Aletheia: A Failure Diagnosis Toolchain

Mojdeh Golagha
Technical University Of Munich
mojdeh.golagha@tum.de

Abu Mohammed Raisuddin
Technical University Of Munich
am.raisuddin@tum.de

Lennart Mittag
Technical University Of Munich
lennart.mittag@tum.de

Dominik Hellhake
Technical University Of Munich
dominik.hellhake@tum.de

Alexander Pretschner
Technical University Of Munich
alexander.pretschner@tum.de

ABSTRACT

Testing and debugging are time-consuming, tedious and costly. As many automated test generation tools are being applied in practice nowadays, there is a growing need for automated failure diagnosis. We introduce Aletheia, a failure diagnosis toolchain, which aims to help developers and testers reduce failure analysis time. The key ideas include: data generation to provide the relevant data for further analysis, failure clustering to group failing tests based on the hypothesized faults, and fault localization to pinpoint suspicious elements of the code. We evaluated Aletheia in a large-scale industrial case study as well as two open-source projects. Aletheia is released as an open-source tool on Github, and a demo video can be found at: https://youtu.be/BP9D68DO2ZI

CCS CONCEPTS

• **Software Engineering** → **Testing and debugging**;

KEYWORDS

Hit spectra, Failure clustering, Fault localization, Parallel debugging

1 INTRODUCTION

Testing and debugging are time-consuming, tedious and costly. With growing size and complexity of software, automated test generation, failure analysis, and fault repair are essential for improving the productivity and reducing the costs. While there is a significant progress in the automated test generation tools and their applicability in practice (i.e. [6], [21] etc.), there is still a need for improving failure diagnosis tools.

There are two types of techniques that help in reducing failure diagnosis time [8], *failure clustering* methods to group failing tests with respect to hypothesized faults and *automated fault localization* techniques to locate the faults. In our terminology, a *failure* is the deviation of actual run-time behavior from intended behavior, and a *fault* is the reason for the deviation [18].

Aletheia implements both of the above techniques to assist developers and testers to reduce failure diagnosis time. Where there are several failing tests due to multiple underlying faults, Aletheia helps users generate relevant data for further analysis while running the tests. After running the tests, it extracts failing tests and clusters them based on their underlying hypothesized faults. It also selects one (or more) representative tests for each cluster. Users can analyze representative failing tests to find all the faults without having to analyze all the failing tests one-by-one. Finally, if users need more help to locate the fault(s) in the code, it applies spectrum-based fault localization on each of the clusters to give users a ranked list of program elements based on their suspiciousness to be faulty. Users can analyze the most suspicious elements first to find the fault(s).

Problem. In different contexts, there is usually a large number of tests that are frequently executed as regressions happen. In case of large failures, how can we reduce failure analysis time?

Solution. We introduce Aletheia, a toolchain for failure diagnosis. Aletheia has 3 main components: data generation, failure clustering, and fault localization. The methodology applied in failure clustering component is based on [8], and the methodology applied in fault localization is based on [19], [2], and [12].

Organization. In the following, we explain the methodology employed by our tool (sec. 2), evaluation results (sec. 3), related works (sec. 4) and our plans for future work (sec. 5).

2 METHODOLOGY

In the following, we describe the main features of Aletheia and the methodology it employs for its users. Fig. 1 shows its main components: data generation, failure clustering, and fault localization. Each component can be used separately or the output of each one can be used as the input for the next one. Our tool can be used as a plug-in for Visual Studio or via command line.

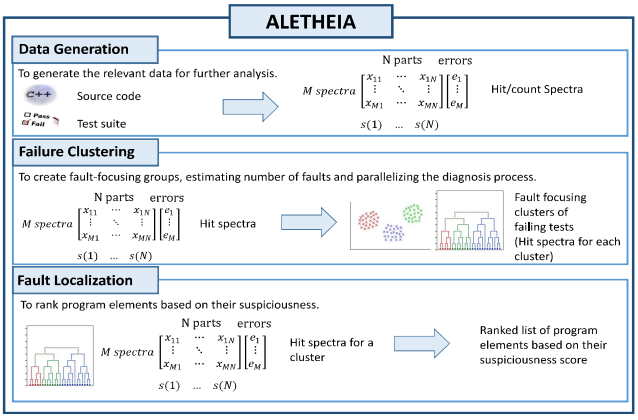

Figure 1: Aletheia with its 3 main components

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3183486

2.1 Data Generation

The first component intends to generate and prepare data for further analysis. To generate the data, Aletheia needs the source-code and its test suite. Then, it runs test, collects coverage information and aggregates them into some hit/count spectra. To collect coverage information, it instruments the code using an open source tool, OpenCPPCoverage [1]. OpenCPPCoverage measures code coverage for C and C++. Executing a program using this tool results in a report about which lines of code have been executed.

Aletheia first prepares test execution according to the start-up parameters of the given test suite. Then, it calls OpenCPPCoverage to execute tests while recording the coverage information and the test results. The final step is to generate a hit spectrum. A "hit spectrum" is a matrix that summarizes tests coverage information. Table 1 shows an example of a function hit spectrum for a hypothetical program and its 5 tests. Each "1" in the table means that the respective function has been executed while running the respective test. This step is optional and depends on the level of abstraction at which the analysis should happen. Aletheia is able to generate 4 different kinds of spectra: **statement** (which statements have been executed), **function calls** (which functions have been invoked), **function calls with input parameters** (same as the previous one but separating calls considering different input parameters), and **function call count** (same as function calls, but recording count number of hits).

Since OpenCPPCoverage provides line coverage only, for other types of spectra, the final step is needed to partially parse the source files to aggregate the line-level information to provide statement- and function-level coverage.

Generated hit/count spectra are saved as csv files. Thus, they can be used as a dataset for machine learning or as the input for failure clustering and/or fault localization.

For an effective fault localization, one needs to decide about the granularity of program elements (i.e. statement, function, branch, etc.). The goal is finding program elements that are highly correlated with failure. If program elements are too simple or too complex for characterizing faults, results contain too many false negatives (fault is complex thus cannot be identified by simple program elements) or false positives (fault is simple thus more elements can be tagged suspicious)[15] [7]. For example, if failures exhibit some unique execution patterns such as a particular sequence of method calls, then statements might be too simple to characterize the fault that induces these failures. Since there is not a best solution that always over-performs others, our tool provides the opportunity for its users to try different spectra to find the more suitable one for their own context.

The data generation component of Aletheia is designed for C++ code and is also compatible with Google testing framework [2]. Google Test is a unit testing library for the C++ programming language, based on the xUnit architecture [3].

[1] OpenCppCoverage, available at https://github.com/OpenCppCoverage, licensed under the GNU General Public License version 3 (GPLv3).
[2] https://github.com/google/googletest
[3] https://xunit.github.io/

Table 1: Hit spectrum - Function invocation

FunctionID	Function	T1	T2	T3	T4	T5
Fun1	Foo1	1	1	1	0	1
Fun2	Foo2	0	0	1	1	0
Fun3	Foo3	1	1	1	0	1
Fun4	main	1	1	1	1	1
Execution Results		fail	fail	pass	pass	fail

2.2 Failure Clustering

Clustering failures is effective in reducing failure analysis time [8]. Failure clustering techniques attempt to group failing tests with respect to the faults that caused them [17]. Clustering failures has three advantages:

(1) It eliminates the need to analyze each failing test individually. To achieve this goal, it is enough to select one representative for each cluster and analyze only the representatives to find all the underlying faults in case of multiple faults[8].
(2) It provides the opportunity for debugging in parallel [11].
(3) It gives an estimation of the number of faults causing the failures. Fault localization, while there are several faults in the code, is more challenging than when there is only one fault in the code. In addition, considering the significantly high execution time of the tests at industrial scale, it is not always possible to find the first fault, resolve it and re-run all the tests. It is more efficient to find all the faults and repair them before re-running tests [8].

Aletheia offers failure clustering as the second component. As input, it receives a hit spectrum (provided by the first component or by other tools such as GZoltar [4] for Java programs). It works based on the methodology explained in [8]. First, it generates a hierarchical tree of failing tests (considering hit spectra as dataset), then cuts the tree into some clusters utilizing spectrum-based fault localization. The number of clusters is decided based on the fault localization results [8]. In the end, it selects k (user's desired number) representatives for each cluster. As output, users receive a csv file declaring the number of clusters, members of each cluster, and representative tests for each cluster, and also a csv file as a hit spectrum for each cluster (for further analysis).

To do hierarchical clustering, one needs a method for computing the distance between clusters and a metric for computing the distance between pairs of data objects, failures in our case. Our tool provides several options for its users. For the **clustering method**, unweighted average distance (UPGMA), centroid distance (UPGMC), furthest distance, weighted center of mass distance (WPGMC), shortest distance, inner squared distance (ward) and weighed average distance (WPGMA) are possible; for the **distance metric**, Euclidean distance, City Block, Minkowski, Cosine (1-Cosine), Correlation (1-Correlation), and Jaccard (1-Jaccard) are possible.

All these methods and metrics are implemented in Matlab Machine Learning toolbox.

2.3 Fault Localization

Automated fault localization techniques aim to "identify some program event(s) or state(s) or element(s) that cause or correlate with the failure to provide the developer with a report that will aid fault

repair" [15]. Spectrum-based techniques are a popular group of advanced techniques for automated fault localization.

Utilizing the information summarized in a hit spectrum (of all tests or one cluster), a spectrum-based technique returns a ranked list of program elements based on their suspiciousness score to be faulty. An element is more suspicious if it is executed more frequently in failing executions and less frequently in passing executions. There is a plethora of similarity metrics to measure the suspiciousness score [20]. An example is the D* metric [19]:

$$D^* = \frac{(N_{CF})^*}{N_{UF} + N_{CS}}, \quad (1)$$

where N_{CF} is the number of failing executions that have covered the element (function in the example); N_{UF} is the number of failing executions that have not covered the element; and N_{CS} is the number of passing executions that have covered the element. Table 2 shows the suspiciousness scores and ranks of the elements (functions) of the hypothetical program in Table 1.

In Aletheia, three of the most popular metrics, Tarantula [12], Ochiai [3], and Jaccard [5] and also D* that has shown to be among the best in different studies [8][16] are implemented.

Table 2: D* (*=4) scores and ranks

FunctionID	Score	Rank
Fun1	81	1
Fun2	0	3
Fun3	81	1
Fun4	40.5	2

Table 3: Clustering effectiveness using Average/Euclidean

	# of failures	# of faults	# of clusters	ARed(%)
Case 1	13	3	5	80
Case 2	27	1	1	100
Case 3	14	3	3	100

Table 4: Fault localization effectiveness using D* (*=4)

spectrum	# of bugs	rank of faulty function(s)
Case 3	3 (bug-57,60,61)	1-3-8
Cluster 1	1 (bug-60)	1
Cluster 2	1 (bug-57)	1
Cluster 3	1 (bug-61)	1

3 EVALUATION

We evaluated our tool, Aletheia, in a large-scale industrial case study with ca. 850 KLoC [8]. To extend the evaluation for generalization beyond industrial case study, we considered two programs; Firstly, *OpenCV* (Open Source Computer Vision)[4], a library for computer vision written in C++. We used Pairika [5] database to find OpenCV's real faults for our evaluation. Secondly, the standard *Apache Commons Lang* [6], a package of Java utility classes. Some of the Lang's real faults are introduced in Defects4j database [14].

[4]https://github.com/opencv/opencv
[5]https://github.com/tum-i22/Pairika
[6]https://github.com/apache/commons-lang

Scenario 1: In the first scenario, we used project Lang to evaluate our failure clustering and fault localization components in single and multiple fault cases. Among 65 faults introduced for Lang in Defects4j database, only 4 of them (Bug-30, -31, -34, and -57) induce more than two failing tests. We deliberately generated 3 cases for evaluation using these 4 bugs. In case 1, we included bug-30, -31, and -39. In case 2, we included bug-34. In case 3, we included bug-57, -60, and -61. We included these combinations, in order to make the clustering/localization tasks more challenging. For each combination, we tried to consider bugs that are in the same source file. We used Gzoltar to generate function-level hit spectra.

The first option to reduce the failure analysis time is to cluster failures. Table 3 shows the evaluation results. The ARed metric is obtained from [8] and denotes the percentage of reduction in analysis time with respect to the best possible reduction. In some cases, Aletheia might find more than one cluster for one fault. Nevertheless, there is a huge reduction in analysis time. For example, in case 1, the 3 faults in the code induce 13 failures. Aletheia found 5 clusters. If we select one representative for each cluster, one should analyze only 5 failures to find all the faults rather than 33. This yields 80% reduction. One might get the list of representative tests and start the debugging process.

The second option is fault localization on found clusters in parallel. The segregation between failures, obtained from clustering, improves the accuracy of multiple fault localization. For example, table 4 shows the ranks of faulty functions in case 3 before clustering and in found clusters. The results show that faults might mask each other. In this case, bug-57 with 11 failures masks bug-60 and -61 with 1 and 2 failures.

Scenario 2: In scenario 2, we used OpenCV to evaluate our data generation and fault localization components. Pairika is a failure diagnosis benchmark which introduces 40 real bugs of OpenCV project. Due to space limitations, we did fault localization for 6 faults we extracted from Pairika. These faults are in Core, Photo, and Machine Learning modules with 196550, 10880, and 19398 LoC and 10528, 166, and 39 tests respectively.

Table 5 shows the fault localization results using D* and Ochiai metrics on 3 different spectra. As the results show, the faulty elements are not always ranked in top 5. We are aware of the shortcomings of spectrum-based fault localization techniques. However, we believe that with the use of domain-specific data, we will be able to tune them for a higher effectiveness.

4 RELATED WORK

To the best of our knowledge, there is no other tool that supports failure clustering and fault localization in one package.

Data generation. There are a plenty of code coverage, instrumenting and profiling tools. Most code coverage tools cater to Java, followed by C, and C++, and some .NET. Among the open source tools for C++ code coverage, we picked OpenCPPCoverage.

Nevertheless, these tools do not generate a hit spectrum based on the coverage information. Gzoltar [4] is an eclipse plug-in for testing and debugging Java programs which has the hit spectra generation feature. It is also compatible with JUnit test framework [7]. Aletheia is a Visual Studio plug-in and is compatible with Google

[7]https://github.com/junit-team

Table 5: Fault localization effectiveness using D* and Ochiai metrics

		Best rank of faulty element using D* (*=4)			Best rank of faulty element using Ochiai		
Bug ID		Statement	Function call	function call w. input	Statement	Function call	function call w. input
ml2	#5413	43	4	28	43	4	28
ml3	#5911	1	3	240	1	3	240
photo1	#8706	1	1	98	1	1	98
photo2	#5045	1	1	117	1	1	117
core5	#8941	248	28	88	126	16	71
core14	#6380	1	300	715	1	38	114

testing framework. We don't know any other openly accessible tool compatible with Google Test for generating hit spectra.

Failure clustering. There are several works on clustering failing (and passing) executions in the literature [8]. Nevertheless, we are not aware of any openly accessible tool. Aletheia's failure clustering component receives a hit spectrum as input. Since the hit spectrum's representation is a simple csv file, it is not limited to any specific programming language. In addition, it can select some representatives for each cluster to start the debugging process.

Fault localization. Pearson et. al. [20] give a list of tools used in fault localization studies. There are different strategies for localizing faults in the literature. In our paper, we focus on spectrum-based techniques. BARINEL [1] is a framework to combine spectrum-based fault localization and model-based diagnosis. GZoltar [4] is an automated testing and debugging framework using Ochiai metric. HSFal [13] is a slice spectrum fault locator. Pinpoint [3] is a fault localization tool using Jaccard coefficient. Tarantula [10] is a fault localization tool using Tarantula. Zoltar [9] is a spectrum-based fault localization tool. Zoltar-M [1] is a tool for detecting multiple bugs. Aletheia's fault localization component covers all the metrics implemented in above-mentioned tools. In addition, it covers D* metric.

5 CONCLUSION AND FUTURE WORK

Debugging is a tedious and time-consuming phase in the software development life-cycle. To help testers and developers in reducing failure diagnosis and repair time, we introduce Aletheia, an open-source plug-in for Visual Studio. Using Aletheia, users can generate data at different granularity levels for further analysis, cluster failing tests with respect to the hypothesized faults, and pinpoint the root cause of the observed failures. The tool, as well as a tutorial, can be obtained from https://github.com/tum-i22/Aletheia.

Future work includes the following: we plan to refine the rankings yielded by spectrum-based fault localization by combining spectrum-based techniques and call/data graph analysis. We believe the new approach can improve localization effectiveness. Furthermore, we plan to add more visualizations of the localization reports. Visualizing call/data graph of failing tests while highlighting most suspicious methods might be helpful in repairing the faults. Finally, we plan to study fault localization in a domain-specific way to be able to suggest the best setting of our tool for different domains. Thus, another useful addition can be automated suggestion of the similarity metrics based on the domain.

REFERENCES

[1] R. Abreu, P. Zoeteweij, and Arjan J.C. Van Gemund. 2009. Localizing software faults simultaneously. In *Proceedings - International Conference on Quality Software.*

[2] R. Abreu, P. Zoeteweij, and A. J. C. Van Gemund. 2006. An evaluation of similarity coefficients for software fault localization. In *Proceedings - 12th Pacific Rim International Symposium on Dependable Computing, PRDC 2006.*

[3] R. Abreu, P. Zoeteweij, and A. J. C. Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Proceedings - Testing: Academic and Industrial Conference Practice and Research Techniques, TAIC PART-Mutation 2007.*

[4] J. Campos, A. Riboira, A. Perez, and R. Abreu. 2012. GZoltar: an eclipse plug-in for testing and debugging. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012* (2012).

[5] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. 2002. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks.*

[6] E. P. Enoiu, Adnan Čaušević, Thomas J. Ostrand, Elaine J. Weyuker, Daniel Sundmark, and Paul Pettersson. 2016. Automated test generation using model checking: an industrial evaluation. *International Journal on Software Tools for Technology Transfer* 18, 3 (01 Jun 2016).

[7] M. Golagha and A. Pretschner. 2017. Challenges of Operationalizing Spectrum-Based Fault Localization from a Data-Centric Perspective. In *Proceedings - 10th IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2017.*

[8] M. Golagha, A. Pretschner, D. Fisch, and R. Nagy. 2017. Reducing failure analysis time: An industrial evaluation. In *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track.*

[9] T. Janssen, A. Gemund, and R. Abreu. 2009. Zoltar: A Spectrum-based Fault Localization Tool. *Instrumentation* (2009).

[10] J.A. Jones, M.J. Harrold, and J.T. Stasko. 2001. Visualization for fault localization. *Proceedings of ICSE 2001 Workshop on Software Visualization* (2001).

[11] J. a. Jones, J. F. Bowring, and M. J. Harrold. 2007. Debugging in Parallel. In *Proceedings of the 2007 international symposium on Software testing and analysis.*

[12] J.a. James a Jones and M. J. Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. *Automated Software Engineering* (2005).

[13] X. Ju, X. Jiang, Sh.and Chen, X. Wang, Y. Zhang, and H. Cao. 2014. HSFal: Effective fault localization using hybrid spectrum of full slices and execution slices. *Journal of Systems and Software* 90, 1 (2014).

[14] R. Just, D. Jalali, and M. D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis.*

[15] W. Masri. 2015. Automated Fault Localization. Advances and Challenges. In *Advances in Computers.* Vol. 99.

[16] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. 2017. Evaluating and Improving Fault Localization. In *Proceedings of the 39th International Conference on Software Engineering.*

[17] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, Jiayang Sun, and Bin Wang. 2003. Automated support for classifying software failure reports. In *25th International Conference on Software Engineering, 2003. Proceedings.*

[18] M. Utting, B. Legeard, and A. Pretschner. 2006. A Taxonomy of Model-Based Testing. *Software Testing, Verification and Reliability* 22, April (2006).

[19] W. E. Wong, V. Debroy, R. Gao, and Y. Li. 2014. The DStar method for effective software fault localization. *IEEE Transactions on Reliability* 63, 1 (2014).

[20] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* PP, 99 (2016).

[21] H. Yoshida, G. Li, T. Kamiya, I. Ghosh, S. Rajan, S. Tokumoto, K. Munakata, and T. Uehara. 2017. KLOVER: Automatic Test Generation for C and C++ Programs, Using Symbolic Execution. *IEEE Software* 34, 5 (2017), 30–37.