002, pp. 467–477.

[5] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. J. Ugherughe, and A. Zeller. (2016) How developers diagnose and repair software bugs (and what we can do about it). [Online]. Available: https://www.st.cs.uni-saarland.de/debugging/dbgbench/dbgbench.pdf

[6] G. K. Baah, A. Podgurski, and M. J. Harrold, "The probabilistic program dependence graph and its application to fault diagnosis," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 528–545, 2010.

[7] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5. ACM, 2001, pp. 57–72.

[8] M. Höschele and A. Zeller, "Mining input grammars from dynamic taints," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 720–725.

[9] 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*, ser. PLDI '05, 2005, pp. 15–26.

[10] A. X. Zheng, M. I. Jordan, B. Liblit, and A. Aiken, "Statistical debugging of sampled programs," in *Advances in Neural Information Processing Systems*, 2003, pp. 9–11.

[11] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *ISSTA*, 2016, pp. 165–176.

[12] D. Binkley, N. Gold, and M. Harman, "An empirical study of static program slice size," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 2, Apr. 2007.

[13] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," *ACM Sigplan Notices*, vol. 37, no. 1, pp. 4–16, 2002.

[14] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin, "Quickly detecting relevant program invariants," in *Proceedings of the 22nd international conference on Software engineering*. ACM, 2000, pp. 449–458.

[15] E. O. Soremekun, M. Böhme, and A. Zeller, "Programmers should still use slices when debugging," 2016. [Online]. Available: https://www.st.cs.uni-saarland.de/debugging/faultlocalization/technicalReport.pdf

[16] M. Collins, "Probabilistic context-free grammars (pcfgs)," *Lecture Notes*, 2013.

[17] M. Böhme and A. Roychoudhury, "CoREBench: studying complexity of regression errors," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, 2014, pp. 105–115.