A Topology-based Model for Estimating the Diagnostic Efficiency of Statistics-based Approaches

Alexandre Perez, André Riboira, Rui Abreu
Department of Informatics Engineering
Faculty of Engineering, University of Porto
Porto, Portugal
{alexandre.perez, andre.riboira}@fe.up.pt, rui@computer.org

***Abstract*—Spectrum-based fault localization (SFL) and dynamic code coverage (DCC) are two statistics-based fault localization techniques used in software fault diagnosis. Due to their nature (statistical analysis of the coverage information), the best technique of the two depends greatly on the system under test code structure and size. We propose a lightweight, topology-based analysis to quickly estimate the project under test coverage matrix when executed, based on the source code structure. This analysis will choose which fault localization technique to use by creating an hierarchical model of the system. To validate our proposed approach, an empirical evaluation was performed, injecting faults in six real-world software projects. We have demonstrated that using the topology-based analysis to choose the best fault localization technique provides a better execution time performance on average (23%) than using DCC (9%), when comparing to SFL.**

***Index Terms*—Coverage estimation, software diagnosis, topology-based model.**

I. Introduction

The always increasing software quality requirements, coupled with current software projects' size and complexity, demand efficient automated fault localization techniques to help developers in pinpointing failures in their systems. Spectrum-based Fault Localization (SFL) [1], [2] is a statistics-based fault localization technique, that uses abstraction of program traces (program spectra) to correlate software component activity with observed failures. SFL is amongst the most diagnostic effective statistic-based technique [2], [3], [4].

However, in SFL, as with other statistics-based techniques, the overhead of gathering the input information so that a diagnostic ranking can be computed remains fairly high. This can lead to scaling problems, particularly when debugging in resource constrained environments.

To mitigate the scaling problems that SFL faces, we have developed a dynamic technique, coined Dynamic Code Coverage (DCC) [5]. This approach instruments the system's source code using a coarse detail granularity, and then decides which components should be re-instrumented and re-tested, based on intermediate fault localization results from executing SFL. This iterative process will continue until the top ranked components (*i.e.*, the ones more likely to be faulty) are of fine granularity. DCC has shown that, for large projects, it can improve execution time. It can also reduce the diagnostic report size when compared to SFL, lessening the effort needed by the developers to inspect the automated fault localization output. However, for small software projects, DCC is not that effective, even showing worse results than SFL [5].

Both SFL and DCC's diagnostic efficiency is greatly dependent on the coverage information per test case (the best is that test cases *touch* different parts of the program). In this paper, we propose a technique that is able to inspect a certain software project and assert which fault localization approach should be used, based on the system's topology. Note that such model may even be used to decide not to use a fault localization technique at all. To the best of our knowledge, our topology-based analysis has not been described before.

In our empirical evaluation, we have validated our approach by performing the fault localization task on faulty versions of six different real-world open-source software projects. We have demonstrated that using our topology-based analysis to decide which fault localization technique should be used provides a better average execution time performance (23%, σ=0.26), than using DCC (9%, σ=0.49), when compared with SFL.

II. Motivation

Automatic fault localization techniques aid developers in pinpointing the root cause of software failures. Amongst the most diagnostic effective techniques is Spectrum-based Fault Localization (SFL), a statistical technique that exploits coverage information from passed an failed system runs [2], [3], [4]. A passed run is a program execution that is completed correctly, and a failed run is an execution where an error was detected [2]. The criteria for determining whether a run has passed or failed can be from a variety of different sources, namely test case results and program assertions, among others. The information gathered from these runs is their program spectra.

A program spectrum is a characterization of a program's execution on a dataset [6]. This collection of data, gathered at runtime, provides a view on the dynamic behavior of a program. The data consists of counters or flags for each software component. In order to obtain information about which components were covered in each execution, the program's source code needs to be instrumented, similarly to code coverage tools [7]. This instrumentation will monitor each component and register those that were executed. Components

can be of several detail granularities, such as classes, methods, and lines of code.

The program spectra of N runs constitutes a binary $N \times M$ matrix A, where M corresponds to the instrumented components of the program. Information of passed and failed runs is gathered in an N-length vector e, called the error vector. The pair (A, e) serves as input for the SFL technique, as seen in Figure 1.

$$\begin{array}{c} \\ N \text{ spectra} \end{array} \overbrace{\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NM} \end{bmatrix}}^{M \text{ components}} \quad \overbrace{\begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_N \end{bmatrix}}^{\text{error detection}}$$

Fig. 1. Input to SFL.

With this input, fault localization consists in identifying what columns of the matrix A resemble the vector e the most. For that, several different similarity coefficients can be used [8]. One of the most effective is the Ochiai coefficient [1], used in the molecular biology domain:

$$s_O(j) = \frac{n_{11}(j)}{\sqrt{(n_{11}(j) + n_{01}(j)) \times (n_{11}(j) + n_{10}(j))}} \quad (1)$$

where $n_{pq}(j)$ is the number of runs in which the component j has been touched during execution ($p = 1$) or not touched during execution ($p = 0$), and where the runs failed ($q = 1$) or passed ($q = 0$). For instance, $n_{11}(j)$ counts the number of times component j has been involved in failed executions, whereas $n_{10}(j)$ counts the number of times component j has been involved in passed executions. Formally, $n_{pq}(j)$ is defined as

$$n_{pq}(j) = |\{i \mid a_{ij} = p \wedge e_i = q\}| \quad (2)$$

SFL can be used with program spectra of several different granularities. However, it is most commonly used at the line of code (LOC) level. Using coarser granularities would be difficult for programmers to investigate if a given fault hypothesis generated by SFL was, in fact, faulty. This can lead to scalability issues, as every LOC has to be instrumented for the program spectra to be gathered.

Instrumentation can hit execution time by as much as 50% in code coverage tools that use similar instrumentation techniques to gather information about the program execution [7]. As such, fault localization techniques that use program spectra may be impractical for large, real-world, and resource-constrained projects that contain hundreds of thousands of LOCs.

In order to solve the potential scaling problem that statistics-based fault localization techniques have, we have proposed a dynamic approach, called Dynamic Code Coverage (DCC) [5]. This technique, which can be seen in Algorithm 1, automatically adjusts the detail granularity per software component.

Algorithm 1 Dynamic Code Coverage.

```
1:  procedure DCC(System, TestSuite,
        InitialGranularity, FinalGranularity)
2:      R ← ∅
3:      F ← System
4:      T ← TestSuite
5:      G ← InitialGranularity
6:      repeat
7:          INSTRUMENT(F, G)
8:          (A, e) ← RUNTESTS(T)
9:          C ← SFL(A, e)
10:         F ← FILTER(C)
11:         R ← UPDATEREPORT(R, F)
12:         T ← NEXTTESTS(TestSuite, A, F)
13:         G ← NEXTGRANULARITY(F)
14:     until ISFINALGRANULARITY(F,
            FinalGranularity)
15:     return R
16: end procedure
```

First, our approach instruments the source code using a desired coarse granularity (*e.g.*, package level in Java) and the fault localization is executed by performing the SFL technique. Then, it is decided which components are *zoomed-in* based on the intermediate results of the of the fault localization technique. With *zooming-in* we mean changing the granularity of the instrumentation on a certain component to the next detail level (*e.g.*, in Java, for instance, instrument classes, then methods, and finally statements). This *zoom-in* can be done in different ways, with the use of filters. See [5] for examples of filtering methods.

After deciding which components will be re-instrumented, the fault localization procedure is executed again, by running the tests that touch the re-instrumented components. This process of performing SFL, filtering the results, and re-instrument the filtered components will repeat itself until the desired final granularity is reached.

DCC, as an iterative technique, is aimed at improving the execution time of the fault localization procedure, and shows a substantial reduction of the execution time (27% on average) and the diagnostic report size (67% on average), when compared with SFL [5]. However, for some projects, the task of re-instrumenting and re-testing may consume more time than performing a single iteration with a fine-grained instrumentation throughout the entire project. This can happen in projects where each test covers a large portion of the code, either due to having an unbalanced code topology, or having a relatively small codebase. In both of these instances, the resulting program spectra matrix will be rather dense, therefore many software components will have to be *zoomed-in*.

A technique able to analyze in a lightweight manner the projects under test, and assert if a project's structure is not suited for DCC would be of crucial importance. It would serve as a decision support technique to whether one should employ

SFL or DCC when performing the fault localization procedure on a given project. Aside from its use in fault diagnosis, this technique would also be useful in other instances of tools that depend on either the project under test coverage information or code structure. One example is with automated test generation tools. This way, these tools could have a fast method of knowing then to stop generating tests, without having to run the entire test suite.

III. Topology Model

As detailed in the previous section, if a developer is to use the best diagnostic and time effective fault localization technique to debug his software application of those considered, some analysis must be done beforehand. In this section, we describe our analysis methodology for quickly estimating a software project's execution coverage density by inspecting its topology.

This analysis is intended to provide a general and coarse overview of a project's execution based on it's source code, in a fast and lightweight manner. It extrapolates information based on how a program is structured. As such, programming paradigms that enforce a certain hierarchy (common throughout different projects) such as object-oriented programming (OOP), is needed. We use this hierarchy to construct a model of the system, facilitating its subsequent analysis. Test cases throughout the project should be also identifiable with minimal static analysis. Throughout this paper, we will be using the OOP hierarchy of Java (in descending order of granularity: packages, classes, methods and statements) for all the examples of how our model is constructed and processed.

As a first step, a tree model of the system topology is constructed. This tree's root node symbolizes the project, with all the other nodes being either packages, classes or methods. Method nodes are the lowest granularity nodes, aimed at speeding up the analysis. One thing to note is that, because we are not analyzing the statements of a method, local classes (*i.e.*, classes that are defined inside methods) also do not show up in the tree. A class node can also have an annotation stating that a certain class is a test case. This way, test cases are easily identifiable. The edges represent relations (*e.g.*, classes are connected to their respective package).

After the topology tree is constructed, its analysis can be performed. For that we use a score function S, defined as

$$S = \frac{\sigma_{m/c}}{N_m} + \frac{\sigma_{c/p}}{N_c} + e^{-\frac{N_c-1}{C}} + e^{-\frac{N_t-1}{T}} \tag{3}$$

The first two terms of the score function S are related to the system's coverage matrix density: $\sigma_{m/c}$ and $\sigma_{c/p}$ are the standard deviation of methods per class and of classes per package, and N_m and N_c are the number of methods and classes, respectively. What both of these terms are estimating is if the topology tree exhibits a balanced structure, or if the nodes show a high variance of children. For example, if a certain package contains more classes than other packages, one can assume that the contents of this package are more likely to be touched by an execution than the ones in a package with less classes. Similarly, if we increase the detail level, the same can be said for the number of class methods. Both terms will tend to zero if the tree is balanced.

The last two terms of the score function S estimate the system's coverage matrix size. N_t is the number of test cases. Coefficients C and T are the weights that the number of classes and tests have in the score function, and they can be adjusted according to a project's topology, and determine the impact that each term has in the overall score function. Being inverse exponential functions, both of these terms exhibit a high value if a system contains few classes or tests, but this value rapidly decreases as the number of classes and tests grows.

The result of this score function S can, then, be regarded as a coarse extrapolation of the attributes of the subject's coverage matrix. If the result of this function is a value close to zero, this means that the matrix should be sparse and fairly big in size. Otherwise, if the value is not close to zero, this means that the matrix is small or/and rather dense.

The score function S can be used as a decision support mechanism. If we revisit our fault localization example from the previous section, we can use this analysis to support the use of DCC or SFL to debug a certain project. For that, one can use:

$$FaultLocalization = \begin{cases} DCC & \text{if } S \leq E \\ SFL & \text{otherwise} \end{cases}$$

meaning that the best fault localization for a given project is DCC if its score is below a given threshold E, which can also be adjusted, and SFL if the score is above the specified threshold. Note that the decision above can be refined to decide not to use any fault localization technique.

IV. Empirical Evaluation

In this section, we evaluate the validity and performance of our topology-based analysis model in its application to statistics-based fault localization. First, we introduce the programs under analysis and the evaluation metrics. Then, we discuss the empirical results.

A. Experimental Setup

For our empirical study, six subjects written in Java were considered:

- `NanoXML`[1] – a small XML parser.
- `org.jacoco.report` – report generation module for the `JaCoCo`[2] code coverage library.
- `Xstream`[3] – an object serialization library.
- `JGAP`[4] – a genetic algorithms library.
- `XML-Security` – a component library implementing XML signature and encryption standards. This library is part of the Apache Santuario[5] project.

[1] NanoXML – http://devkix.com/nanoxml.php
[2] JaCoCo – http://www.eclemma.org/jacoco/index.html
[3] Xstream – http://xstream.codehaus.org/
[4] JGAP – http://jgap.sourceforge.net/
[5] Apache Santuario – http://santuario.apache.org/

- `JMeter`[6] – a desktop application designed to load test functional behavior and measure performance of web applications.

The project details of each subject are in Table I. The LOC count information was gathered using the metrics calculation and dependency analyzer plugin for Eclipse `Metrics`[7]. Test count and coverage percentage were collected with the Java code coverage plugin for Eclipse `EclEmma`[8].

TABLE I
EXPERIMENTAL SUBJECTS.

Subject	Version	LOCs (M)	Test Cases	Coverage
`NanoXML`	2.2.6	5393	8	53.2%
`org.jacoco.report`	0.5.5	5979	33	97.2%
`Xstream`	1.4.3	35944	174	84.8%
`JGAP`	3.6.2	48590	88	67.1%
`XML-Security`	1.5.0	60946	460	59.8%
`JMeter`	2.6	127359	534	34.2%

To assess the efficiency and effectiveness of our model the following experiments were performed, using fifteen faulty versions per subject program. As the subject programs are bug-free, we injected common mistakes in the programs – one fault in each of the fifteen versions, and executed:

- Fault localization with SFL.
- Fault localization with DCC.
- The topology analysis model, followed by the fault localization technique that the model considers appropriate based on the subject's score.

For each execution, we have gathered the total execution time and the program spectra matrix density. After some training, for this set of experimental subjects, we have used $C = 100$ and $T = 50$ in our model. The model decides employing DCC over SFL when the score function is $S \leq 1.0$.

The experiments were run on a 2.7 GHz Intel Core i7 MacBook Pro with 4 GB of RAM, running OSX Lion.

B. Experimental Results

Figures 2 to 7 summarize the overall execution time outcomes for all experimental subjects. It is worth to note that the results shown are gathered by running the entire fault localization and topology analysis experiments detailed in the previous section, and do not pertain only to the instrumentation overhead.

The first two subjects to be analyzed were `NanoXML` and `org.jacoco.report`, whose experimental results can be seen in Figures 2 and 3. In both projects, the SFL execution is faster than employing DCC. In fact, the DCC shows an execution time increase of 50% on average (σ=0.35). According to Table I, both projects show a relatively small codebase. At the same time, the amount of test cases in both is fairly low. This kind of projects, small in size and with a low amount of test cases, but with a coverage of over 50% are not fit for use with regular DCC. Their generated program spectra matrices, detailed in Section II, will be rather dense. Because of this, many components would have similar coefficients, rendering the filtering operation ineffective: either discarding many different components, or keeping a lot of components to be re-instrumented and re-tested. The best fault localization technique for `NanoXML` and `org.jacoco.report` is, in fact, SFL.

As a result of having a small codebase, both `NanoXML` and `org.jacoco.report` are scored above the exploration threshold by the topology-based analysis. This means that

[6]JMeter – http://jmeter.apache.org/
[7]Metrics – http://metrics.sourceforge.net/
[8]EclEmma – http://www.eclemma.org/

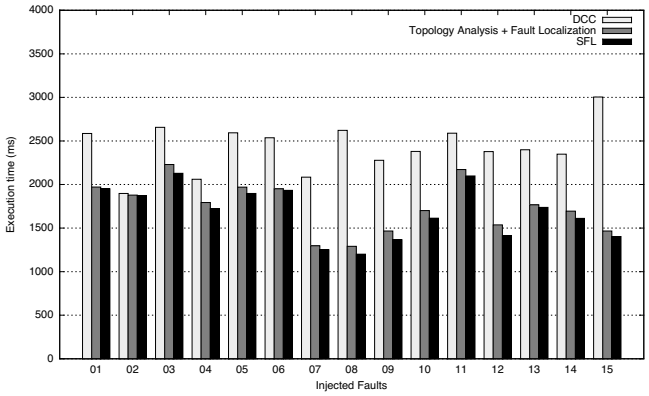

Fig. 2. `NanoXML` time execution results.

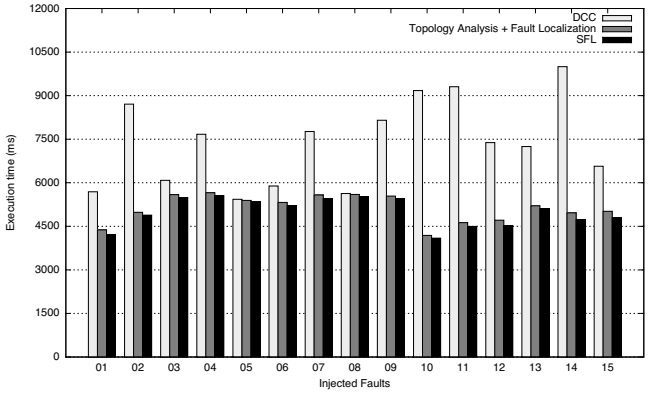

Fig. 3. `org.jacoco.report` time execution results.

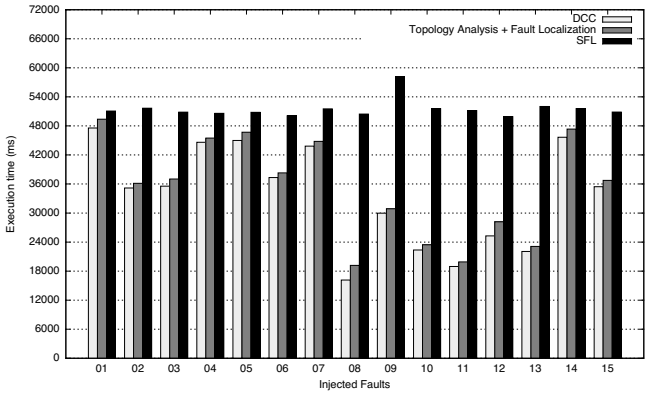

Fig. 4. `Xstream` time execution results.

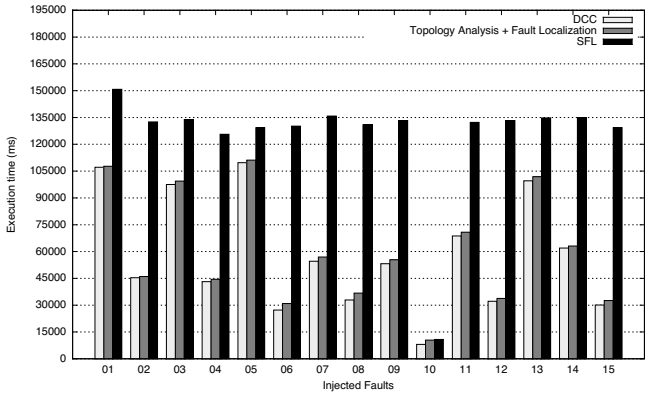

Fig. 5. `JGAP` time execution results.

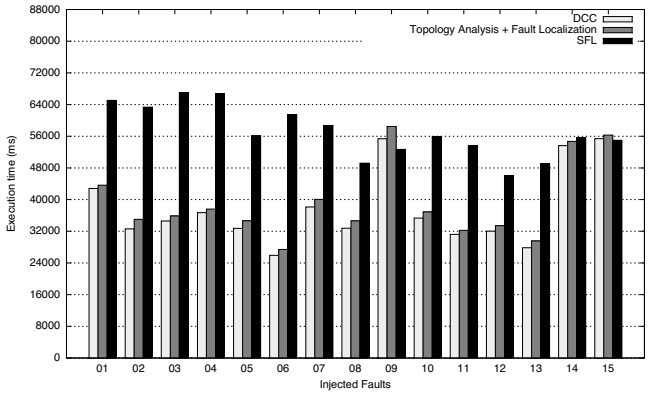

Fig. 6. `XML-Security` time execution results.

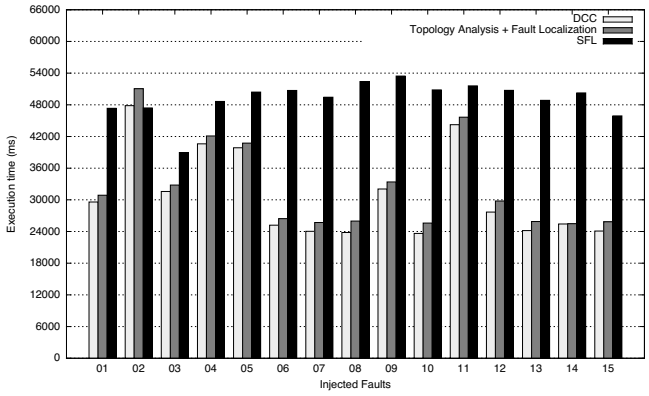

Fig. 7. `JMeter` time execution results.

the fault localization technique chosen by our analysis to be performed is SFL – the best fault localization technique of those considered. Our lightweight analysis, when compared to running just SFL, shows a 3% increase in execution time on average (σ=0.02).

In the other test subjects, `Xstream` (Figure 4), `JGAP` (Figure 5), `XML-Security` (Figure 6) and `JMeter` (Figure 7), the fault localization with DCC is faster than using SFL by 39% on average (σ=0.20). This is due to the fact that the generated program spectra matrices are sparser. Also, as programs grow in size, the overhead of a fine-grained instrumentation (used in methodologies such as SFL) is much more noticeable. In this kind of sizable projects (see project information in Table I), and if the matrix is sparse enough, it is preferable to re-run some of the tests, than to instrument every LOC at the start of the fault localization process.

These four subjects have all scored lower than the exploration threshold by the topology-based analysis. Thus, the fault localization technique used by the analysis in these instances is DCC. Our analysis, combined with the fault localization, shows a 5% increase in execution time on average (σ=0.05).

Overall, when compared with SFL, using the topology-based analysis to choose the best fault localization technique for each subject provides a better execution time performance (23% on average, σ=0.26) than pure DCC (9% on average, σ=0.49).

TABLE II
PROGRAM SPECTRA MATRIX DENSITIES.

Subject	Density	σ
`NanoXML`	61%	0.094
`org.jacoco.report`	15%	0.008
`Xstream`	12%	0.007
`JGAP`	7%	0.001
`XML-Security`	7%	0.003
`JMeter`	2%	0.001

The other metric that was gathered in these experiments was the program spectra matrix density. Table II shows the average matrix densities for all test subjects. As was shown previously, the two subjects that our topology-based analysis considers unfit for DCC are `NanoXML` and `org.jacoco.report`. This was supported by the DCC execution times in our experiments, which under-performed when compared with SFL. These two subjects are those whose matrices were of higher density out of all considered subjects. Thus, our analysis correctly estimated the application behavior during execution by inspecting their source code's topology.

It is worth noting that the main threat to the validity of this empirical evaluation is related to the injected faults. These injected faults, despite being fifteen in total for each experimental subject, may not represent the entire conceivable software fault spectrum.

V. RELATED WORK

Statistics-based fault localization techniques, as stated above, use an abstraction of program traces, also known as program spectra, to find a statistical relationship between software components and observed failures. Well-known examples of such approaches are the Tarantula tool by Jones, Harrold, and Stasko [9], the Nearest Neighbor technique by Renieris and Reiss [10], the Sober tool by Lui, Yan, Fei, Han, and Midkiff [3], the work of Liu and Hand [11], CrossTab by Wong, Wei, Qi, and Zap [4], the Cooperative Bug Isolation (CBI) by Liblit and his colleagues [12], [13], [14], [15], the Time Will Tell approach by Yilmaz, Paradkar, and Williams [16], HOLMES by Chilimbi *et al.* [17], and MKBC by Xu, Chan,

Zhang, Tse, and Li [18]. Although differing in the way they derive the statistical fault ranking, all techniques are based on measuring program spectra. Note that this list is by no means exhaustive. However, none of these employ a dynamic approach to fault localization, such as DCC. Moreover, to our knowledge, there is no toolset or approach that uses more than one different fault localization technique, selecting which to use based on source code topology analysis.

VI. Conclusions & Future Work

We have shown that current statistics-based automated software fault localization techniques, namely Spectrum-based Fault Localization (SFL) and Dynamic Code Coverage (DCC), still have some inefficiencies that need to be addressed. The former can suffer from scalability issues, since the entire system under test must be instrumented for the diagnosis to be performed. The latter reduces the instrumentation needed to perform the fault localization task, however, its performance can drop if its coverage matrix is rather dense, generally due to an unbalanced and/or small codebase topology. As a result, and according to how a certain system is structured, the best fault localization technique to be used can vary.

A topology-based analysis method was presented with the aim of supporting the decision of whether employing SFL or DCC as the fault localization technique for a certain software application. This analysis method creates a hierarchical model of the system under test and inspects it with the use of a score function. This function can be regarded as a coarse extrapolation of the coverage per test case, and is used as a decision support mechanism by comparing its outcome with a fixed threshold. As such, this analysis estimates a system's execution coverage density by examining its topology. In our empirical evaluation, we have demonstrated that using the topology-based analysis to choose the best fault localization technique provides a better execution time performance on average (23%, σ=0.26) than using DCC (9%, σ=0.49), when comparing to SFL.

As for future work, we intend to investigate the use of this topology-based analysis within other problems that depend on code coverage information or code structure. Its use in test generation tools, as a method of knowing when to stop the test generation without running the entire test suite may be of key importance.

Acknowledgment

This work is financed by the ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project PTDC/EIA-CCO/116796/2010.

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 - MUTATION*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 89–98.

[2] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. Van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[3] C. Liu, L. Fei, X. Yan, J. Han, and S. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Transactions on Software Engineering (TSE)*, vol. 32, no. 10, pp. 831–848, 2006.

[4] E. Wong, T. Wei, Y. Qi, and L. Zhao, "A crosstab-based statistical method for effective fault localization," in *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST'08)*, R. Hierons and A. Mathur, Eds. Lillehammer, Norway: IEEE Computer Society, 9 – 11 April 2008, pp. 42–51.

[5] A. Perez, *Dynamic Code Coverage with Progressive Detail Levels*. MSc Thesis, Faculdade de Engenharia da Universidade do Porto, 2012.

[6] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," in *Proceedings of the 6th European Software Engineering conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC '97/FSE-5. New York, NY, USA: Springer-Verlag New York, Inc., 1997, pp. 432–449.

[7] Q. Yang, J. J. Li, and D. Weiss, "A survey of coverage based testing tools," in *Proceedings of the 2006 international workshop on Automation of software test*, ser. AST '06. New York, NY, USA: ACM, 2006, pp. 99–103.

[8] A. K. Jain and R. C. Dubes, *Algorithms for clustering data*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988.

[9] 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*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 273–282.

[10] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, J. Grundy and J. Penix, Eds. Montreal, Canada: IEEE Computer Society, 6 – 10 October 2003, pp. 30–39.

[11] C. Liu and J. Han, "Failure proximity: a fault localization-based approach," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT FSE'06)*, M. Young and P. T. Devanbu, Eds. Portland, Oregon, USA: ACM Press, 5 – 11 November 2006, pp. 46–56.

[12] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)*, V. Sarkar and M. W. Hall, Eds. Chicago, Illinois, USA: ACM Press, 12 – 15 June 2005, pp. 15–26.

[13] B. Liblit, "Cooperative debugging with five hundred million test cases," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'08)*, B. G. Ryder and A. Zeller, Eds. Seattle, Washington, USA: ACM Press, 20 – 24 July 2008, pp. 119–120.

[14] P. A. Nainar, T. Chen, J. Rosin, and B. Liblit, "Statistical debugging using compound boolean predicates," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'07)*, D. S. Rosenblum and S. G. Elbaum, Eds. London, UK, July: ACM Press, 9 – 12 July 2007, pp. 5–15.

[15] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: simultaneous identification of multiple bugs," in *Proceedings of the 23rd International Conference on Machine Learning (ICML'06)*, ser. ACM International Conference Proceeding Series, W. W. Cohen and A. Moore, Eds., vol. 148. Pittsburgh, Pennsylvania, USA: ACM Press, 25 – 29 June 2006, pp. 1105–1112.

[16] C. Yilmaz, A. M. Paradkar, and C. Williams, "Time will tell: fault localization using time spectra," in *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds. Leipzig, Germany: ACM Press, 10 – 18 May 2008, pp. 81–90.

[17] 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*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 34–44.

[18] J. Xu, W. K. Chan, Z. Zhang, T. H. Tse, and S. Li, "A dynamic fault localization technique with noise reduction for java programs," in *Proceedings of the 11th Int. Conference on Quality Software (QSIC 2011)*, 2011, pp. 11–20.