A Dynamic Code Coverage Approach to Maximize Fault Localization Efficiency

Alexandre Perez[a], Rui Abreu[a], André Riboira[a]

[a] *Department of Informatics Engineering*
Faculty of Engineering, University of Porto
Porto, Portugal

Abstract

Spectrum-based fault localization is amongst the most effective techniques for automatic fault localization. However, abstractions of program execution traces, one of the required inputs for this technique, require instrumentation of the software under test at a statement level of granularity in order to compute a list of potential faulty statements. This introduces a considerable overhead in the fault localization process, which can even become prohibitive in, e.g., resource constrained environments. To counter this problem, we propose a new approach, coined Dynamic Code Coverage (DCC), aimed at reducing this instrumentation overhead. This technique, by means of using coarser instrumentation, starts by analyzing coverage traces for large components of the system under test. It then progressively increases the instrumentation detail for faulty components, until the statement level of detail is reached. To assess the validity of our proposed approach, an empirical evaluation was performed, injecting faults in six real-world software projects. The empirical evaluation demonstrates that the dynamic code coverage approach reduces the execution overhead that exists in spectrum-based fault localization, and even presents a more concise potential fault ranking to the user. We have observed execution time reductions of 27% on average and diagnostic report size reductions of 77% on average.

Keywords: dynamic coverage, software diagnosis, spectrum-based fault localization

1. Introduction

Automatic fault localization techniques aid developers/testers to pinpoint the root cause of failures, thereby reducing the debugging effort. Amongst the most diagnostic effective techniques is spectrum-based fault localization (SFL). SFL is a statistical technique that uses abstraction of program traces (also

Email addresses: alexandre.perez@fe.up.pt (Alexandre Perez), rui@computer.org (Rui Abreu), andre.riboira@fe.up.pt (André Riboira)

known as program spectra) to correlate software component (e.g., statements, methods, classes) activity with program failures (Abreu et al., 2009c; Liu et al., 2006; Wong et al., 2008). As SFL is typically used to aid developers in identifying the root cause of observed failures, it is used with low-level of granularity (*i.e.*, statement level).

Statistical approaches are very attractive due to the relatively small overhead with respect to CPU time and memory requirement (Abreu et al., 2009c,b). However, gathering the input information, per test case, to compute the diagnostic ranking may still impose a considerable (CPU time) overhead. This is particularly the case for resource constrained environments. The effort required to inspect SFL's diagnostic report is also noteworthy.

As said before, typically, SFL is used at development-time at a statement level granularity (since debugging requires to locate the faulty statement). But not all components need to be inspected at such detailed granularity. In fact, components that are unlikely to be faulty do not need to be inspected. With this reasoning in mind, we propose a technique, coined Dynamic Code Coverage (DCC), that automatically adjusts the granularity per component. First, our approach instruments the source code using a coarse granularity (e.g., package level in Java), and then decides which components to *expand* based on the output of the fault localization technique. With expanding we mean changing the granularity of the instrumentation (e.g., in Java, for instance, instrument classes, then methods, and finally statements). This expansion can be done in different ways, for instance, by selecting the top ranked components, according to a set percentage.

Our empirical evaluation demonstrates that DCC has the potential to drastically reduce the execution overhead, while still maintaining the diagnostic effectiveness of statement-based spectrum-based fault localization. In our experiments, we have observed a time reduction of 27% on average. Furthermore, a 77% reduction of the diagnostic report size was observed in our empirical evaluation, lessening the effort required by developers to perform an inspection.

In particular, this paper makes the following contributions:

- We propose DCC, a technique that automatically decides the instrumentation granularity per module in the system.
- We provide an implementation of the DCC approach within the GZoltar (Campos et al., 2012) testing framework;
- An empirical study to validate the proposed technique, demonstrating its efficacy and efficiency using real-world, large programs. The empirical results shows that DCC can indeed decrease the overhead imposed in the software under test, while still maintaining the same diagnostic accuracy as current approaches to fault localization. DCC also decreases the diagnostic report size when compared to traditional SFL.

This work builds on top of previous work (Perez et al., 2012), where we proposed a lightweight topology-based model to estimate the diagnostic efficiency

of fault localization techniques, extending it as follows. First we provide a motivation for a hierarchical approach to fault localization. Second, we detail our proposed technique, coined DCC. Finally, we provide an empirical evaluation of the efficacy and efficiency of both DCC and our topology-based analysis model by injecting single and multiple faults into real-world, large applications.

The remainder of this paper is organized as follows. In Section 2 we present concepts relevant to this paper as well as a motivational example for our work. In Section 3 the dynamic code coverage approach, DCC, is described. In Section 4, a topology-based analysis to assess whether to use Spectrum-based Fault Localization (SFL) or DCC is detailed. In Section 5 the findings of our empirical evaluation are presented. We compare DCC with related work in Section 6. In Section 7 we conclude and discuss future work.

2. Concepts & Motivational Example

In this section, we introduce the concept of program spectra, and its use in fault localization. Throughout this paper, we use the following terminology (Avižienis et al., 2004):

- A *failure* is an event that occurs when delivered service deviates from correct service.
- An *error* is a system state that may cause a failure.
- A *fault* (defect/bug) is the cause of an error in the system.

In this paper, we apply this terminology to software programs, where faults are bugs in the program code. Failures and errors are symptoms caused by faults in the program. The purpose of fault localization is to pinpoint the root cause of observed symptoms.

Definition 1 *A software program* Π *is formed by a sequence* M *of one or more statements.*

Given its dynamic nature, central to the fault localization technique considered in this paper is the existence of a test suite.

Definition 2 *A test suite* $T = \{t_1, \ldots, t_N\}$ *is a collection of test cases that are intended to test whether the program follows the specified set of requirements. The cardinality of* T *is the number of test cases in the set* $|T| = N$*.*

Definition 3 *A test case* t *is a* (i, o) *tuple, where* i *is a collection of input settings or variables for determining whether a software system works as expected or not, and* o *is the expected output. If* $\Pi(i) = o$ *the test case passes, otherwise fails.*

2.1. Program Spectra

A program spectrum is a characterization of a program's execution on a dataset (Reps et al., 1997). 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. Various different program spectra exist (Harrold et al., 2000), such as path-hit spectra, data-dependence-hit spectra, and block-hit spectra.

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 (Yang et al., 2006). 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.

2.2. Fault Localization

A fault localization technique that uses program spectra, called SFL, exploits information from passed and failed system runs. A passed run is a program execution that is completed correctly, and a failed run is an execution where an error was detected (Abreu et al., 2009c). The criteria for determining if 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 hit spectra (Abreu et al., 2009c).

The hit 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 a N-length vector e, called the error vector. The pair (A, e) serves as input for the SFL technique, as seen in Figure 1.

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

Figure 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 (Jain and Dubes, 1988). One of the most effective is the Ochiai coefficient (Abreu et al., 2007), used in the molecular biology domain:

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

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

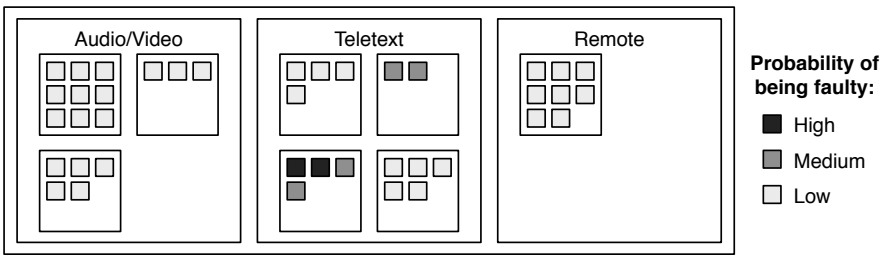


Figure 2: SFL output example.

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

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

SFL can be used with program spectra of several different granularities. However, it is most commonly used ad the line of code (LOC) level and at the basic block level. Using coarser granularities would be difficult for programmers to investigate if a given fault hypothesis generated by SFL was, in fact, faulty. Throughout this work, we will be using a LOC level as the instrumentation granularity for the fault localization diagnosis report.

2.3. Motivational Example

Suppose a program responsible for controlling a television set is being debugged. Consider that such program has three main high-level modules:

1. Audio and video processing;
2. Teletext decoding and navigation;
3. Remote-control input.

If one is to use SFL to pinpoint the root cause of observed failures, hit spectra for the entire application have to be gathered. Furthermore, the hit spectra have to be of a fine granularity, such as LOC level, so that the fault is more easily located.

An output of the SFL technique applied to this specific example can be seen in Figure 2. The smaller squares represent each LOC of the program, which are grouped into methods, and then into the three main modules of the program under test.

As seen in Figure 2, every LOC in the program has an associated fault coefficient that represents the probability of that component being faulty. In this example, the bottom-left function of the teletext decoding and navigation module has two LOCs with high probability of being faulty, and other two with medium probability. The upper-right function of the teletext module also

contains two medium probability LOCs. There are, however, many LOCs with low probability of containing a fault. In fact, in some methods, and even entire modules, such as the audio/video processing and remote-control modules, all components have low probability. Such low probability is an indication that the fault might be located elsewhere, and thus these components need not to be inspected first.

As SFL needs to have information about the entire program spectra to perform an analysis on the most probable fault locations, this can lead to scalability problems, as every LOC has to be instrumented. Instrumentation can hit execution time by as much as 50% in code coverage tools that use similar statement-based instrumentation techniques (Yang et al., 2006). This instrumentation, for each statement in the code, injects a call to a method that will log the activity of that statement. Usually this logging method will flip the hit-spectra matrix bit $A_{i,j}$ where i is the current statement and j is the current system run. As such, a fault localization that uses hit spectra is acceptable for debugging software applications, but may be impractical for large, real-world, and resource-constrained projects that contain hundreds of thousands of LOCs.

In order to make SFL amenable to large, real, and resource-constrained applications, a way to avoid instrumenting the entire program must be devised, while still having a fine granularity for the most probable locations in the results.

3. Dynamic Code Coverage

In order to solve the potential scaling problem that automated fault localization tools have, we propose a dynamic approach, called DCC. The main idea behind this approach is that coarser instrumentation entails less performance overhead as a more detailed instrumentation, as the number of probing instructions needed is smaller. This method uses, at first, a coarser granularity level of instrumentation for the initial program spectra gathering. After that, it progressively increases the instrumentation detail of potential faulty components and re-executes the tests that exercise them.

DCC is shown in Algorithm 1. It takes as parameters $System$, $TestSuite$, $InitialGranularity$ and $FinalGranularity$. These parameters correspond to the System Under Test (SUT), its test suite, and the initial and final instrumentation detail levels, respectively.

First, an empty report $\mathcal{R}$ is created. After that, a list of the components to instrument $\mathcal{F}$ is initialized with all $System$ components. Similarly, the list of test cases to run in each iteration $\mathcal{T}$ is initialized with all test cases in $TestSuite$. An initial granularity $\mathcal{G}$ is also initialized with the desired initial exploration granularity $InitialGranularity$, which can be set from a class level to a LOC level.

After the initial assignments, the algorithm will start its iteration phase in line 6. At the start of each iteration, every component in the list $\mathcal{F}$ is instrumented with the granularity $\mathcal{G}$ with the method INSTRUMENT. What this method does is to alter these components so that their execution is registered

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 ← Update(R, F)
12:         T ← NextTests(TestSuite, A, F)
13:         G ← NextGranularity(F)
14:     until IsFinalGranularity(F,
            FinalGranularity)
15:     return R
16: end procedure
```

in the program spectra. For this execution to be registered, a logging method call is inserted. Depending on the granularity, the instrumentation is performed differently. In a statement-level granularity, each logging call is placed before its corresponding statement, whereas at a method-level the logging call is placed before any other method statements. In the case of a class-level granularity (and assuming we are using Java), we can intercept the classloader request to load that class, and log it as covered.

Afterwards, the test cases $\mathcal{T}$ are run with the method RunTests. Its output is a hit spectra matrix A for all the previously instrumented components, and the error vector e, that states what tests passed and what tests failed. As explained in Section 2.2, these are the necessary inputs for spectrum-based fault localization, performed in line 9. This SFL method calculates, for each instrumented component, its failure coefficient using the Ochiai coefficient, previously shown in equation 1.

Following the fault localization step, the components are passed through a Filter that eliminates the low probability ones according to a set threshold, and the list $\mathcal{F}$ is updated, as well as the fault localization report $\mathcal{R}$.

In line 12, the test case set is updated to run only the tests that touch the current components $\mathcal{F}$. Such tests can be retrieved by analyzing the coverage matrix A.

The last step in the iteration is to update the instrumentation granularity for next iterations. Method NextGranularity finds the coarser granularity in all the components of list $\mathcal{F}$, and updates that granularity to the next level of detail.

Every iteration is tested for recursion with IsFinalGranularity, that returns true if every component in the list $\mathcal{F}$ is at the desired final granularity defined in *FinalGranularity*. This final granularity can be of different detail levels, such as method level or LOC level, according to the needs of the software project being tested. If the IsFinalGranularity condition is not met, a new iteration is performed.

Lastly, the DCC algorithm returns the fault localization report $\mathcal{R}$. $\mathcal{R}$ contains diagnosis candidates of different granularities, typically with the top ones at a finer-grained level of detail.

In each iteration of this algorithm, tests have to be run to generate the program spectra, and then the SFL algorithm is performed. Running the test cases entails a time complexity of $O(N)$ and SFL has a time complexity of $O(M.N)$, where N is the number of test cases and M is the number of components of the system. Although all the filtering operations mentioned in this work entail a complexity of $O(M)$, there may be filters suitable to be plugged into DCC that entail a non-linear complexity. Assuming a worst case scenario, where no components are ever filtered, then the DCC algorithm has a time complexity of $O(I.M.N)$, where I is the total number of iterations performed, and depends on the initial and final granularity parameters (note that, in our approach, the worst case for the number of iterations I is 3). As for the space complexity, the worst case scenario is that all components are explored. Space complexity, like SFL, is $O(M.N)$.

DCC's performance is very dependent on the Filter function, which is responsible to decide whether or not it is required to zoom-in[1] in a given component. Although many filters may be plugged into the algorithm, in this paper we study the impact of the percentile filter P_f. This filter discards components that have a similarity coefficient below a given percentile. For instance, the filter $P_f = 0.80$ discards a component c if its similarity score $s_O(c)$ is lower than 80% of the other scores.

During our initial experiments with the DCC technique, we have also considered a filter that would discard a component c if its similarity $s_O(c)$ was lower than a given threshold. The results obtained (in terms of processing time and diagnostic accuracy) varied greatly across different projects. These inconsistent results happen since this filter depends on the coverage density of the program. For instance, in certain projects, the top ranked components would score a s_O value of 1.0, whereas top ranked component scores from other projects would be closer to 0.5. Thus, we have discarded this filter.

The main advantages of our dynamic code coverage algorithm, DCC, are twofold. The first one is the decrease of instrumentation overhead in the program execution (as demonstrated by the empirical results). This is due to the fact that not every LOC is instrumented – only the LOCs most likely to contain a fault will be instrumented at that level of detail.

The second advantage is the fact that, in every iteration, the generated

[1] In this context, zooming-in is to explore the inner components of a given component.

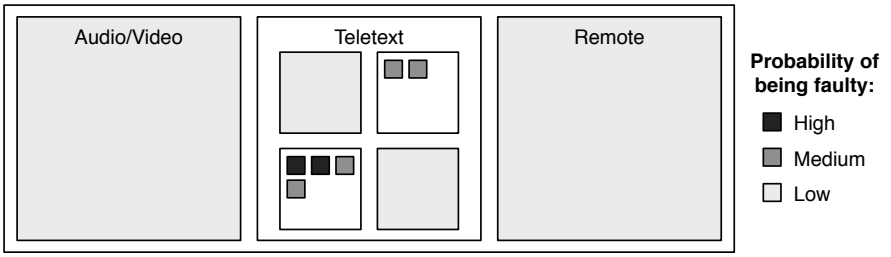


Figure 3: DCC output example.

program spectra matrices, seen in line 8 of Algorithm 1, will be shorter in size when compared to traditional SFL. That way, the fault coefficient calculation, described in Section 2.2, will be inherently faster, as there are fewer components to calculate.

The iterative nature of the DCC algorithm also provides some benefits. In each iteration, the algorithm is walking towards a solution, narrowing down the list of components which are likely to contain a fault. As such some information about those components can be made available, directing the developer to the fault location even before the algorithm is finished. Secondly, as low probability components are being filtered, the final report will also be shorter, providing the developer with a more concise fault localization report.

To illustrate the overhead reduction, let us revisit the motivational example given in Section 2.3. If use the DCC approach to debug this program, we get the output shown in Figure 3. In this example, a filter responsible for not exploring components with low suspiciousness of containing faults is being used. In particular, the algorithm executes as follows:

1. The three modules – Audio/Video, Teletext, and Remote – are instrumented at the module level. Upon running the tests and SFL, the only component with high probability of being faulty is the Teletext module.
2. The Teletext module is instrumented at a method level. After that, the tests that touch the Teletext module are run. Fault localization states that the upper-right (UR) and the bottom-left (BL) functions have medium and high probability of being faulty, respectively.
3. The UR and BL functions are instrumented at the LOC level. After the tests that touch those functions are run and fault localization is performed, every LOC in those functions has an associated fault coefficient. As all the non-filtered components are of LOC granularity, the execution is terminated.

It is worth noting that this approach reports only LOCs which are more likely to contain a fault and, at the same time, requires less software components to be instrumented – 13 in total. Compared to the pure SFL approach of Section 2.3,

where 40 components were instrumented, DCC has reduced instrumentation (thus, overhead) by 67.5%.

4. Topology Model

DCC, as an iterative technique, is aimed at improving the execution time of the fault localization procedure. 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. Even though our approach only re-executes a subset of the test suite, this repeated instrumentation and execution step can incur in a higher overall execution time. This can happen if test cases' activity patterns encompass a significant portion of a project's components as DCC has to re-run all tests that cover suspicious components. A lower detail on component activity will result in the existence of *ambiguity groups* (González-Sanchez et al., 2011a): groups of components with identical coverage signatures, undistinguishable from each other. As spectrum-based techniques rely on differences and similarities between components and the error vector, this will result in a poor diagnostic performance.

This kind of behavior is common for projects that have either a relatively small codebase, or an unbalanced code topology. In both of these instances each test covers a large portion of the code, and the resulting program spectra matrix will be dense, therefore many software components will have to be zoomed in. González-Sanchez et al. (2011c) have shown that the optimal coverage density for fault localization is $\rho = 0.5$, according to the IG (information gain) heuristic (González-Sanchez et al., 2011b). In this section, we describe our analysis methodology for quickly estimating a software project's execution coverage density ρ by inspecting its topology, as a way do decide what fault localization technique to use for that project.

This analysis, initially proposed in (Perez et al., 2012), is intended to provide a general and coarse overview of a project's execution based on its 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.

5. Empirical Evaluation

In this section, we evaluate the validity and performance of the DCC approach for real projects. This approach, as well as the topology analysis model,

were implemented in the GZoltar (Campos et al., 2012) testing framework for Eclipse. First, we introduce the programs under analysis and the evaluation metrics. Then, we discuss the empirical results and finish this section with a threats to validity discussion.

5.1. Experimental Setup

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

- `NanoXML`[2] – a small XML parser.
- `org.jacoco.report` – report generation module for the `JaCoCo`[3] code coverage library.
- `Xstream`[4] – an object serialization library.
- `JGAP`[5] – a genetic algorithms library.
- `XML-Security` – a component library implementing XML signature and encryption standards. This library is part of the Apache Santuario[6] project.
- `JMeter`[7] – a desktop application designed to load test functional behavior and measure performance of web applications.

The project details of each subject are in Table 1. The LOC count information was gathered using the metrics calculation and dependency analyzer plugin for Eclipse `Metrics`[8]. Test count and coverage percentage were collected with the Java code coverage plugin for Eclipse `EclEmma`[9].

Table 1: Experimental Subjects.

Subject	**Version**	**LOCs (M)**	**Test Cases**	**Coverage**
`NanoXML`	2.2.6	5393	8	53.2%
`org.jacoco.report`	0.5.5	5979	225	97.2%
`Xstream`	1.4.3	35944	1418	84.8%
`JGAP`	3.6.2	48590	1377	67.1%
`XML-Security`	1.5.0	60946	461	59.8%
`JMeter`	2.6	127359	593	34.2%

[2]NanoXML – `http://devkix.com/nanoxml.php`
[3]JaCoCo – `http://www.eclemma.org/jacoco/index.html`
[4]Xstream – `http://xstream.codehaus.org/`
[5]JGAP – `http://jgap.sourceforge.net/`
[6]Apache Santuario – `http://santuario.apache.org/`
[7]JMeter – `http://jmeter.apache.org/`
[8]Metrics – `http://metrics.sourceforge.net/`
[9]EclEmma – `http://www.eclemma.org/`

To assess the efficiency and effectiveness of DCC when tackling a single bug and multiple simultaneous bugs, the following experiments were performed, using 25 faulty versions per subject program. As the subject programs are bug-free, we injected common mistakes in the programs: one fault in fifteen versions, and five simultaneous faults in 10 versions and executed:

- Fault localization with SFL. This is the reference baseline.
- 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.

We have used $C = 100$ and $T = 50$ in our topology model. These values were obtained by training the model with a set of software projects for which we had previously labeled the best fault localization technique that suited each specific project. The criteria for choosing one technique over the other was their execution time. The model decides employing DCC over SFL when the score function is $S \leq 1.0$.

The metrics gathered were the fault localization execution time, the size of the fault localization report, and the average LOCs needed to be inspected until the fault is located. The latter metric can be calculated by sorting the fault localization report by the value of the coefficient, and finding the injected fault's position. This metric assumes that the developer performs the inspection in an ordered manner, starting from the highest fault coefficient LOCs.

As spectrum-based fault localization creates a ranking of components in order of likelihood to be at fault, we can retrieve how many components we still need to inspect until we hit a faulty one. Let $d \in \{1, \ldots, K\}$, where K is the number of ranked components and $K \leq M$, be the index of the statement that we know to contain the fault. For all $j \in \{1, \ldots, M\}$, let s_j. Then the ranking position of the faulty statement is given by

$$\tau = \frac{|\{j|s_j > s_d\}| + |\{j|s_j \geq s_d\}| - 1}{2} \tag{4}$$

$|\{j|s_j > s_d\}|$ counts the number of components that outrank the faulty one, and $|\{j|s_j \geq s_d\}|$ counts the number of components that rank with the same probability as the faulty one plus the ones that outrank it. In the case of multiple faults, we are considering the first faulty component in the ranking (see Steimann et al. (2013) for more information). As the user does not know *a priori* how many faults the system has, it is assumed that he/she re-runs the fault localization technique after fixing the faulty component.

We define quality of diagnosis as the effectiveness to pinpoint the faulty component. As said before, this metric represents the percentage of components that need not be considered when searching for the fault by traversing the ranking. It is defined as

$$(1 - \frac{\tau}{K_{SFL}}) \cdot 100\% \tag{5}$$

where K_{SFL} is the number of ranked components of SFL without DCC – the reference baseline.

For each faulty version, we have also gathered the execution time of the test suite, without any instrumentation. This will serve as a reference for the theoretical lower bound of fault localization techniques in terms of execution time (*i.e.*, no instrumentation overhead). However, note that all other execution times besides this test execution include a GZoltar bootstrapping phase where, for instance, the code tree is traversed and every component is assigned a unique id.

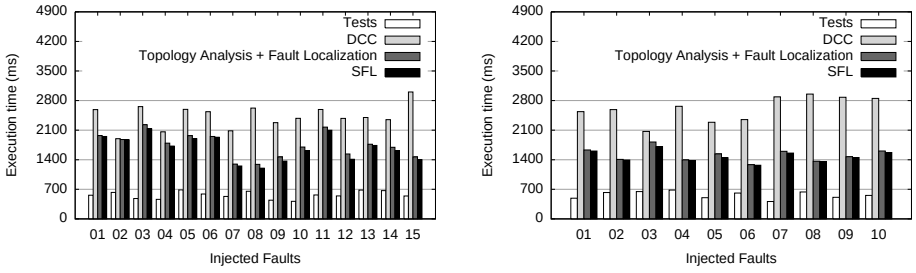
The experiments were run on a 2.7 GHz Intel Core i7 MacBook Pro with 4 GB of RAM, running OSX Lion.

5.2. Experimental Results

Figures 4 to 9 summarize the overall execution time outcomes for all experimental subjects. These results are gathered by running the entire fault localization experiments detailed in the previous section, and do not pertain only to the instrumentation overhead. Also shown is the execution time of running the test suite without any instrumentation, evidencing the optimal lower bound of fault localization techniques that require test executions. Note that the difference between test execution times and fault localization execution times is not only due to instrumentation overheads, but also other factors, such as the similarity coefficient calculation and initial bootstrapping.

Due to space constraints, only one DCC filter is shown. We have chosen to show the $P_f = 70\%$ filter, since it was the best performing filter of those considered, and is able to find the injected faults for every experiment (*i.e.*, the resulting diagnostic report contains the injected fault).

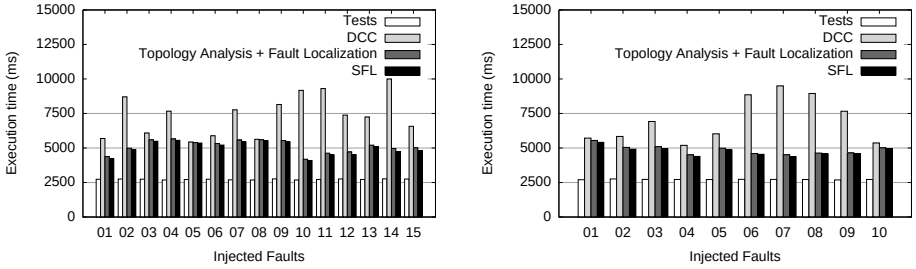
The first two subjects to be analyzed were `NanoXML` and `org.jacoco.report`, whose experimental results can be seen in Figures 4 and 5. As a result of having a small codebase, both projects are scored above the exploration threshold by the topology-based analysis. This means that the recommended fault localization method to be used in these projects is SFL. As the aforementioned figures depict, the SFL execution time is indeed better than that of DCC.



(a) Single fault

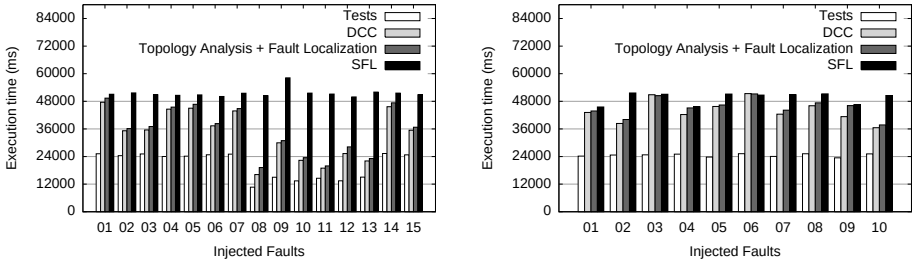
(b) 5 simultaneous faults

Figure 4: `NanoXML` execution time results.



(a) Single fault (b) 5 simultaneous faults

Figure 5: `org.jacoco.report` execution time results.



(a) Single fault (b) 5 simultaneous faults

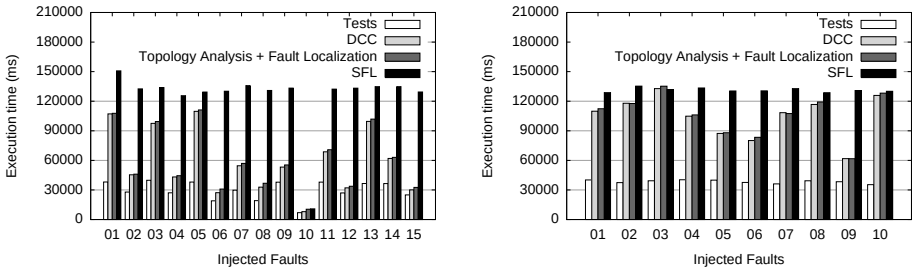
Figure 6: `Xstream` execution time results.

These kinds 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 Sections 2.1 and 2.2 will be rather dense. Because of this, many components would have similar coefficients, rendering the filtering operation ineffective, keeping a lot of components to be re-instrumented and re-tested in subsequent iterations. Therefore, in these particular cases, the use of DCC's repeated instrumentation and testing incurs in an increase of the overall execution time.

`NanoXML` and `org.jacoco.report` show, then, a very similar execution time performance using both approaches. As seen in Table 3, the diagnostic accuracy of both approaches is the same. The diagnostic report sizes of the two approaches are identical for every execution, and so is the quality of diagnosis – 79% on average (standard deviation: $\sigma = 0.20$).

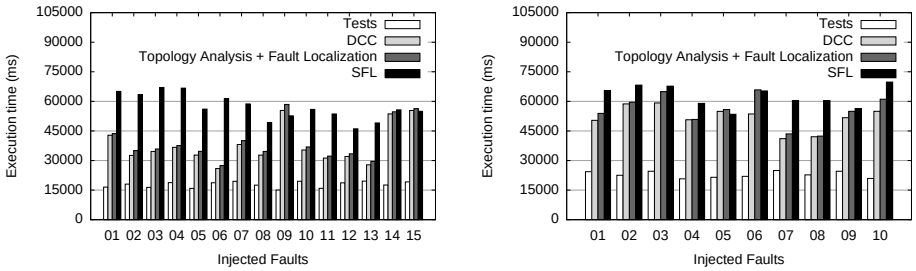
The other test subjects, `Xstream` (Figure 6), `JGAP` (Figure 7), `XML-Security` (Figure 8) and `JMeter` (Figure 9) are scored below the exploration threshold, and thus our topology model indicates that they should by explored with DCC.

All of these test subjects show improvements in performance when DCC is used, even when more than one fault is present. This is due to the fact that



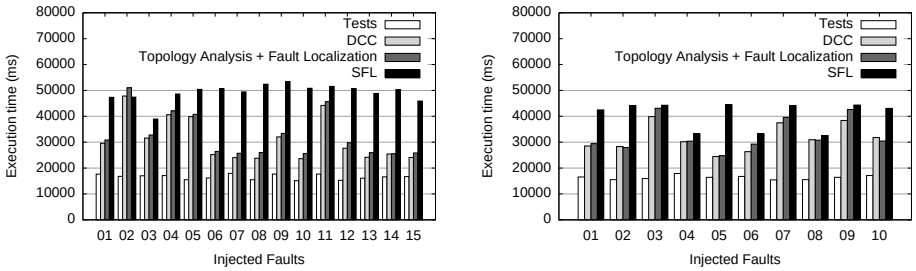
(a) Single fault (b) 5 simultaneous faults

Figure 7: `JGAP` execution time results.



(a) Single fault (b) 5 simultaneous faults

Figure 8: `XML-Security` execution time results.



(a) Single fault (b) 5 simultaneous faults

Figure 9: `JMeter` execution time results.

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 1), 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.

Using DCC when appropriate (*i.e.*, using the topology model to decide the fault localization method) can, then, reduce the execution time by 27% on average ($\sigma = 0.24$) in this kind of projects. We can see from Table 2 a comparison for the average execution times for both SFL and DCC for all projects and their respective number of statements (LOCs), as well as the execution times of the test cases without any kind of instrumentation. The test execution times can give us an insight on the theoretical lower bound times for any dynamic analysis that uses test information. The execution times for both SFL and DCC encompass all steps of these techniques, including instrumentation, execution and diagnostic report generation. We have refrained from presenting individual overheads for the different steps, as we are interested in the algorithms as a whole. As evidenced from the previous discussion, projects with higher program sizes benefit from using DCC. The other metrics gathered in this empirical evaluation also show a consistent improvement over SFL in every project.

On average, the DCC approach reduced 77% ($\sigma = 0.16$) the generated fault localization ranking, providing a more concise report when compared to SFL. This is due to the fact that many coarsely detailed components (*e.g.*, classes or methods) were filtered out of the algorithm, and thus were not explored and ranked at the line of code level. Therefore, the report size is reduced when compared to SFL because the detail granularity changes the number of components.

Table 2: Execution times and DCC time reduction.

Subject	**LOCs**	**Tests (s)**	**Execution Time (s)**		**DCC Time Reduction (%)**
			SFL	**DCC**	
`NanoXML`	5393	0.6	1.6	2.6	-62.5
`org.jacoco.report`	5979	2.7	4.9	7.3	-49.0
`Xstream`	35944	22.0	50.4	39.1	22.4
`JGAP`	48590	33.2	126.9	75.3	40.7
`XML-Security`	60946	19.8	62.6	47.7	23.8
`JMeter`	127359	16.5	45.1	33.9	24.8

Once again, we can see from Table 3 that the diagnostic accuracy has remained practically unaltered when comparing DCC to traditional SFL. In fact, the quality of diagnosis, described in equation 5, only exhibited a slight improvement, from 95% ($\sigma = 0.09$) without DCC to 96% ($\sigma = 0.09$) with DCC. Note that, for multiple faults, the order by which multiple faults appear in the ranking is the same in both SFL and DCC, meaning that both techniques will lead to the same diagnostic results.

Table 3: Number of statements inspected until the fault is located (τ).

	τ	
Subject	**SFL**	**DCC**
`NanoXML`	81.6	81.6
`org.jacoco.report`	13.1	13.1
`Xstream`	9.1	9.7
`JGAP`	42.0	45.8
`XML-Security`	13.3	16.6
`JMeter`	16.9	17.1

Although diagnostic accuracy remains unchanged, we should point out that our τ metric assumes that developers inspect faults by their suspiciousness ranking. However, according to Parnin and Orso (2011), this is not always the case: developers do not closely follow a statement ranking computed by fault localization tools. Therefore, the fact that our approach produces less suspicious components (as the diagnostic report size is considerably smaller) is indeed an important and meaningful feature of the DCC technique. Another example where a reduced report size is useful is in the use of code visualizations to represent the diagnostic report (e.g., (Gouveia et al., 2013)). Smaller diagnostic reports yield less *cluttered* visualizations, allowing users to focus on the more suspicious components.

It is worth to note that for percentile filters whose threshold was higher than 70%, some faults were not found in the diagnostic report, because they were not explored. Percentile filters below 70% presented slower fault localization results. When we decrease this threshold in the filter, the diagnostic report size increases, until it reaches the original size. It is also slower because, in every iteration, more components are instrumented and also, as a consequence, more tests need to be executed. Based on our empirical evidence, we recommend a $P_f = 0.70$ when DCC is used to perform fault localization.

5.3. Threats to Validity

The main threat to external validity of these empirical results is the fact that only six test subjects were considered. Although the subjects were all real, open source software projects, it is plausible to assume that a different set of subjects, having inherently different characteristics, may yield different results. Other threat to external validity is related to the injected faults used in the experiments. These faulty program versions, despite being 25 in total for each experimental subject, may not represent the entire conceivable software fault spectrum. Also, potentially, the overhead exhibited by the instrumentation is different in other software programs. For instance, if programs need to wait for a network reply, the execution time after instrumentation will be only a little more than that of the original execution time. One solution to mitigate this problem is to define the starting detail granularity of the instrumentation in

a per-component basis, so that these code regions can be manually configured to be executed only once. Furthermore, the topology model values C and T, trained with a set of manually labeled projects, may not generalize for every kind of software project.

Threats to internal validity are related to some fault in the DCC implementation, or any underlying implementation, such as SFL or even the instrumentation for gathering program spectra. To minimize this risk, some testing and individual result checking were performed before the experimental phase.

6. Related Work

The process of pinpointing the fault(s) that led to symptoms (failures/errors) is called fault localization, and has been an active area of research for the past decades. Based on a set of observations, automatic approaches to software fault localization yield a list of likely fault locations, which is subsequently used either by the developer to focus the software debugging process. Depending on the amount of knowledge that is required about the system's internal component structure and behavior, the most predominant approaches can be classified as (1) statistical approaches or (2) reasoning approaches. The former approach uses an abstraction of program traces, dynamically collected at runtime, to produce a list of likely candidates to be at fault, whereas the latter combines a static *model* of the expected behavior with a set of observations to compute the diagnostic report.

Statistics-based fault localization techniques, as stated above, use an abstraction of program traces, also known as *program spectra*, to find a statistical relationship with observed failures. Program spectra are collected at run-time, during the execution of the program, and many different forms exist (Harrold et al., 2000). For example, component-hit spectra indicate whether a component was involved in the execution of the program or not. In contrast to model-based approaches, program spectra and pass/fail information are the only *dynamic* source of information used by statistics-based techniques.

Well-known examples of statistical approaches are the Tarantula tool by Jones and Harrold (2005), the Nearest Neighbor technique by Renieris and Reiss (2003), the Sober tool by Liu et al. (2006), the work of Liu and Han (2006), CrossTab by Wong et al. (2008), the Cooperative Bug Isolation (CBI) by Liblit and his colleagues (Liblit et al., 2005; Liblit, 2008; Nainar et al., 2007; Zheng et al., 2006), the Time Will Tell approach by Yilmaz et al. (2008), MKBC by Xu et al. (2011) and the causal inference approach by Baah et al. (2010). 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.

A statistics-based approach that also uses re-instrumentation is HOLMES (Chilimbi et al., 2009). This approach tries to avoid instrumentation overhead by initially profiling only the parts of code that more likely contain the root causes of bugs. These code parts are selected based on stack traces of coverage reports, and a static analysis on the system.

Next, after re-execution, HOLMES analyses the partial profiles to compute a statistical model of the program. If the model identifies bug predictors that explain the failures, HOLMES returns them as diagnostic candidates. If the model is inconclusive, then HOLMES increases the number of code parts to be instrumented, using dependency analysis. In contrast, DCC starts with a lightweight, coarse instrumentation of the program, and iteratively narrows the search space (by increasing the instrumentation detail in a subset of components) to compute the suspicious diagnostic candidates.

Toolsets providing fault localization using spectrum-based fault localization exist, namely in Zoltar (Janssen et al., 2009) and Tarantula (Jones et al., 2002; Jones and Harrold, 2005) for C projects, and GZoltar (Riboira et al., 2011) for Java projects. However, none of these tools employ a dynamic code coverage approach to SFL, having to instrument the entire SUT. Also, their instrumentation granularity is set at a LOC level of detail. A DCC approach could certainly be added to any of these tools, with minimal algorithmic changes, provided that the underlying instrumentation tool that these tools use to gather program spectra supports different levels of detail.

Statistical approaches require several traces, and their pass/fail information, in order to obtain an accurate diagnosis. Traces also need to be varied, or else would result in the existence of *ambiguity groups* (González-Sanchez et al., 2011a) (*i.e.* groups of components with identical coverage signatures, undistinguishable from each other). The commonly used source for execution traces is the SUT's test suite, which may not be of sufficient quality to obtain accurate diagnostic cues. Some approaches have been proposed to generate test cases (Campos et al., 2013) and/or to select the best test cases (Baudry et al., 2006) to maximize the fault localization efficiency. Both of these approaches are orthogonal to our DCC approach, and the contributions are complementary: the test suite can be generated/selected and then used to locate faults with DCC.

Reasoning approaches to fault localization use prior knowledge of the system, such as required component behavior and interconnection, to build a model of the correct behavior of the system. An example of a reasoning technique is model-based diagnosis (see, e.g., (de Kleer and Williams, 1987)), where a diagnosis is obtained by logical inference from the *static* model of the system, combined with a set of run-time observations. In the software engineering community this approach is often called model-based software debugging (Mayer and Stumptner, 2007). Well-known approaches to model-based software debugging include the approaches of Friedrich et al. (1996, 1999), Nica and Wotawa (2008), Wotawa et al. (2002), and Mayer and Stumptner (2007).

As model-based techniques technique may suffer from large diagnostic results and not scale to sizable projects, some work was already combining SFL with Model-Based Software Debugging (MBSD) has been proposed (Mayer et al., 2008; Abreu et al., 2009a), where MBSD is used to refine the output report generated by the spectrum-based fault localization, filtering the components that do not explain the observed failures. Our DCC approach could also be combined with this technique, by performing the fault localization until a certain middle-grained level of component detail (*e.g.*, method level), and submit the

top components to be analyzed by MBSD.

Recent approaches to fault localization include the work of Burger and Zeller (2011) and Rößler et al. (2012). These works use *experimental approaches*: techniques that may alter inputs and object iterations, among others, to narrow the diagnostic report. Provided they have at least one failing test, these approaches can generate subsequent test cases to increase diagnostic accuracy. The downside of these approaches is the substantial overhead required to generate test cases.

7. Conclusions & Future Work

We have shown that current approaches to spectrum-based fault localization face some challenges concerning scalability due to the input gathering overhead caused by a fine grained instrumentation throughout the system under test. For instance, this may be an issue in resource-constrained systems. A solution to this problem was presented, coined Dynamic Code Coverage (DCC), that initially uses a coarser granularity of instrumentation, and progressively increases the instrumentation detail of potential faulty components. In our empirical evaluation, we have validated our approach, and demonstrated that it not only reduces the average execution time by 27%, but also reduces the average number of components reported to the user by 77%, lessening the report inspection burden.

As for future work, some aspects of the dynamic code coverage technique still require further investigation. One of those is the way of how the initial system granularity is established. Currently, this value is set manually and is the same across the entire system under test. A way to change this would be by using static analysis to assess program information and to adjust the system's initial granularity accordingly. Another approach would be to learn what were the most frequently expanded components from previous executions, and change these components' initial granularity independently. Another issue that requires further investigation pertains to the filtering methods. It is possible that there are better filtering methods than those presented in this paper, namely methods that employ dynamic strategies, that change the cutting threshold based on program spectra analysis.

Acknowledgements

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

Abreu, R., Mayer, W., Stumptner, M., van Gemund, A.J.C., 2009a. Refining spectrum-based fault localization rankings, in: Proceedings of the 2009 ACM Symposium on Applied Computing, SAC 2009, pp. 409–414.

Abreu, R., Zoeteweij, P., van Gemund, A.J.C., 2007. On the accuracy of spectrum-based fault localization, in: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, pp. 89–98.

Abreu, R., Zoeteweij, P., van Gemund, A.J.C., 2009b. Spectrum-based multiple fault localization, in: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering, ASE 2009, pp. 88–99.

Abreu, R., Zoeteweij, P., Golsteijn, R., van Gemund, A.J.C., 2009c. A practical evaluation of spectrum-based fault localization. Journal of Systems and Software 82, 1780–1792.

Avižienis, A., Laprie, J.C., Randell, B., Landwehr, C.E., 2004. Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable Secure Computing 1, 11–33.

Baah, G.K., Podgurski, A., Harrold, M.J., 2010. Causal inference for statistical fault localization, in: Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA 2010, pp. 73–84.

Baudry, B., Fleurey, F., Traon, Y.L., 2006. Improving test suites for efficient fault localization, in: Proceedings of the 28th International Conference on Software Engineering, ICSE 2006, pp. 82–91.

Burger, M., Zeller, A., 2011. Minimizing reproduction of software failures, in: Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, pp. 221–231.

Campos, J., Abreu, R., Fraser, G., d'Amorim, M., 2013. Entropy-Based Test Generation for Improved Fault Localization, in: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, pp. 257–267.

Campos, J., Riboira, A., Perez, A., Abreu, R., 2012. GZoltar: an eclipse plug-in for testing and debugging, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, pp. 378–381.

Chilimbi, T.M., Liblit, B., Mehra, K.K., Nori, A.V., Vaswani, K., 2009. Holmes: Effective statistical debugging via efficient path profiling, in: Proceedings of the 31st International Conference on Software Engineering, ICSE 2009, pp. 34–44.

Friedrich, G., Stumptner, M., Wotawa, F., 1996. Model-based diagnosis of hardware designs, in: Proceedings of the 12th European Conference on Artificial Intelligence, ECAI 1996, pp. 491–495.

Friedrich, G., Stumptner, M., Wotawa, F., 1999. Model-based diagnosis of hardware designs. Artificial Intelligence 111, 3–39.

González-Sanchez, A., Abreu, R., Gross, H.G., van Gemund, A., 2011a. Prioritizing tests for fault localization through ambiguity group reduction, in: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, pp. 83–92.

González-Sanchez, A., Abreu, R., Gross, H.G., van Gemund, A.J.C., 2011b. Spectrum-based sequential diagnosis, in: Proceedings of the 25th AAAI Conference on Artificial Intelligence, AAAI 2011, pp. 189–196.

González-Sanchez, A., Gross, H.G., van Gemund, A.J.C., 2011c. Modeling the diagnostic efficiency of regression test suites, in: Proceedings of the 4th International IEEE Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2011, pp. 634–643.

Gouveia, C., Campos, J., Abreu, R., 2013. Using HTML5 visualizations in software fault localization, in: Proceedings of the 1st IEEE Working Conference on Software Visualization, VISSOFT 2013, pp. 1–10.

Harrold, M.J., Rothermel, G., Sayre, K., Wu, R., Yi, L., 2000. An empirical investigation of the relationship between spectra differences and regression faults. Software Testing, Verification, and Reliability 10, 171–194.

Jain, A.K., Dubes, R.C., 1988. Algorithms for clustering data. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Janssen, T., Abreu, R., van Gemund, A.J.C., 2009. Zoltar: A toolset for automatic fault localization, in: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering, ASE 2009, pp. 662–664.

Jones, J.A., Harrold, M.J., 2005. Empirical evaluation of the tarantula automatic fault-localization technique, in: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE 2005, pp. 273–282.

Jones, J.A., Harrold, M.J., Stasko, J.T., 2002. Visualization of test information to assist fault localization, in: Proceedings of the 22nd International Conference on Software Engineering, ICSE 2002, pp. 467–477.

de Kleer, J., Williams, B.C., 1987. Diagnosing multiple faults. Artificial Intelligence 32, 97–130.

Liblit, B., 2008. Cooperative debugging with five hundred million test cases, in: Proceedings of the 2008 International Symposium on Software Testing and Analysis, ISSTA 2008, pp. 119–120.

Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I., 2005. Scalable statistical bug isolation, in: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, PLDI 2005, pp. 15–26.

Liu, C., Fei, L., Yan, X., Han, J., Midkiff, S., 2006. Statistical debugging: A hypothesis testing-based approach. IEEE Transactions on Software Engineering 32, 831–848.

Liu, C., Han, J., 2006. Failure proximity: a fault localization-based approach, in: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, pp. 46–56.

Mayer, W., Abreu, R., Stumptner, M., van Gemund, A.J., 2008. Prioritizing model-based debugging diagnostic reports, in: Proceedings of the 19th International Workshop on Principles of Diagnosis, DX 2008, pp. 127–134.

Mayer, W., Stumptner, M., 2007. Model-Based Debugging State of the Art And Future Challenges. Electronic Notes in Theoretical Computer Science 174, 61–82.

Nainar, P.A., Chen, T., Rosin, J., Liblit, B., 2007. Statistical debugging using compound boolean predicates, in: Proceedings of the 2007 International Symposium on Software Testing and Analysis, ISSTA 2007, pp. 5–15.

Nica, M., Wotawa, F., 2008. From constraint representations of sequential code and program annotations to their use in debugging, in: Proceedings of the 18th European Conference on Artificial Intelligence, ECAI 2008, pp. 797–798.

Parnin, C., Orso, A., 2011. Are automated debugging techniques actually helping programmers?, in: Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, pp. 199–209.

Perez, A., Riboira, A., Abreu, R., 2012. A topology-based model for estimating the diagnostic efficiency of statistics-based approaches, in: Proceedings of the 23rd IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops 2012, pp. 171–176.

Renieris, M., Reiss, S.P., 2003. Fault localization with nearest neighbor queries, in: Proceedings of the 18th IEEE International Conference on Automated Software Engineering, ASE 2003, pp. 30–39.

Reps, T.W., Ball, T., Das, M., Larus, J.R., 1997. The use of program profiling for software maintenance with applications to the year 2000 problem, in: Proceedings 6th European Software Engineering Conference Held Jointly with the 5th Symposium on Foundations of Software Engineering, ESEC/FSE 1997, pp. 432–449.

Riboira, A., Abreu, R., Rodrigues, R., 2011. An OpenGL-based eclipse plug-in for visual debugging, in: Proceedings of the 1st Workshop on Developing Tools as Plug-ins, TOPI 2011, pp. 60–60.

Rößler, J., Fraser, G., Zeller, A., Orso, A., 2012. Isolating failure causes through test case generation, in: Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012, pp. 309–319.

Steimann, F., Frenkel, M., Abreu, R., 2013. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators, in: Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA 2013, pp. 314–324.

Wong, W.E., Wei, T., Qi, Y., Zhao, L., 2008. A crosstab-based statistical method for effective fault localization, in: Proceedings of the 1st International Conference on Software Testing, Verification, and Validation, ICST 2008, pp. 42–51.

Wotawa, F., Stumptner, M., Mayer, W., 2002. Model-based debugging or how to diagnose programs automatically, in: Proceedings of the 15th International Conference on Industrial and Engineering, Applications of Artificial Intelligence and Expert Systems, IEA/AIE 2002, pp. 746–757.

Xu, J., Chan, W.K., Zhang, Z., Tse, T.H., Li, S., 2011. A dynamic fault localization technique with noise reduction for java programs, in: Proceedings of the 11th International Conference on Quality Software, QSIC 2011, pp. 11–20.

Yang, Q., Li, J.J., Weiss, D.M., 2006. A survey of coverage based testing tools, in: Proceedings of the 2006 International Workshop on Automation of Software Test, AST 2006, pp. 99–103.

Yilmaz, C., Paradkar, A.M., Williams, C., 2008. Time will tell: fault localization using time spectra, in: Proceedings of the 30th International Conference on Software Engineering, ICSE 2008, pp. 81–90.

Zheng, A.X., Jordan, M.I., Liblit, B., Naik, M., Aiken, A., 2006. Statistical debugging: simultaneous identification of multiple bugs, in: Proceedings of the 23rd International Conference on Machine Learning, ICML 2006, pp. 1105–1112.