Contextualizing Spectrum-based Fault Localization

Higor A. de Souza\textsuperscript{a,}\textsuperscript{*}, Danilo Mutti\textsuperscript{b}, Marcos L. Chaim\textsuperscript{b}, Fabio Kon\textsuperscript{a}

\textsuperscript{a}Department of Computer Science, Institute of Mathematics and Statistics, University of São Paulo, Rua do Matão, 1010, São Paulo, SP, 05508-090, Brazil
\textsuperscript{b}School of Arts, Sciences, and Humanities, University of São Paulo, Rua Arlindo Béttio, 1000, São Paulo, SP, 03828-000, Brazil

Abstract

Context: Fault localization is among the most expensive tasks in software development. Spectrum-based fault localization (SFL) techniques seek to pinpoint faulty program elements (e.g., statements), by sorting them only by their suspiciousness scores. Developers tend to fall back on another debugging strategy if they do not find the bug in the first positions of a suspiciousness list.

Objective: In this study, we assess techniques to contextualize code inspection whose goal is two-fold: to provide guidance during fault localization, and to improve the effectiveness of SFL techniques in classifying bugs within the first picks. Code Hierarchy (CH) and Integration Coverage-based Debugging (ICD) techniques provide a search roadmap—a list of methods—that guide the developer toward faults. CH assigns a method with the highest suspiciousness score of its blocks, and ICD captures method call relationships from testing to establish the roadmap. Two new filtering strategies—Fixed Budget (FB) and Level Score (LS)—are combined with ICD and CH for reducing the amount of blocks to inspect in each method.

Method: We evaluated the effectiveness of ICD, CH, FB, LS, and a suspiciousness block list (BL) on 62 bugs from 7 real programs.

Results: ICD and CH using FB found more faults inspecting less blocks than BL with statistical significance. More than 50\% of the faults were found inspecting

\textsuperscript{*}Corresponding author

Email addresses: hamario@ime.usp.br (Higor A. de Souza), dmutti@gmail.com (Danilo Mutti), chain@usp.br (Marcos L. Chaim), kon@ime.usp.br (Fabio Kon)
at most 10 blocks using ICD-FB and CH-FB. Moreover, ICD and CH located 70% of the faults by inspecting, at most, 4 methods.

Conclusions: These results suggest that the contextualization provided by roadmaps and filtering strategies is useful for guiding developers toward faults and improves the performance of SFL techniques.

Keywords: spectrum-based fault localization, automated debugging, contextual information, method-level spectrum, code structure

1. Introduction

The existence of faults in programs is inevitable. Faults occur during the software development process for various reasons, such as a misunderstood functionality, a typing error, or a programming mistake. In practice, debugging is a manual task. Developers often use symbolic debuggers and print statements to search for faults [1].

Several techniques to automate fault localization have been proposed [2], aiming at reducing the effort and time spent for improving the debugging process. Spectrum-based fault localization (SFL) techniques use program elements executed by test cases for debugging purposes [3, 4]. These elements are statements, basic blocks (or simply “blocks”), predicates, and method\(^1\) calls. Ranking metrics, also known as heuristics or similarity coefficients [5], are used to assign suspiciousness scores to each element. These techniques generate a list of elements sorted in descending order of suspiciousness. The more suspicious a statement is, the more likely it is to contain a fault.

A developer using an SFL technique may start by investigating the top ranked statements. If s/he does not find the bug after investigating the top one, the next statement may be verified. This process is repeated until locating the fault site, or until the developer abandons the SFL technique. Current SFL techniques lack contextual information. Suppose a developer is looking for faults

\(^1\)We use the term method to refer to a routine (e.g., procedure, function, method) in a program.
in a large program, in which several code excerpts have the same suspiciousness score. If s/he had to choose one to initiate the investigation, which piece of code should s/he investigate first? If the bug is not found in the first pick, which ones should be investigated next? Moreover, a typical ranking list, sorted only by suspiciousness, may comprise scattered statements—a statement at the top of the list followed by a statement that belongs to a completely different site in the source code, requiring a large cognitive effort from the developer to jump from one context to the other.

Parnin and Orso [6] carried out a study with developers using the Tarantula technique [4]. Their results suggest that developers tend not to follow the order proposed by the lists when the bug is not present among the first picks. This indicates that developers have an effort budget for using fault localization techniques. If the developer is unable to find the bug within a fixed amount of investigated statements (i.e., within the effort budget), s/he tends to use alternative ways to locate the bug. Therefore, contextual information that helps to position the fault site within the effort budget can leverage the developer’s performance. Furthermore, the participants of their study indicated that more information about the source code would enhance a fault localization technique.

Thus, relating blocks to their methods can be a useful way to improve the fault localization process. In this regard, Gouveia et al. [7] proposed a technique to group statements by their code structures (methods, classes, packages, and so on). These structures were displayed in a visualization tool called GZoltar [8], bringing contextualized information to the debugging task; a user study showed that developers benefited from this information.

This paper investigates the use of roadmaps combined with filtering strategies for adding contextual information to support fault localization. A roadmap is a list of suspicious methods that should be inspected according to the order established. At each method, a list of suspicious blocks is used to locate the fault. Thus, a developer using a roadmap will inspect blocks that are grouped by their methods. We use two techniques, proposed in preliminary studies, to create roadmaps: Code Hierarchy (CH) [9] and Integration Coverage-based
Debugging (ICD) [10]. CH assigns as the suspiciousness score of a method the highest suspiciousness score of its internal blocks. Thus, CH uses block-hit spectra\(^2\) to create the roadmap. To deal with ties, CH also assigns to methods the number of blocks with the highest score. ICD, in turn, uses spectra from communication between methods (caller-callee relationships) during a test execution to establish a method’s suspiciousness score. Both techniques use block-hit spectra to provide information about the most suspicious blocks for each method.

We propose two new filtering strategies for narrowing down the amount of blocks to be inspected at each method: Level Score (LS) and Fixed Budget (FB). LS and FB are combined with CH and ICD to provide more context for fault localization. LS deems each different suspiciousness score of blocks inside a method as a level. Only blocks within the level score are inspected. FB is the number of blocks at most that a developer is willing to inspect until finding a bug. It limits the number of blocks to be inspected using the block list. The block lists of the visited methods are investigated until the fixed budget is exhausted. In this sense, our techniques support the contextualization of spectrum-based fault localization at two levels. CH and ICD provide contextual information at a coarse level through the roadmaps, while LS and FB reduce the amount of blocks investigated inside each method at a fine-grained level.

We carried out an experiment to evaluate the effectiveness of the contextualization techniques. They were compared to a ranking list composed only of blocks—block list (BL). We used sixty-two faulty versions of seven real-world open source programs in the evaluation. The techniques were assessed by the absolute number of blocks inspected before reaching the bug, instead of the percentage of code inspected. Indeed, studies suggest that absolute values of effort lead to more accurate assessment of fault localization techniques [6, 11].

In the evaluation, we used the effort budget concept, varying it from 5 to 100 blocks. The rationale is that if the developer is unable to find the bug by

\(^2\)Code coverage collected at block-level.
investigating the number of blocks established in a particular effort budget (e.g.,
20 blocks), then the technique offers little help in locating the fault. Thus, the
effectiveness was measured by the number of bugs located within each effort
budget.

The results indicate that the contextualization techniques improve the fault
localization effectiveness with statistical significance for a confidence level of
95%—ICD using FB (ICD-FB) and CH using FB (CH-FB), found more faults
than BL for all budgets; ICD-FB and CH-FB inspect less blocks to locate
the faults. ICD-FB, CH-FB, and ICD-LS are also more effective than BL for
multiple-fault programs, especially with small budgets.

Considering budgets from 5 to 50 blocks, the contextualization techniques
performed particularly well in comparison to BL, which needed to investigate up
to 50% more blocks to hit the same amount of bugs. This significant difference
in the number of blocks inspected may be the tipping point between the success
or failure of an SFL technique.

Moreover, only four methods need to be inspected using ICD-FB and CH-FB
to locate 70% of all faults. Thus, our results suggest that the contextualization
provided by roadmaps and filtering strategies is useful for guiding the developer
toward faults, and improves the performance of spectrum-based fault localiza-
tion techniques.

The main contributions of this paper are:

• New approaches to contextualize and improve fault localization, which
  combine two previous roadmap-based techniques with new filtering strate-
gies;

• An experimental evaluation in which the contextualization techniques were
  used to debug faulty versions of programs of various sizes, with real and
  seeded bugs, and containing single and multiple bugs; and

• Assessment of the effectiveness of fault localization techniques using 15
different effort budgets (from 5 to 100 blocks) and the number of methods
  inspected.
We implemented CH, ICD, LS, and FB in the Road2Fault tool, which is an open source program under the Apache v2 license. Road2Fault, the data, and the scripts used in the experiments are available at github.com/saeg/road2fault.

In the remainder of the paper: SFL techniques are described in Section 2; CH, ICD, LS, and FB are presented in Section 3; Section 4 describes the experimental design; the results and their discussion are presented in Section 5; threats to validity are discussed in Section 6; and related work is presented in Section 7. We draw our conclusions and discuss future directions in Section 8.

2. Spectrum-based fault localization

Program spectra or coverage\(^3\) [12, 3] consists of program elements (e.g., statements, blocks, predicates, methods) covered during a test case execution [13, 14]. Most spectrum-based fault localization techniques use ranking metrics to assign suspiciousness scores to these elements [4, 3]. The outcome is a list of elements ranked in descending order of suspiciousness.

In this context, the metrics used to select suspicious program elements are a pivotal issue. Several metrics have been proposed or used to indicate elements most likely to contain faults [4, 3, 15]. The most effective metrics are characterized by identifying suspicious code excerpts from the frequency of elements exercised in passing and failing test cases. There are metrics created specifically for fault localization, such as Tarantula [4], whereas other metrics, e.g., Ochiai [3], have been adapted from areas such as Molecular Biology.

Tarantula was one of the first metrics proposed for fault localization. Its formula (\(M_T\)) is shown in Equation 1:

\[
c_{ef} \text{ indicates the number of times a program component } (c) \text{ is executed } (e) \text{ in failing } (f) \text{ test cases},
\]

\[
c_{nf} \text{ is the number of times a component is not } (n) \text{ executed by failing } (f) \text{ test cases},
\]

\[
c_{ep} \text{ is the number of times a component is executed } (e) \text{ by passing } (p) \text{ test cases, and } c_{np}
\]

\[^3\text{Henceforth, we use indistinctly the terms spectra and coverage.}\]
represents the number of times a component is not \((n)\) executed by passing \((p)\) test cases. Thus, for each component, Tarantula calculates the frequency that a component is executed in failing test cases, and divides it by the frequency that this component is executed in failing and passing test cases.

Used in several works, the ranking metric Ochiai has presented the best results for fault localization \([3, 16, 17, 18]\). Equation 2 shows its formula. Ochiai does not take into account statements not executed in passing runs \((c_{np})\). The square root also reduces the weight of \(c_{nf}\) and \(c_{ep}\) in the suspiciousness score.

\[
\begin{align*}
M_T &= \frac{c_{ef}}{c_{ef} + c_{nf}} + \frac{c_{ep}}{c_{ep} + c_{np}} \\
M_O &= \frac{c_{ef}}{\sqrt{(c_{ef} + c_{nf})(c_{ef} + c_{ep})}}
\end{align*}
\]

A limitation of the current techniques is the lack of guidance to search for faults. As they assign suspiciousness scores just for statements (blocks), the top ranked ones may come from scattered packages, classes, and methods. This fact may lead to the inspection of large excerpts of code, especially in large-sized programs.

One way to tackle this problem is by listing the most suspicious methods and by bringing together statements that belong to the same method. The list of most suspicious methods can guide the developer’s search in large programs. Additionally, a method is a code unit that contains the logic of a specific functionality; thus, it can help to contextualize a fault inside it. The larger the program is, the more important method-level information becomes. The intuition is that first one understands the logic of a method, and then investigates its blocks. Kochhar et al. \([19]\) asked 386 software engineering practitioners about their preferred granularity levels in a fault localization tool. Most of them preferred methods, slightly more than statements and blocks.
3. Contextualizing fault localization

In this section, we present techniques to support contextual guidance for fault localization. The contextualization strategy proposed is two-fold. It comprises the selection of most suspicious methods and also the selection of blocks that should be inspected at each method. Thus, the techniques aim to reduce the amount of code to be inspected using method-level information. Moreover, grouping blocks by their methods can add more information to understand faults.

*Code Hierarchy* (CH) and *Integration Coverage-based Debugging* (ICD) provide method-level information through a roadmap for fault localization. A developer searching for a bug investigates the most suspicious methods and, for each method, investigates its most suspicious blocks.

We then introduce two filtering strategies to narrow down the amount of blocks to be inspected. The strategies are called *Level Score* (LS) and *Fixed Budget* (FB). Both strategies aim to reduce the effort needed to search for bugs inside of the methods. We show an example in which CH and ICD roadmaps are used to locate faults along with LS and FB filtering strategies.

3.1. Code hierarchy

*Code Hierarchy* (CH) [9] is a technique based on the structure of object-oriented programs. It uses the code hierarchical structure (packages, classes, and methods) to organize the debugging information.

CH assigns suspiciousness scores to packages, classes, and methods based on the suspiciousness scores of their internal blocks. The CH suspiciousness score is given by a pair *(susp, number)* where *susp* is the highest suspiciousness score assigned to a block belonging to the program entity (package, class, or method) and *number* is the number of blocks that have such a score. Only block-hit spectra is required to compute CH.

In the case of a tie, the number of blocks with the same *susp* score (*number*) determines which entity will be investigated next. Ties in ranking lists are
common occurrences that impair the performance of SFL techniques [20, 21]. Xie et al. [21] present four tie-breaking strategies for ranking lists, while Wong et al. [20] assessed ties as an impact factor regarding the performance of fault localization techniques.

CH generates a roadmap—a list of suspicious methods—to be visited during fault localization derived from CH. A CH’s roadmap suggests an order for fault examination by establishing which methods should be inspected first. The rationale is that a method is a self-contained contextualized fault localization unit—it is the lowest code level structure encapsulating blocks associated with the logic of a program’s function. A developer starts by inspecting the most suspicious method of the roadmap, and then investigates the method’s most suspicious blocks. If the developer does not locate the fault in the first method, s/he investigates the second one. This process continues until locating the bug, or giving up on using the roadmap.

The motivation for using the CH roadmap and block coverage lies in the assumption that the suspiciousness of a faulty block and its method are correlated. If a block is executed often by failing test cases, it will likely receive a high score, which leads to a well-ranked method. As a result, the faulty method tends to be well ranked in the roadmap because it contains the faulty block that causes a test to fail.

Algorithm 1 shows how the CH roadmap is determined. The input consists of a list of suspicious blocks and a list of methods. The suspiciousness scores are assigned to blocks according to a ranking metric $M$, such as Ochiai or Tarantula. To create the CH roadmap, each method is assigned with the highest suspiciousness score of its blocks ($\text{susp}$) and the number of blocks with that score ($\text{number}$). Next, the methods are sorted by their $\text{susp}$ score in descending order. CH establishes that the $\text{susp}$ score is the most important factor for fault localization. For tie cases, we assumed that the smaller the $\text{number}$, the better. Thus, if a method has only one block with the highest suspiciousness score, it should be investigated first. The idea behind first inspecting methods with the fewest most suspicious blocks is that these methods contain blocks that are
better discriminated by ranking metrics, which in turn may lead to the faulty block. Moreover, it may be less tiresome for developers to inspect less blocks first if the bug is not in the method. In case of a double tie, the next entity is randomly decided.

**Algorithm 1:** CH roadmap creation

| Input: | blockList – a block list with suspiciousness scores assigned by a metric M; methodList – a list of methods. |

```plaintext
foreach method in methodList do
    method.susp ← 0;
    method.number ← 0;
endforeach
foreach block in blockList do
    if block.method = method then
        if block.susp > method.susp then
            method.susp ← block.susp;
            method.number ← 1;
        end
    else if block.susp = method.susp then
        method.number+++;
    end
end
chRoadmap ← methodList sorted in decreasing order by method.susp and then in increasing order by method.number;
return chRoadmap;
```

Table 1 presents the CH roadmap generated for a faulty version of the Ant program. Details on the characteristics of Ant are presented in Section 4.

3.2. Integration coverage-based debugging

Integration Coverage-based Debugging (ICD) [10] is another technique for tackling the lack of context in SFL techniques. ICD also provides a roadmap to investigate the most suspicious methods.

To create the roadmap, ICD uses an integration coverage called MethodCall-Pair (MCP) [9]. The ICD roadmap is created from a ranking of pairs of method calls—caller-callee relationships—performed during a test execution. At each
method, block-hit spectra is also used to locate bugs. Thus, ICD combines MCP spectra to create the roadmap with block-hit spectra to fine tune the inspection. In the following subsections, we present the MCP spectra and how the ICD roadmap is built.

3.2.1. MethodCallPair

The MethodCallPair (MCP) integration coverage was developed to collect information with respect to the most suspicious method call sequences. MCP was inspired by integration testing techniques [22], but it was not devised as a testing requirement. Rather, it was conceived as integration information captured during a test suite execution for debugging purposes.

MCP represents relationships between methods executed during test runs. A pair is composed of a caller method and a callee method. Instead of simply capturing the method call stack trace to calculate the suspiciousness of methods, the idea is to highlight methods more often related to other methods in failing executions. Thus, the integration between methods is used to indicate those more likely to contain faults. Figure 1 illustrates MCP coverage information, in which the method caller of class A invokes the method callee of class B.

```
Class A
void caller(int i){
    int a;
    b = new B();
    if(i>=0)
        a = b.callee(i);
}

Class B
int callee(int x){
    int y = 0;
    if(x > 10)
        y = 5;
    return y;
}
```

Figure 1: MethodCallPair (MCP)

3.2.2. Creating the ICD roadmap

The ICD roadmap is a simplified guide for inspecting a list of suspicious method call pairs (MCPs) to be used when searching for faults. ICD creates
the roadmap from the MCP spectra according to the following steps.

1. MCPs are tracked during the execution of each test case.
2. ICD uses the MCP spectra to assign suspiciousness scores to each MCP. Any ranking metric $M$ can be used for this purpose. In the case of Ochiai, MCPs are the components used in Equation 2.
3. ICD generates a list of MCPs sorted by suspiciousness. MCPs with the same suspiciousness scores are sorted according to their order of occurrence in the execution.
4. ICD creates the roadmap by visiting the MCP list from the top ranked elements to the least ranked ones, and from the caller method to the callee method. If a method does not yet appear in the roadmap, it is added to the roadmap with the same suspiciousness score assigned to the visited pair. Algorithm 2 shows how the ICD roadmap is created.

| Input: mcpSortedList – an MCP list sorted in decreasing order of suspiciousness assigned by a metric $M$. |

```
icdRoadmap ← ∅;
while not mcpSortedList.empty() do
  mcp ← mcpSortedList.removeFirstElement();
  if not mcp.caller in icdRoadmap then
    method ← mcp.caller;
    method.susp ← mcp.susp;
    icdRoadmap.addLast(method);
  end
  if not mcp.callee in icdRoadmap then
    method ← mcp.callee;
    method.susp ← mcp.susp;
    icdRoadmap.addLast(method);
  end
end
return icdRoadmap;
```

**Algorithm 2: ICD roadmap creation**

Consider that an MCP ($A.caller, B.callee$) is executed in 4 out of 10 failing test cases, and in only 2 out of 90 passing test cases. Thus, the values for $c_{e,f}$,
$c_{nf}$, $c_{cp}$, and $c_{np}$ of Equation 2 will be, respectively, 4, 6, 2, and 88. As a result, the Ochiai suspiciousness score ($M_O$) will be 0.51 for the MCP ($A$.caller, $B$.callee). Once all MCPs have been assigned their $M_O$, they are sorted in decreasing order; the list of sorted MCPs is then used to create the roadmap. During roadmap creation, if $A$.caller is yet not in the roadmap, it is inserted with a suspiciousness score of 0.51. Next, $B$.callee is checked. If $B$.callee is not yet in the roadmap, it is also inserted with a suspiciousness score of 0.51.

Therefore, the roadmap is composed of each method in the MCP list with its highest suspiciousness score. Methods that appear in more than one pair in the MCP list are included only once in the roadmap. Thus, using the ICD roadmap avoids repeatedly checking methods.

Table 2 shows the ICD roadmap generated for fault PJH_AK_1 of the Ant program.

### 3.3. Filtering strategies to reduce code inspection

CH and ICD provide a roadmap to search for bugs in the most suspicious methods. For each method, a developer should also inspect its most suspicious blocks. We propose two strategies that aim at limiting the number of blocks inspected when a particular method of the roadmap is investigated. Therefore, both strategies are applied at block-level. These filtering strategies are called Level Score (LS) and Fixed Budget (FB).

The main idea of LS and FB is to avoid the inspection of blocks that are less likely to contain faults, thus reducing the effort needed to use the roadmap. Moreover, the filtering strategies were devised to mimic developers’ behavior while inspecting for bugs since they may not check all hints. Top hints are more likely to be verified. Both strategies vary according to the scores of the blocks in each method. Some methods may contain a large amount of LOCs being tedious and imprecise to decide how much code inside a method to inspect before moving to the following method. Even methods with reduced size may only contain a few blocks with higher suspiciousness scores, and so avoiding the other lower ranked blocks may improve the code inspection.
These filtering strategies act as criteria for the inspection of blocks inside the methods. However, we do not propose a single threshold value to apply for all methods as, for example, a single suspiciousness score value. Therefore, both filtering strategies can be applied with different ranking metrics no matter how they assign suspiciousness scores to the code elements. For instance, Ochiai generally assigns suspiciousness scores to code elements with large distance among them while Tarantula assigns values with small differences.

To propose these filtering strategies, we also regarded that developers are generally impatient and may abandon a technique if the bug is not found soon. Thus, strategies should effectively narrow down the amount of code to inspect. In doing so, they reduce the effort needed to find bugs and keep the developers’ interest.

3.3.1. Level score

Level Score (LS) is a strategy that deems each unique suspiciousness score of blocks inside a method as a level. A method generally contains blocks with different suspiciousness scores. One or more blocks may have the same score.

Level score was based on the concept breadth-first search. Vessey [23] suggests that experienced developers use a breadth-first approach while debugging. Thus, a fault inspection can be improved using a more breadth inspection of methods and blocks through the roadmap instead of an in-depth search through blocks of each roadmap’s method.

Consider LS(\(l\)) in which \(l\) is the \(l^{th}\) distinct suspiciousness score level of the blocks inside a method in descending order. All blocks with a score equal to or higher than the \(l\) score should be investigated. As \(l\) increases, more levels with lower scores are included and, thus, more blocks must be inspected.

For \(l = 1\) (LS(1)), only blocks with the highest score are inspected. If the bug is not found among them, the following method should be inspected. For LS(2), all blocks with the highest suspiciousness score, and all blocks with the second highest suspiciousness score will be examined. Blocks with lower scores will be ignored, and, if the fault is not found, the following method should be
investigated. The same reasoning is valid for other $l$ values.

3.3.2. Fixed budget

The Fixed Budget (FB) strategy aims at determining a maximum number of blocks ($b$) to inspect, regardless of how many methods will be inspected. It is based on the idea of representing an effort value that a developer would be willing to spend to find a bug.

Fixed budget assumes that an absolute number of code elements should be inspected to search for a bug without decreasing the developers’ interest [6]. This effort budget may vary a lot between developers, and also may vary according to the severity of the fault—a severe bug tends to increase such a willingness.

To perform this strategy, we first use a general block list obtained from block-hit spectra (i.e., before grouping blocks by their methods) in descending order of suspiciousness. Consider $FB(b)$, where $b$ is the maximum number of blocks to be inspected. To determine the blocks of $FB(b)$, we verify the suspiciousness score of the $b^{th}$ block in the block list—the budget score.

If the number of blocks with score equal to or greater than the budget score exceeds $b$, the score immediately above is used. The idea behind it is to avoid an over-budget. The only exception is when the budget score of the $b^{th}$ block is the highest score of the list. In this case, if the number of blocks exceeds $b$, we prefer to select an over-budget number of blocks to be inspected instead of an empty list.

After assigning the budget score, blocks are grouped by their respective methods. Following the roadmap, all blocks with scores equal to or greater than the budget score should be inspected.

FB is directly affected by ties in the block list—if the score of the $b^{th}$ block is present in other blocks, the score above will be chosen, leading to less than $b$ blocks to be inspected, except for the highest score. Small $b$s lead to few methods to be inspected, since only methods that contains blocks within the budget score are inspected.

Suppose $a b = 5$ ($FB(5)$), the first two most suspicious blocks have scores
equal to 1.0, the following three blocks have scores equal to 0.99, and the sixth block has a score of 0.98. The suspiciousness score of the fifth block will be the budget score, which is 0.99. This score is used to inspect the blocks inside the methods indicated by the roadmap.

### 3.4. Debugging with roadmaps

We illustrate these strategies below with an example from a program used in our experiments. Consider a developer using a roadmap to search for the bug PJH_AK_1 of the Apache Ant program (described in Section 4). Figure 2 shows the code excerpt containing the bug. It was seeded for experimental purposes [24]. In the figure, the C preprocessor command `#ifdef PJH_AK_1` was used to control the inclusion of the buggy line 112, instead of the correct line 114.

```java
private void parse() throws BuildException {
    try {
        SAXParser saxParser = getParserFactory().newSAXParser();
        parser = saxParser.getParser();
        String uri = "file:" + buildFile.getAbsolutePath().replace("\", "/");
        for(int index = uri.indexOf("#") ; index != -1; index = uri.indexOf("#")){
            if (index + 1 < uri.length()) {
                uri = uri.substring(0, index) + "%24" + uri.substring(index + 1);
            } else {
                uri = uri.substring(0, index) + "%23" + uri.substring(index + 1);
            }
        }
        inputStream = new FileInputStream(buildFile);
        inputSource = new InputSource(inputStream);
        inputSource.setSystemId(uri);
    }
}
```

Figure 2: Code excerpt with the Ant’s PJH_AK_1 bug in line 112.

Table 1 shows the CH roadmap using Ochiai. The method `parse()`, which contains the faulty block, is the second method of the roadmap; its suspiciousness score is 1.0. Table 2 shows the ICD roadmap using Ochiai for the same fault. For ICD, the faulty method was classified in the first position. In both tables, the faulty method is highlighted in gray. The roadmaps were created following the guidelines described in Sections 3.1 and 3.2.

Table 3 shows `parse()`’s block list for bug PJH_AK_1. The Pos. column has the block position in the general block list. The Id column has the block id. The following columns mean, respectively, the first line of the block, the source
The first three blocks of `parse()` (the first three lines of Table 3) have the highest suspiciousness scores, but the fault site is not located in them. Thus, a strategy which allows us to investigate a few more blocks without excessively increasing the amount of blocks examined in each method improves the ability of CH and ICD to search for bugs at block-level.

For LS equal to one (LS(1)), and using either a CH or an ICD roadmap, only the most suspicious blocks of each method will be inspected. In this case, the bug will be missed because the three blocks of `parse()` with highest suspiciousness scores do not include the bug. If LS is two (LS(2)), only one extra block will be inspected in `parse()`, which is the faulty block.

Since `resolveEntity()` is the most suspicious method of the CH roadmap, nine more blocks from this method will be investigated before the four blocks of `parse()` using LS(2), resulting in a total of thirteen blocks investigated until reaching the faulty one. Using the ICD roadmap, only the four blocks of `parse()` will be examined. In Table 3, the lines highlighted in gray are the blocks belonging to the inspection range for LS(2).

Suppose a developer opted to use FB, and selects an FB equal to 10 (FB(10)). The block list, including blocks of all methods for PJH\_AK\_1, is shown in Table 4. The gray lines are blocks from the `parse()` method. The score of the tenth block is 0.5. However, there are more than ten blocks with this score, which exceeds the budget. Thus, the budget score will be 0.57, i.e., the score immediately above. Only blocks with suspiciousness scores equal to or greater than 0.57 will be examined.

Using the CH roadmap, two blocks of the method `resolveEntity()` are inspected (in Table 4, these blocks are in positions four and five). After that, four blocks of `parse()` will be inspected; a total of six blocks are inspected until reaching the bug. Using the ICD roadmap, only four blocks of `parse()` will be inspected before the bug is reached.

Using both roadmaps and the filtering strategies for PJH\_AK\_1, in the worst case the developer will inspect thirteen blocks before reaching the fault site. In
the best case, only four blocks will be inspected. As a result, the bug is in the set of suspicious blocks of `parse()` without excessively increasing the amount of blocks to be examined.

Table 1: CH roadmap of the PJH AK-1 bug

<table>
<thead>
<tr>
<th>Method</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>resolveEntity()</td>
<td>1.00</td>
</tr>
<tr>
<td>parse()</td>
<td>1.00</td>
</tr>
<tr>
<td>BuildException()</td>
<td>0.37</td>
</tr>
<tr>
<td>getMessage()</td>
<td>0.14</td>
</tr>
<tr>
<td>getPriority()</td>
<td>0.09</td>
</tr>
<tr>
<td>setName()</td>
<td>0.09</td>
</tr>
<tr>
<td>setDefaultTarget()</td>
<td>0.09</td>
</tr>
<tr>
<td>setUserProperty()</td>
<td>0.09</td>
</tr>
<tr>
<td>addBuildListener()</td>
<td>0.09</td>
</tr>
<tr>
<td>fireMessageLoggedEvent()</td>
<td>0.09</td>
</tr>
<tr>
<td>access$202()</td>
<td>0.09</td>
</tr>
<tr>
<td>access$400()</td>
<td>0.09</td>
</tr>
<tr>
<td>access$300()</td>
<td>0.09</td>
</tr>
<tr>
<td>access$100()</td>
<td>0.09</td>
</tr>
<tr>
<td>configureProject()</td>
<td>0.09</td>
</tr>
<tr>
<td>setDocumentLocator()</td>
<td>0.09</td>
</tr>
</tbody>
</table>

Table 2: ICD roadmap of the PJH AK-1 bug

<table>
<thead>
<tr>
<th>Method</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>parse()</td>
<td>1.00</td>
</tr>
<tr>
<td>BuildException()</td>
<td>1.00</td>
</tr>
<tr>
<td>resolveEntity()</td>
<td>0.44</td>
</tr>
<tr>
<td>log()</td>
<td>0.44</td>
</tr>
<tr>
<td>access$300()</td>
<td>0.44</td>
</tr>
<tr>
<td>access$400()</td>
<td>0.44</td>
</tr>
<tr>
<td>set()</td>
<td>0.37</td>
</tr>
<tr>
<td>setMessage()</td>
<td>0.37</td>
</tr>
<tr>
<td>createTask()</td>
<td>0.35</td>
</tr>
<tr>
<td>Echo()</td>
<td>0.35</td>
</tr>
<tr>
<td>execute()</td>
<td>0.35</td>
</tr>
<tr>
<td>createAttributeSetter()</td>
<td>0.33</td>
</tr>
<tr>
<td>IntrospectionHelper$15()</td>
<td>0.33</td>
</tr>
<tr>
<td>fireMessageLoggedEvent()</td>
<td>0.14</td>
</tr>
<tr>
<td>getMessage()</td>
<td>0.14</td>
</tr>
<tr>
<td>executeTarget()</td>
<td>0.13</td>
</tr>
</tbody>
</table>

3.5. CH and ICD complexity

To understand the execution costs of CH and ICD, we present issues regarding their time and space complexities. As for the time complexity, CH is built based on block-hit spectra. Each block is associated with its method during the instrumentation. Therefore, block and method spectra are gathered together, resulting in a complexity of $O(N)$, where $N$ is the number of blocks executed by the test suite. Regarding space complexity, CH stores the coverage of each block, i.e., the block-hit spectra. Being $B$ the total of blocks, the space needed to store block-hit spectra costs $O(B)$. It also requires an additional space to store the method information, which is proportional to the number of methods ($M$) of a program. This extra space for CH is quite small in comparison with the space needed to store block-hit spectra, which entails a space complexity of $O(B + M) = O(B)$. 
Table 3: Block list of the PJH AK 1 bug for parse()

<table>
<thead>
<tr>
<th>Pos.</th>
<th>Id</th>
<th>Line</th>
<th>Code</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>337</td>
<td>149</td>
<td>catch(FileNotFoundException exc) {</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>359</td>
<td>156</td>
<td>if (inputStream != null){</td>
<td>1.00</td>
</tr>
<tr>
<td>3</td>
<td>364</td>
<td>156</td>
<td>try{</td>
<td>1.00</td>
</tr>
<tr>
<td>6</td>
<td>68</td>
<td>112</td>
<td>uri = uri.substring(0, index) + &quot;%24&quot; ...; (\Leftarrow\text{bug})</td>
<td>0.57</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
<td>102</td>
<td>FileInputStream inputStream = null;</td>
<td>0.09</td>
</tr>
<tr>
<td>27</td>
<td>120</td>
<td>118</td>
<td>inputStream = new FileInputStream(buildFile);</td>
<td>0.09</td>
</tr>
<tr>
<td>28</td>
<td>367</td>
<td>156</td>
<td>if(inputStream != null){</td>
<td>0.09</td>
</tr>
<tr>
<td>29</td>
<td>373</td>
<td>158</td>
<td>inputStream.close();</td>
<td>0.09</td>
</tr>
<tr>
<td>30</td>
<td>382</td>
<td>162</td>
<td>}</td>
<td>0.09</td>
</tr>
<tr>
<td>31</td>
<td>4</td>
<td>106</td>
<td>SAXParser saxParser = getParserFactory()...</td>
<td>0.09</td>
</tr>
<tr>
<td>32</td>
<td>62</td>
<td>110</td>
<td>for(int index = uri.indexOf('#'); index != -1; ...{</td>
<td>0.09</td>
</tr>
<tr>
<td>490</td>
<td>201</td>
<td>123</td>
<td>}</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Table 4: Block list of the PJH AK 1 bug for all methods

<table>
<thead>
<tr>
<th>Pos.</th>
<th>Id</th>
<th>Line</th>
<th>Code</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>337</td>
<td>149</td>
<td>catch(FileNotFoundException exc) {</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>359</td>
<td>156</td>
<td>if (inputStream != null){</td>
<td>1.00</td>
</tr>
<tr>
<td>3</td>
<td>364</td>
<td>156</td>
<td>try{</td>
<td>1.00</td>
</tr>
<tr>
<td>4</td>
<td>248</td>
<td>259</td>
<td>} catch (FileNotFoundException fne) {</td>
<td>1.00</td>
</tr>
<tr>
<td>5</td>
<td>284</td>
<td>265</td>
<td>return null;</td>
<td>1.00</td>
</tr>
<tr>
<td>6</td>
<td>68</td>
<td>112</td>
<td>uri = uri.substring(0, index) + &quot;%24&quot; ...; (\Leftarrow\text{bug})</td>
<td>0.57</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>226</td>
<td>project.log(&quot;resolving systemId: &quot; + systemId,...);</td>
<td>0.50</td>
</tr>
<tr>
<td>8</td>
<td>102</td>
<td>239</td>
<td>String entitySystemId = path;</td>
<td>0.50</td>
</tr>
<tr>
<td>9</td>
<td>113</td>
<td>245</td>
<td>while (index != -1) {</td>
<td>0.50</td>
</tr>
<tr>
<td>10</td>
<td>167</td>
<td>250</td>
<td>File file = new File(path);</td>
<td>0.50</td>
</tr>
<tr>
<td>11</td>
<td>202</td>
<td>256</td>
<td>InputSource inputSource = new InputSource(...);</td>
<td>0.50</td>
</tr>
<tr>
<td>12</td>
<td>39</td>
<td>229</td>
<td>String path = systemId.substring(5);</td>
<td>0.50</td>
</tr>
<tr>
<td>13</td>
<td>53</td>
<td>234</td>
<td>while (index != -1) {</td>
<td>0.50</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Table 5: Average of memory size overhead of CH and ICD compared with BL

<table>
<thead>
<tr>
<th>Program</th>
<th>CH/BL</th>
<th>ICD/BL</th>
<th>ICD/CH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant</td>
<td>0.57%</td>
<td>36.76%</td>
<td>35.98%</td>
</tr>
<tr>
<td>Commons-Math</td>
<td>0.56%</td>
<td>67.41%</td>
<td>66.47%</td>
</tr>
<tr>
<td>HSQLDB</td>
<td>0.15%</td>
<td>68.50%</td>
<td>68.24%</td>
</tr>
<tr>
<td>JTopas</td>
<td>2.53%</td>
<td>44.58%</td>
<td>41.01%</td>
</tr>
<tr>
<td>PMD</td>
<td>0.63%</td>
<td>58.11%</td>
<td>57.12%</td>
</tr>
<tr>
<td>XML-security</td>
<td>0.90%</td>
<td>35.20%</td>
<td>33.99%</td>
</tr>
<tr>
<td>XStream</td>
<td>0.16%</td>
<td>88.81%</td>
<td>88.51%</td>
</tr>
<tr>
<td>Average</td>
<td>0.78%</td>
<td>57.05%</td>
<td>55.90%</td>
</tr>
</tbody>
</table>

ICD is based on MCP and block-hit spectra. To collect MCP spectra, we need to instrument the entry block and all possible exit blocks of each method. These blocks are also covered by the block-level instrumentation. Thereby, both MCP and block-hit spectra are gathered together. Regarding the time complexity, ICD also depends on the number of executed blocks $O(N)$. For the space complexity, ICD stores MCP spectra, which is linear with the number of caller-callee pairs ($P$) of a program run. ICD also stores block-hit spectra ($B$). In the worst case scenario, all methods call and are called by all methods ($M^2$), i.e., $P \leq M^2$, which entails a total space complexity of $O(B + M^2)$.

The above analysis suggests that the extra space for CH is negligible, and that ICD is a more space demanding technique. Table 5 shows the average memory space overhead to calculate the CH and ICD roadmaps in comparison with a single block list (BL) for all the programs evaluated in our experiments (see Section 4). Memory space overhead is the ratio between the memory size between CH, ICD, and BL. Such data was measured using a PC with an Intel Core i5 processor, 4GB of RAM running Ubuntu 13.04. CH had an average space overhead of 0.78% compared with BL, which confirms the estimate that the CH overhead is insignificant. ICD had an average overhead of 57% compared with BL. Though a sizable memory increase, it is far from the worst case scenario.

Thus, for the large programs used in the evaluation, even considering that ICD has a larger overhead, both the time and the space required to compute the roadmaps is affordable for current commodity computers.
4. Experimental evaluation

We carried out an experiment to evaluate the CH and ICD roadmaps combined with the filtering strategies LS and FB. We compared the contextualization techniques with block list (BL). BL represents the existing SFL techniques, in which the suspiciousness list comprises more fine-grained level elements, such as blocks or statements. The blocks are ordered by their suspiciousness scores. The experiment evaluates the effectiveness of the techniques in locating faults, i.e., the amount of code to inspect to reach the faults.

Next, we present details of the experiment. We describe the research questions, and then we present the subject programs, treatments assessed, measurements, and data collection procedure.

4.1. Research questions

The experiment comparing our techniques and BL was guided by the following research questions:

1. Are roadmaps combined with filtering strategies more effective than BL?
2. Which is the best level (l) value to use in LS?
3. Which is the most helpful filtering strategy for fault localization?
4. How many methods are inspected to reach the bugs using roadmaps and filtering strategies?
5. Are roadmaps combined with filtering strategies more effective than BL for programs containing multiple faults?

4.2. Subject programs

We selected seven medium to large size open source Java programs from various application domains: Ant, Commons-Math, HSQLDB, JTopas, PMD, XML-security, and XStream. Ant is a tool for building applications, especially for Java projects. Commons-Math is a library composed of mathematical and statistical functions. HSQLDB is a relational database. JTopas is a library for parsing arbitrary text data. PMD is a source code analyzer available for several
programming languages. XML-security is a component library implementing XML signature and encryption standards. Finally, XStream is a library to (de)serialize objects to XML.

Ant, JTopas, and XML-security were obtained from the Software-artifact Infrastructure Repository (SIR) [24], which contains programs prepared for experimentation. The SIR program bugs were seeded. A faulty version of XStream [25] was created with the same bug seeded by Gouveia et al. [7]. Commons-Math [26], HSQLDB [27], and PMD [28] contain real bugs extracted from their repositories. We identified twenty, four, and two real faults in Commons-Math, HSQLDB, and PMD, respectively. All versions of the programs used in the experiment contain a single bug.

Table 6 presents the characteristics of the selected programs. The columns indicate, respectively: the name of the program; the size variation of the faulty versions in thousands of lines of code (KLOC); the number of versions; the number of faults; the fault type, in which R means real faults and S means seeded faults; and the variation of the test suite size used in the different versions.

<table>
<thead>
<tr>
<th>Program</th>
<th>KLOC</th>
<th>Versions</th>
<th>Faults</th>
<th>Real</th>
<th>Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant</td>
<td>25-80</td>
<td>8</td>
<td>18</td>
<td>S</td>
<td>141-851</td>
</tr>
<tr>
<td>Commons-Math</td>
<td>16-39</td>
<td>3</td>
<td>20</td>
<td>R</td>
<td>1,167-2,114</td>
</tr>
<tr>
<td>HSQLDB</td>
<td>155-158</td>
<td>2</td>
<td>4</td>
<td>R</td>
<td>1,146-1,299</td>
</tr>
<tr>
<td>JTopas</td>
<td>1.8-5</td>
<td>3</td>
<td>4</td>
<td>S</td>
<td>22-31</td>
</tr>
<tr>
<td>PMD</td>
<td>50</td>
<td>1</td>
<td>2</td>
<td>R</td>
<td>616</td>
</tr>
<tr>
<td>XML-security</td>
<td>16-18</td>
<td>3</td>
<td>13</td>
<td>S</td>
<td>84-94</td>
</tr>
<tr>
<td>XStream</td>
<td>17</td>
<td>1</td>
<td>1</td>
<td>S</td>
<td>1,457</td>
</tr>
</tbody>
</table>

4.3. Treatments

The treatments were devised to assess the effectiveness of the contextualization techniques in finding bugs. The effectiveness was measured by the number of faults found using the techniques, and by the number of blocks inspected until finding the faults. We used a block-level suspiciousness list (BL) as baseline for the comparison. Two metrics were used to compare the techniques: Ochiai
and Tarantula. Both CH and ICD use the filtering strategies LS and FB for inspection at block-level.

The experiment was also designed to verify the amount of methods investigated to locate a bug. Our intuition is that locating bugs by investigating a limited number of methods may incentivize the use of automated SFL techniques.

**CH and ICD roadmaps.** Each technique generates its own roadmap. However, the procedure to examine the roadmaps is the same: a developer using a roadmap will start by investigating the most suspicious method. Inside the method, s/he will investigate the most suspicious blocks using one of the filtering strategies, LS or FB.

**LS(2).** LS(2) measures the Level Score strategy, where the level used to search for the bug is two. Thus, the blocks are verified until the second highest score. In Section 5.2 we discuss the motivation for using LS(2).

**FB(b).** We assessed FB using b values from 5 to 50 in 5-block intervals, and 10-block intervals from 50 to 100. For b = 5—FB(5)—only 5 blocks are verified.

The treatment **CH-FB(b)** is the CH roadmap along with FB for a b value. **CH-LS(2)** is the CH roadmap along with LS for a level 2. The same *rationale* is applied to **ICD-LS(2)** and **ICD-FB(b)**.

**Block list.** In this treatment, referred to simply as BL, the developer starts the search by inspecting blocks with the maximum suspiciousness scores. In failing to discover the faulty block, s/he proceeds to the next score, and so forth until reaching the faulty block. The BL treatment is given by the number of blocks inspected in all scores, including the faulty block score.

The number of blocks with the same suspiciousness score varies for the different faults investigated. To calculate the number of blocks inspected, we count all blocks with a suspiciousness score greater than or equal to the faulty block. Thus, we evaluate the worst case scenario of the number of blocks inspected, in which all blocks with the same suspiciousness are verified.
4.4. Procedure

In this section, we present the procedures used to collect the data on the effectiveness of the CH-LS(2), CH-FB, ICD-LS(2), ICD-FB, and BL techniques.

4.4.1. Effort budget

We assessed the techniques by measuring their effectiveness to rank the faults within the effort budgets. We estimate the effort budget as a fixed amount of blocks to be inspected by a developer, independent of the program size. The intuition is that if s/he is unable to locate the bug in this fixed number of blocks, the developer will resort to another fault localization strategy [6]. Steimann et al. [11] also argued that the use of absolute measures reflects a more realistic effort for fault localization. In practice, a technique that narrows down a bug to 1% of the code—as for EXAM score [20]—for a particular program \( P \) still demands the inspection of 1K LOC if \( P \) has 100K LOC. That is too much code to inspect! The effort budget can also be considered as a measure of efficiency—the fault not only must be found, it also must be classified within a limited number of elements.

Suppose that a fault is located by inspecting 19 and 57 blocks using ICD-LS(2) and BL, respectively. In this case, the bug can be found within an effort budget of 20 inspected blocks using ICD-LS(2). However, using only BL, the same budget is not enough, since it requires the inspection of 57 blocks. Nonetheless, with an effort budget of 70 blocks, the fault can be found by both techniques. We collected data for fifteen different budgets, namely, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 60, 70, 80, 90, and 100 blocks.

4.4.2. Creation of roadmaps and lists

First, we identified the faults that caused observable failures during test execution. In total, sixty-two faults met this requirement and were used in the experiment, as shown in Table 6. MCP and block coverage were collected for each test case, as well as their outcomes. Coverage data were collected using the Instrumentation Strategies Simulator (InSS) [29], a framework that allows...
rapid implementation of program instrumentation strategies in Java. Then, the roadmaps and the lists of suspicious blocks were created by the Road2Fault tool using Ochiai and Tarantula ranking metrics.

Since each method in a JUnit test class is a test case, failing and passing test cases were determined by verifying the output of each method. Methods that raised an exception during the test suite execution were also considered as failing test cases; otherwise, they were considered as passing test cases. Some test cases fail even when executed in the original version (without bugs) of the programs. These test cases were excluded, since these failures are unrelated to the bugs being investigated.

For HSQLDB, we excluded one of the test classes, which contains 181 test cases. Due to InSS limitations, this class took too much time to execute when instrumented. However, no failures were revealed by these test cases when we executed the test suite without InSS. In PMD, two test cases were excluded because they represent faults not in the program code, but rather in the test data (XML files with faulty Java code). For XStream, we excluded 10 test classes that conflicted with libraries, a total of 57 test cases, which could not instantiate abstract classes, used mock libraries, or failed to deal with threads.

4.4.3. Multiple-fault versions

To evaluate the performance of our techniques in programs containing multiple faults, we combined faults from the single-fault versions of the subject programs. We generated 62 multiple-fault versions to keep the same number of single-fault versions. Table 7 shows the number of multiple-fault versions.

We created 4-fault versions whenever possible, for program versions with 4 or more faults. Otherwise, we created 2-fault versions. For versions with a large number of faults (e.g., Commons-Math has a version containing 12 faults), we randomly generated multiple-fault versions ensuring that each fault was used at least once. Moreover, all faults are placed in single lines to avoid different chances of finding each bug [30, 31].

To evaluate the fault localization effectiveness in the multiple-fault versions,
we deem the number of blocks inspected until reaching the first fault [32,33,34].
The rationale is that a developer tends to inspect a list from the most suspicious
elements. If s/he locates the well-ranked fault, s/he would fix it and rerun the
tests, which would change the suspiciousness scores of remaining faults. Thus,
s/he would repeat the process until there are no more failures. This scenario
simulates a developer that tries to locate one bug at time.

4.4.4. Effectiveness data collection

To collect the effectiveness data, Road2Fault has functions for analyzing
the roadmaps and block lists according to the treatments presented above. This
approach avoids the need for manual intervention, as it is less prone to evaluation
errors. We collected the number of blocks and methods inspected until reaching
each fault for the five treatments.

5. Results and discussion

In this section, we present the results of the experiments by evaluating the
effectiveness of CH-LS(2), CH-FB, ICD-LS(2), ICD-FB, and BL. The results
and discussion are guided by the research questions introduced in Section 4.1.

In the following figures, we present the data obtained in the experiments.
CH-LS(2) and ICD-LS(2) are represented, respectively, by black and red bars.
CH-FB and ICD-FB, are represented, respectively, by green and blue bars.
The baseline SFL technique, Block List (BL), is represented by light gray bars.

Table 7: Multiple-fault versions.

<table>
<thead>
<tr>
<th>Program</th>
<th>2-fault</th>
<th>4-fault</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant</td>
<td>16</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>Commons-Math</td>
<td>3</td>
<td>21</td>
<td>24</td>
</tr>
<tr>
<td>HSQLDB</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>JTopas</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>PMD</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>XML-security</td>
<td>1</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>Total</td>
<td>25</td>
<td>37</td>
<td>62</td>
</tr>
</tbody>
</table>
The bars were organized following the legends’ order. Hereafter, whenever we mention CH and ICD, we refer to the use of CH and ICD roadmaps, respectively.

5.1. Are roadmaps combined with filtering strategies more effective than BL?

Figures 3 and 4 show the aggregated effectiveness values considering all programs for Ochiai and Tarantula, respectively. The comparison between the roadmaps and BL was carried out to verify not only which technique reaches more faults, but also which technique finds them by inspecting less blocks.

For Ochiai (Figure 3), ICD using the FB filtering strategy found 21 faults for the tightest budget (ICD-FB(5)). This means that 33.87% of the faults are found by inspecting at most 5 blocks. BL found 14 faults, one more than CH using LS(2) (CH-LS(2)). ICD-FB hit more faults than the other techniques for all budgets, especially in lower ranges. CH-FB also reached more faults than CH-LS(2), ICD-LS(2) and BL. CH-LS(2) hit more faults than BL within the budget ranges between 10 and 50. ICD-LS(2) had the worst performance among the techniques.

Using Tarantula (Figure 4), ICD-FB hit more faults in the ranges 5 to 35, except for budget 20. CH-FB and CH-LS(2) were more closely to ICD-FB, with some draws. CH-FB overcame the other techniques from budget 60 to 100. All treatments obtained better results using Ochiai than using Tarantula for all budgets. This fact is not a surprise, since several studies have shown that Ochiai outperforms Tarantula [3, 16, 17, 18]. It is worthy to note, though, that CH and ICD improve both Ochiai and Tarantula results; that is, they are orthogonal to the current SFL techniques.

ICD-FB hit 46 faults by inspecting up to 30 blocks using Ochiai, which represents 74.19% of all faults. To find a similar number of faults with BL, we need to inspect up to 50 blocks, i.e., 66.67% more blocks than ICD-FB. Taking into account only the faults located by any technique at the highest budget, which are 52 faults, ICD-FB found 88.46% of the located faults by inspecting up to 30 blocks. CH-FB reached 72.58% of all faults and 86.54% of all located faults, whilst BL reached 66.13% and 78.85%, respectively, for
the same budget. Thus, the contextualization techniques perform particularly well for small budgets in comparison to BL. This difference may be the tipping point between success or failure of an SFL technique, especially if the developer allocates tighter budgets.

The programs used in our evaluation have different characteristics in regard-
ing to number of LOCs, test suite size, program domain, coupling, and so on. Some programs have a limited number of faults. However, the behavior of the treatments was similar among all programs.
Tables 8 and 9 show the medians of blocks inspected by technique until reaching the bugs, respectively for Ochiai and Tarantula, for all programs and also for each program separately. The bold values are the cases in which the faults were better ranked by the techniques. ICD-FB obtained the lowest medians using Ochiai and Tarantula for all programs. ICD-FB or CH-FB achieved the lowest medians for all programs, except for HSQLDB.

We conducted a statistical analysis to compare the performance of our techniques against BL. First, we compared number of blocks inspected by the techniques to reach the faults for each budget, which we call Statistical Test 1 ($ST_1$).

We used the Wilcoxon signed-rank test [35] with a confidence level of 95%. The null hypothesis is that the techniques inspect the same number of blocks. The alternative hypothesis is that the left-hand side technique (e.g., CH-LS(2) x BL)

---

4We use the median for the comparison due to the non-normal distribution of our data.
Table 10: Budgets with statistical significance for ST1

<table>
<thead>
<tr>
<th>Metric</th>
<th>CH-LS(2) x BL</th>
<th>ICD-LS(2) x BL</th>
<th>CH-FB x BL</th>
<th>ICD-FB x BL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ochiai</td>
<td>—</td>
<td>—</td>
<td>10–100</td>
<td>5–100</td>
</tr>
<tr>
<td>Tarantula</td>
<td>50, 80–100</td>
<td>—</td>
<td>10–100</td>
<td>10–100</td>
</tr>
</tbody>
</table>

find faults inspecting less blocks for a particular budget.

Table 10 shows the results of ST1. The cells contain those budgets for which the null hypothesis was rejected. Whenever the cell is empty, the null hypothesis could not be rejected.

ICD-FB and CH-FB had the null hypothesis rejected for budgets 10 to 100 using either Ochiai or Tarantula. Only ICD-FB had also statistical significance at budget 5 for Ochiai. We believe that the absence of statistical significance for the budget 5 occurs because this budget is very restrictive, with a large number of draws, reducing the sample size for statistical tests.

Using Ochiai, CH-LS(2) could not reject the null hypothesis for all budgets; though, the null hypothesis was rejected for some budgets (50, 80, 90, 100) using Tarantula. Thus, one cannot assume that CH-LS(2) inspects less blocks than BL significantly. ICD-LS(2) could not reject the null hypothesis for all budgets and the two ranking metrics. Therefore, a developer will tend to find bugs inspecting less code when s/he uses ICD-FB for all budgets and CH-FB for almost all budgets used to limit the inspection.

We also compared the number of faults found by the techniques for all budgets (ST2). We used the Kolmogorov-Smirnov test [36] for this comparison, which is suitable for cumulative distribution data. We considered a confidence level of 95%. The null hypothesis is that the compared techniques reach the same number of faults through the budgets. The alternative hypothesis is that our techniques reach more faults than BL.

Table 11 shows the statistical results (p-values) for this comparison. We reject the null hypothesis for CH-FB and ICD-FB using Ochiai, which means that these techniques found more faults than BL with statistical significance. There is no significant difference in the number of faults found through the
budgets for the other comparisons. Also, BL found more faults than ICD-LS(2) with statistical significance. Therefore, the results suggest that ICD-FB and CH-FB improve the fault localization with respect to BL using a top performing ranking metric.

Regarding the research question, CH-FB and ICD-FB found more faults inspecting less blocks than BL with statistical significance. CH-LS(2) also reached more faults than BL, as shown in Figures 3 and 4, but without statistical support. As a result, CH-FB and ICD-FB appear to be promising. Furthermore, CH is less computational costly than ICD with a similar cost to BL.

Restricted budgets tend to be more realistic than larger ones [6, 11]. ICD-FB, CH-FB, and CH-LS(2) located more bugs within restricted budgets, independent of the ranking metric (Ochiai or Tarantula) used. In practice, we do not know how much code a developer is willing to investigate when searching for bugs. Because each developer has his or her own effort budget, it is safer to assume smaller budgets than larger ones. A technique that works best for such cases will more likely be used. ICD-FB, CH-FB, and CH-LS(2) seem to be a better choice than BL for these situations.

Thus, the experimental data suggest that CH, ICD and the filtering strategies LS and, especially, FB improve the effectiveness of SFL techniques. Also, they can provide guidance to developers performing debugging tasks. However, user studies are needed to verify these results in practice.

5.2. Which is the best level (l) value to use in LS?

We evaluated LS using different levels to understand which l values can reach more bugs. We used the same programs assessed in Section 4. Figures 5 and 6 show the number of faults found for different effort budgets, respectively for ICD and CH using five LS levels, from 1 to 5, and Ochiai.

Table 11: P-values (%) for ST2

<table>
<thead>
<tr>
<th></th>
<th>CH-LS(2) x BL</th>
<th>ICD-LS(2) x BL</th>
<th>CH-FB x BL</th>
<th>ICD-FB x BL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ochiai</td>
<td>33.42</td>
<td>100.00</td>
<td><strong>3.18</strong></td>
<td><strong>3.18</strong></td>
</tr>
<tr>
<td>Tarantula</td>
<td>33.42</td>
<td>100.00</td>
<td>9.07</td>
<td>9.07</td>
</tr>
</tbody>
</table>
Figure 5: Faults found by ICD for different effort budgets and LS values using Ochiai.

For ICD and CH, most of the budgets reached more faults using LS(2). The exceptions are the tightest budget (5), in which LS(1) performs better; and some instances of higher budgets (from 80 to 100 for ICD and from 50 to 100 for CH), in which LS(3), LS(4), or LS(5) reaches more faults than LS(2). Levels from 3 to 5 find more faults in such cases because higher budgets allow the inspection of more blocks, which conversely leads to less faults found in more restricted budgets. Although we do not show the results of this analysis for Tarantula, the results obtained presented similar behavior.

The lower the level, the lesser the number of inspected blocks to reach a fault. Thus, it seems reasonable choosing a level that reaches more faults for most of the effort ranges and does not increase excessively the amount of blocks to inspect in the methods.

LS(1) is very restrictive and may result in finding too few faults. For cases in which a faulty block have the highest suspiciousness, LS(1) will reach the bug inspecting the lowest amount blocks possible. In many other cases, LS(1) will not reach the bug. Conversely, LS with \( l > 2 \) may lead to the inspection of too many blocks, increasing the chances of a developer giving up on using the technique. When the faulty block has the highest suspiciousness, LS(2) will reach it, but useless blocks may also be inspected. Faulty blocks with lower
scores have more chance of being found as $l$ increases. However, a good fault localization technique is supposed to classify faults among the higher scores.

We used LS(2) in the evaluation. This value seems reasonable in practice, since only two levels of suspiciousness scores are investigated using such a strategy. Moreover, using LS(2) seems closer to the developer’s behavior. A developer investigating a method will check the most suspicious blocks, perhaps the second and third most suspicious ones, but not all blocks. The Level Score must be lower enough to avoid the inspection of an excessive number of blocks, but it should not be too restrictive to not include the fault.

5.3. Which is the most helpful filtering strategy for fault localization?

Using FB, both ICD and CH had better results than BL with statistical significance. CH also exhibited better results using LS compared to BL. The combination between ICD and LS presented the worst results in the experiments.

FB is an aggressive strategy for small budgets (e.g., 15 blocks). Even for a budget of 10 blocks, CH-FB and ICD-FB reached more than half of the faults using Ochiai. Moreover, they found more faults than BL, which means that they help to reach additional faults, especially for small budgets.

We used LS with a value equal to 2, which means that inspecting two levels...
of suspiciousness scores is enough to find a large amount of the faults for the CH roadmap. Regarding the number of faults found, the results of CH-LS(2) had no statistical significance compared with BL. However, CH-LS(2) reached more faults than BL using Ochiai or Tarantula. Conversely, LS(2) and ICD were less effective than BL, which indicates that the ICD roadmap using LS(2) does not improve fault localization.

As a result, we conclude that FB has a better overall performance. Its effectiveness is independent of the roadmap chosen by the developer. If s/he decides to use CH or ICD, the strategy of choice should be FB.

5.4. How many methods are inspected to reach the bugs using roadmaps and filtering strategies?

Figures 3 and 4 show the number of blocks inspected using the CH and ICD roadmaps until hitting the bug site. These blocks are spread across several methods. A valid question a developer using CH and ICD may pose is: how many methods belonging to a roadmap should I inspect to locate a bug?

Figures 7 and 8 show the number of methods inspected to locate the bugs for Ochiai and Tarantula, respectively. For CH and ICD using LS, a fixed number of methods are inspected because the LS value is always 2. For CH and ICD using FB, each budget has its own number of inspected methods. We present the highest number of faults found through the different budgets.

In Figure 7, ICD-FB found 30 faults inspecting only the first method of the roadmap and 28 and 27 faults were found by CH-FB and CH-LS(2), respectively. This means that ICD with the filtering strategy FB classified the faulty method in the first position for 48.39% of all faults (62), and 57.69% of the faults found (52) using Ochiai. A percentage of 70.97% of all faults were reached inspecting at most four methods using CH-FB, which represents 84.62% of the faults found. The techniques reached a great number of faults by inspecting only a few methods: more than 50% of all faults for ICD-FB, CH-FB, and CH-LS(2) after inspecting two methods. ICD-LS(2) found less faults, 35.48% of all faults after inspecting four methods.
The results for Tarantula (Figure 8) show that ICD-FB, CH-FB, and CH-LS(2) also reached a similar amount of bugs by visiting less methods: 19 faults were reached by inspecting one method. CH-FB found 36 faults by inspecting up to four methods. For all treatments, Ochiai reached more faults than Tarantula. If a developer using CH with Ochiai is recommended to inspect at most four methods, s/he will locate around 70% of the bugs, independently of the filtering strategy. For this particular experiment, a small number of the roadmaps’ methods lead to the localization of a sizable amount of bugs. Thus, the number of methods can be used as a guide for debugging based on SFL techniques, since
a developer tends to use a technique that will likely find bugs by inspecting only a few methods.

5.5. Are roadmaps combined with filtering strategies more effective than BL for programs containing multiple faults?

Figures 9, 10, and 11 show the effectiveness results using Ochiai for all multiple-fault versions, 2-fault versions, and 4-fault versions, respectively. Considering all multiple-fault versions (Figure 9), ICD-FB and CH-FB reached more faults than BL, especially for small budgets. CH-LS(2) was slightly better than BL, and ICD-LS(2) had the worst performance. The results are similar to the ones obtained by looking at the 2-fault and 4-fault versions separately.

Figure 9: Effectiveness of treatments for all multiple-fault versions using various budgets for Ochiai.

Figure 12 shows the results for all multiple-fault versions using Tarantula. All techniques performed worse compared to Ochiai. CH-LS(2) had better overall performance, and even ICD-LS(2) outperformed BL using Tarantula. We omitted the results for the 2-fault and 4-fault versions for Tarantula because they are similar to those shown in Figure 12.

As in Section 5.1, we performed a statistical analysis of the multiple-fault versions to compare (1) the number of blocks inspected by each technique to reach the first faults for each budget ($ST_1$) and (2) the number of faults found
Figure 10: Effectiveness of treatments for 2-fault versions using various budgets for Ochiai.

Table 12: Budgets with statistical significance for multiple-fault versions

<table>
<thead>
<tr>
<th>Treatments</th>
<th>Ochiai</th>
<th>Tarantula</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH-LS(2) x BL</td>
<td>5–100</td>
<td>10–100</td>
</tr>
<tr>
<td>ICD-LS(2) x BL</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CH-FB x BL</td>
<td>5–100</td>
<td>5–100</td>
</tr>
<tr>
<td>ICD-FB x BL</td>
<td>5–100</td>
<td>10–100</td>
</tr>
</tbody>
</table>

by each technique for all budgets ($ST_2$). Tables 12 and 13 show the statistical results for $ST_1$ and $ST_2$ regarding the multiple-fault versions, respectively. The $2 \& 4$, $2$, and $4$ columns represent all multiple faults, 2-fault, and 4-fault versions, respectively. We can see that CH-FB and ICD-FB locate the first fault better than BL with statistical significance for almost all budgets. CH-LS(2) is better than BL for large budgets in the 2-fault versions using Ochiai and for all multiple-fault versions and budgets using Tarantula. For $ST_2$, CH-FB and ICD-FB can locate more faults than BL with statistical significance in most cases, while CH-LS(2) is better for the 2-fault versions using Ochiai and for all multiple-fault versions using Tarantula.

Overall, the results for multiple faults show that at least one of the faults is well-ranked by the techniques in most cases for all techniques, including BL, which resulted in a large number of faults found in the small budgets. ICD-FB, CH-FB, and CH-LS(2) performed better than BL for almost all budgets. The
Figure 11: Effectiveness of treatments for all 4-fault versions using various budgets for Ochiai.

Table 13: P-values (%) for $ST_2$ for multiple-fault versions

<table>
<thead>
<tr>
<th>Treatments</th>
<th>Ochiai 2 &amp; 4</th>
<th>2</th>
<th>4</th>
<th>2 &amp; 4</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH-LS(2) x BL</td>
<td>76.59</td>
<td>0.13</td>
<td>0.01</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICD-LS(2) x BL</td>
<td>100.0</td>
<td>34.82</td>
<td>18.89</td>
<td>34.42</td>
<td>18.89</td>
<td></td>
</tr>
<tr>
<td>CH-FB x BL</td>
<td>3.81</td>
<td>3.81</td>
<td>9.01</td>
<td>0.01</td>
<td>0.13</td>
<td>0.01</td>
</tr>
<tr>
<td>ICD-FB x BL</td>
<td>0.01</td>
<td>0.03</td>
<td>0.01</td>
<td>1.4</td>
<td>9.01</td>
<td>0.13</td>
</tr>
</tbody>
</table>

prevalence of at least one fault observed in our results corroborates findings of previous work [37]. Thus, our techniques performed well in programs with multiple faults. We caution that more effort is needed to locate the other existing faults.

6. Threats to validity

The discussion regarding threats to validity focuses on internal, external, and constructing validities. The internal threats are the tools used to produce the block lists and roadmaps. The implementation of these tools was manually checked by applying them to small programs; the data collected in the experiment, however, were manually checked by sampling due to the size of the programs.

Regarding the external validity, we used programs from different domains...
The constructing validity relates to the suitability of our effectiveness metric. Effectiveness compares the ability of a technique to fit the bug site within a particular effort budget (from 5 to 100 blocks). Our choice of the effort budgets was arbitrary. However, our focus on lower budgets aims at replicating realistic scenarios for debugging techniques. Nevertheless, user studies should be performed to assess this issue.

Our experiment was built to evaluate how quickly a technique will reach the fault site. We caution, though, that reaching the bug site does not necessarily mean identifying a fault. It has been shown that the perfect bug detection assumption, which presumes that a developer will identify a faulty statement simply by inspecting it, is not always guaranteed in practice [6].

Figure 12: Effectiveness of treatments for all multiple-fault versions using various budgets for Tarantula.

(e.g., mathematics, database systems, text processing), and with fairly large sizes, to expose the techniques to a variety of contexts. However, the results presented in this paper cannot be generalized for other programs.
7. Related work

7.1. Method-level spectrum

The search for bugs in methods or functions has been a goal since the very inception of automated debugging. Shapiro [38] proposed an algorithm to automatically locate the faulty procedure in a program. His approach requires that the developer answer whether the outputs of procedure calls are correct for a particular set of parameters. Unfortunately, such an approach does not scale for large programs, since many procedure calls should be checked.

Dallmeier et al. [39] used method call sequences occurring in test executions to indicate classes that are more likely to contain faults. Burger and Zeller [40] used method call coverage to obtain information from interactions between methods that occur during the execution of a failing test case. After applying delta debugging [41] and program slicing, this information was used to generate a minimal set of interactions relevant to the failure. Mariani et al. [42] presented a technique—called Behavior Capture and Test (BCT)—that uses method coverage to generate an interaction model. This model is compared with interactions from failing test cases to indicate method calls that are more likely to contain faults. Shu et al. [43] presented a technique called Method-Level Fault Localization (MFL). MFL uses test execution to construct a dynamic call graph and a dynamic data dependence graph of inter-method data. These graphs are merged and applied to a causal inference technique to create a list of suspicious methods.

Other recent studies have also addressed the use of method-level information for fault localization. Kochhar et al. [19] conducted a survey study with 386 professional practitioners, asking for their expectations about fault localization research. Regarding the granularity level, most respondents prefer method level, closely followed by statements and blocks. Classes and component levels were less preferred. Laghari et al. [44] proposed a technique—called pattern spectrum analysis—that provides a list of suspicious methods. They identify patterns of sequences of method calls during the execution of integration testing. A frequent
itemset mining algorithm is used to select patterns for each method within a minimum support threshold. Each method is assigned with the highest score from its respective patterns of method calls.

Le et al. [45] also presented a method-level fault localization technique, called Savant. First, they cluster methods executed by failing and passing test cases. They generate method invariants for the clusters created in the previous step. Invariants that differ between passing and failing executions are deemed as suspicious. Then, they apply ranking metrics and use these results along with the invariants as an input for a learning-to-rank machine learning technique to produce a ranking model. This model is used to create lists of suspicious methods for faulty programs. Musco et al. [46] presented a technique, called Vautrin, that provides a list of suspicious methods. They use mutants to identify call graph edges (i.e., method calls) more likely to contain faults. The rationale is that if a mutant is killed by a test, some of its edges can be related to a fault.

Differently from other techniques that provide method-level information, CH and ICD also include block-level information for code inspection. Thus, two granularity levels are available for the fault localization task.

7.2. Contextual information

Several works have addressed the use of contextual information for fault localization. Jiang and Su [47] proposed a technique based on machine learning to find predicates that correspond to failures from control flow paths. Developers can inspect these paths to search for faults. The technique presented by Hsu et al. [48] analyzes parts of failing execution traces to identify subsequences of statements frequently executed, which can be examined to search for faults.

Cheng et al. [49] proposed a model of subgraphs composed of nodes (methods or blocks) and edges (method calls or method returns). These subgraphs are obtained from the difference between failing and passing control flow executions. A graph mining algorithm is then applied to obtain a list of the most suspicious subgraphs. Liu et al. [50] proposed a technique that uses variance analysis to locate faults from program behavior patterns. These patterns are
obtained from object behaviors that belong to the same class. The technique indicates the suspicious pattern messages that occur in a class during a failing execution. DiGiuseppe and Jones [51] presented the Semantic Fault Diagnosis (SFD) technique. SFD gathers information from the source code, such as developer’s comments, class and method names, and variable expressions. Then, the terms are normalized and classified using SFL. The outcome is a list of top terms related to a fault.

Gouveia et al. [7] proposed a technique that provides contextual information by grouping statements according to their code structure. Their technique, called GZoltar, assigns for each code level (e.g., packages, classes, methods) the highest suspiciousness of its internal statements. Three visualizations are provided for code inspection. Thus, a developer navigates the most suspicious packages, classes, and methods to locate the bug. A user study shows that the technique helps developers to find bugs. Li et al. [52] proposed a technique, called Swift, that interacts with developers during debugging. The technique provides a list of suspicious methods and asks developers about inputs and outputs of such methods. Then, the technique uses the developers’ feedback to recalculate the fault localization results.

Le et al. [53] proposed a technique—called Adaptive Multi-modal bug Localization (AML)—that combines spectrum- and information retrieval-based fault localization. AML uses program spectra, text from bug reports, and suspicious words—words from bug reports that are also present in source code comments—to indicate methods that are more suspicious of being faulty. Youm et al. [54] use information from bug reports, stack traces, source code structure, and source code changes in their information retrieval (IR)-based bug localization technique—called Bug Localization using Integration Analysis (BLIA). BLIA provides file- and method-level information for inspection.

CH uses an approach similar to GZoltar to generate the suspiciousness of methods. Additionally, CH establishes a list of methods for inspection and uses the number of blocks with the maximum suspiciousness score as a tie-breaker for methods with the same score. ICD differs from other techniques by
ranging pairs of method calls (caller-callee methods) executed by the test run. CH and ICD roadmaps are used along with the LS and FB filtering strategies. Thus, they combine both inter- and intra-procedural spectra to contextualize the search for faults. The roadmap establishes the order in which the methods will be inspected; filtering strategies establish how long a developer will inspect a method using block-level spectra.

7.3. Absolute ranking

Recent fault localization techniques were assessed using the absolute number of programs elements inspected until reaching a fault [53, 44, 45, 52, 46]. Absolute positions in a list seem to be closer to real debugging scenarios than percentage metrics [6, 11]. In the study of Kochhar et al. [19], a large amount of respondents considered using a fault localization technique that leads them to inspect at most five program elements.

Some studies evaluate fault localization techniques by restricting the number of program elements to be inspected. They use absolute values, i.e., effort budgets, to measure their fault localization effectiveness. Le et al. [55] proposed a technique to evaluate the output of SFL techniques to indicate whether this output should be used by developers. To evaluate the technique, they used the top ten positions of an output to verify when the approach is effective for classifying such outputs.

In this paper, we use the absolute ranking as a criterion for assessing our contextualization strategies at method- and block-levels. The values can be set to fit developer's choices.

8. Conclusions and future work

This paper presented techniques to provide contextual information for fault localization. Code Hierarchy (CH) and Integration Coverage-based Debugging (ICD) generate a roadmap to search for bugs. The roadmap is a list of methods that are more likely to be faulty. Each method contains a list of its basic blocks.
to allow for code inspection at a fine-grained level inside the methods. We
proposed two filtering strategies—Level Score (LS) and Fixed Budget (FB)—to
limit the inspection inside the methods, which are used along with roadmaps
to locate faults. Using LS or FB, the developer only needs to inspect the most
suspicious blocks. Thus, contextualization establishes an order of inspection for
the methods and which blocks to inspect in these methods.

We used the concept of effort budget to measure the effectiveness of the
techniques during the fault localization process. An effort budget represents
the amount of blocks a developer investigates before abandoning a fault lo-
calization technique. This metric leads toward more practical evaluations of
fault localization techniques by deeming amounts of code that are feasible for
practical use.

The experiments carried out in this work used sixty-two faults from seven
real programs containing from 2K to 158K lines of code. We compared the
effectiveness of our techniques with a block list (BL).

ICD and CH using FB (ICD-FB and CH-FB) were more effective to find
faults inspecting less code than BL with statistical significance. Moreover, ICD-
FB and CH-FB found more faults than BL with statistical significance over all
budgets using Ochiai. The results also show that ICD-FB, CH-FB, and CH
using LS (CH-LS) were more effective for small effort budgets, from 5 to 50
blocks, finding more faults than ICD using LS (ICD-LS) and BL. BL inspects
up to 50% more code than CH and ICD to find the same amount of bugs in
small budgets (e.g., 5 to 20 blocks). Moreover, 70% of the faults were found
by inspecting at most four methods using ICD-FB or CH-FB, and 67% using
CH-LS. ICD-FB, CH-FB, and ICD-LS(2) were also more effective for programs
containing multiple faults, especially with small budgets.

CH in association with FB seems to be a particularly good combination,
because it significantly improves the performance of Ochiai for small budgets
at a negligible extra cost. This improvement may be the tipping point between
success or failure of an SFL technique. We consider that techniques that re-
duce absolute number of blocks/methods are more likely to be adopted at real
development settings.

The results suggest that the contextualization provided by roadmaps and filtering strategies is useful for guiding developers toward faults, bringing together information of suspicious methods and their related suspicious blocks. Moreover, they improved the performance of SFL techniques (Ochiai and Tarantula), reducing the amount of blocks inspected to find bugs.

Other issues involving the proposed techniques require further investigation. We intend to perform user studies to understand how the roadmaps and filtering strategies are used in real development environments. In an exploratory experiment with the CH roadmap implemented in the CodeForest tool, programmers frequently used the roadmap and highly ranked the usefulness of it [56]. User studies can help us further assess the effort budget issue in practice.

The evaluation of other real programs, and an analysis to understand the influence of program characteristics (e.g., test suite size, fault types, cohesion, coupling) on the performance of fault localization techniques are important issues for future research. We also intend to assess roadmaps and filtering strategies along with other spectra such as definition-use associations (dua) and branches. Dua spectra seems particularly interesting because they indicate both suspicious lines and variables for investigation.

Acknowledgement

The authors acknowledge the support granted by the São Paulo Research Foundation (FAPESP), processes 2013/24992-2 and 2014/23030-5, and by the Research Nucleus on Open Source Software (NAPSoL) of the University of São Paulo.

References

References


[56] D. Mutti, Coverage-based Debugging Visualization, Master thesis, School of Arts, Sciences and Humanities, University of Sao Paulo, Sao Paulo, Brazil, 2014.