beszedes_242_24

Code Analysis Techniques for Debugging and Software Maintenance

DSc Dissertation

Árpád BESZÉDES

Szeged, 2024

beszedes_242_24

Contents

beszedes_242_24

List of Tables

List of Figures

beszedes_242_24

1 Introduction

Research has shown – but developers also experience it in their daily work – that a significant part of software development costs, around 50-75%, can be attributed to various debugging activities, i.e. locating faults in the software and fixing them [51, 73, 142]. Despite the spread of various artificial intelligence-based supporting tools, this activity consists of mostly manual work, which requires great expertise and a very good knowledge of internals of the software in question. Thus, any solution that partially or fully automates the phases of fault localization and repair can result in significant resource savings.

Debugging is a complex activity that, in addition to the program code itself, often relies on other data such as test cases, error tickets, and various documentation. In most cases, however, the source code is the most important source of information, so its analysis is a fundamental part of any related technique [52]. Code analysis can be either static (without executing the software), in which case we collect general information about the code such as dependences between code elements, or dynamic, when we analyze the relationship of code elements to each other during specific program executions or their other dynamic properties.

In this dissertation, we discuss source code analysis methods that support debugging in various ways and which are designed to be practically usable for real size software systems and executions. In addition to debugging, these solutions can also be used in other areas of software maintenance, such as impact analysis, program comprehension, code documentation or software quality measurement.

When the goal is to identify a software bug (and eventually fix it) based on a faulty program behavior the first step is a detailed reproduction of the bug. This usually means running the software using a set of test cases so, in this case, the greatest emphasis is on *dynamic code analysis*. A significant part of the methods presented in this thesis are also aimed at this area.

Since this type of analysis processes the program execution, a large amount of data must be reckoned with—if we follow every detail of the program logic, for example the execution of individual instructions or elementary data. A common feature of the presented solutions is that they can be used in the case of real, large-scale programs since they handle the stored data sparingly, and in many cases the analysis is performed on elements with higher granularity (e.g., procedures instead of statements).

beszedes_242_24

Thesis points

The thesis points of the dissertation are divided into three parts, and the document follows this structure as well. The first set of thesis points are related to Dynamic Dependence Analysis:

T1.1 **Graph-Less Dynamic Program Slicing.** I describe a general framework that can be used to specify a set of related dynamic program slicing algorithms, the common basis of which is the static Definition-Use relationship and the linear processing of the execution history. The specific algorithms with their most important properties and possible applications are provided as well (Chapter 2).
Related publication: [4].

T1.2 **Dynamic Function Coupling.** I present the concept of Dynamic Function Coupling, which is based on the relative sequence and distance of procedure calls, and the related metric. Potential applications of Dynamic Function Coupling in the field of debugging and software maintenance are then discussed (Chapter 3).
Related publication: [3].

The topic of the following thesis point are the Dependence Clusters, for which some basic background information is provided in Chapter 4.

T2 **Computation and Analysis of Dependence Clusters.** I define the concept of Dependence Clusters based on Static Execute After relations between procedures and present their comparison with other dependence cluster concepts of higher granularity. The results of the theoretical and empirical investigation of such dependence clusters with possible applications are given as well (Chapter 5).
Related publications: [9, 10, 11, 24, 25].

The third set of thesis points deals with the enhancement possibilities of Spectrum-Based Fault Localization. Chapter 6 provides background information related to this topic which makes the understanding of the following chapters easier.

T3.1 **Reliable Code Coverage Measurement.** Reliable code coverage is essential for various dynamic code analysis tasks, including Spectrum-Based Fault Localization. Here I describe the practical difficulties we face when measuring code coverage. I present the evaluation methodology with which code coverage tools can be objectively evaluated in detail. The problems found during our specific tool evaluation and recommendations for handling them are introduced as well (Chapter 7).
Related publications: [17, 30].

T3.2 **Use of Call Chains in Spectrum-Based Fault Localization.** The most common variants of Spectrum-Based Fault Localization methods are based on simple code coverage. I present how dynamic call chains between procedures calculated during program execution can be used as context information in fault localization and describe the different ways of using call chains in existing algorithms (Chapter 8).
Related publication: [8].

T3.3 **Use of Call Frequencies in Spectrum-Based Fault Localization.** Traditional code coverage-based fault localization does not take into account the number of times each program element was called during execution. In this thesis point, I examine how the basic algorithms can be adapted to make use of procedure call frequency and introduce the measurement results of the different methods (Chapter 9).
Related publications: [31, 32].

T3.4 **Use of Dynamic Slicing in Spectrum-Based Fault Localization.** The accuracy of fault localization based on code coverage can be improved by using the much more precise dynamic program slices in the spectrum matrix instead of the coverage. I describe a theoretical model which shows the mathematical relationship between the two types of spectra, and under what conditions and to what extent the slicing-based fault localization improves the coverage-based method. Results of an empirical evaluation of this concept are also presented (Chapter 10).
Related publication: [26].

Finally, Chapter 11 concludes the dissertation and outlines some possible directions for future work.

beszedes_242_24

Part I

Dynamic Dependence Analysis

beszedes_242_24

2 Graph-Less Dynamic Program Slicing

2.1 Introduction

Program slicing [124, 129] is a program analysis technique proposed for many software engineering fields including verification and testing, maintenance, reengineering, impact analysis, program comprehension, debugging and others. In general, a slice of a program is its subset which is relevant from a specific computation's point of view. A slice may be an executable program or any other relevant subset of the program code. A *backward slice* is a subset that consists of all statements that might affect a set of variables at a specific program point, called the *slicing criterion*. A *forward slice*, on the other hand, is a set of program locations whose later execution might depend on the values computed at the slicing criterion. Typical applications of backward slicing are debugging and program understanding, while forward slices can be used for impact analysis, among others. If we determine the slice such that it involves the relations for any possible execution then it is referred to as *static slicing*, whereas if only one specific execution is addressed then it is *dynamic slicing*. A dynamic slicing criterion also includes the parameters of a concrete execution of the program (a test case with a set of program inputs) and a specific occurrence of the instruction involved during the execution.

While there are several works about the details and variations of static slicing, e.g. the work of Horwitz *et. al.* [80] which served as the starting point for subsequent implementations whose basis is the program dependence graph (PDG), a relatively few works addressed the practical sides of dynamic slicing. The basic dynamic slicing methods use different concepts [43, 44, 70, 88, 92, 93]. Among those, we are mostly interested in the traditional dynamic dependence-based method by Agrawal and Horgan [44] who used a graph representation called the Dynamic Dependence Graph (DDG). It includes a distinct vertex for each occurrence of a statement and the edges correspond to the dynamically occurring dependences upon program execution. A serious drawback of this approach is that the size of the DDG is proportional to the number of executed instructions and not to the program size, which makes it impractical for real size software and executions.

In previous work, we proposed a backward dynamic slicing algorithm that computes all possible dynamic slices globally with only one pass through the execution history [14]. The details of the algorithm for C programs [6] and for Java [123] have been elaborated, and its usefulness has also been demonstrated for various applications [2, 14]. Based on the original

idea it is possible to construct similar graph-less algorithms. We investigated all practical ways for computing the dynamic slices based on dynamic dependences but without requiring costly global preprocessing prior to slicing.

We proposed alternative methods that are based on the same dynamic dependences but instead of dynamic dependence graphs various data structures are maintained. These are different depending on the slicing scenario. We investigated global vs. demand driven slicing, computing backward vs. forward slices, and whether it traverses the execution history in a forward or in a backward way. This totals eight possibilities, of which some give useful algorithms, while others are impractical combinations.

2.2 Algorithms

2.2.1 Notations

All of the presented algorithms operate on two data sets: the execution history, which is a simple list of instruction occurrences called actions, and the static D/U representation of the program, which records the defined and used variables at each program point along with a special handling of branching instructions. Since the aim is to introduce the basic algorithms, the discussion will be limited to simple programs of a simple programming language. The algorithms will process the trace in either forward or backward way to follow the dynamic dependences. The dynamic slices will be computed "on the fly" during processing the execution history taking into account the local definition-use information.

In the rest of the chapter, the following notations will be used. The execution history, $EH = \langle {i_1}^1, {i_2}^2, \ldots, {i_j}^j, \ldots, {i_J}^J \rangle$, is a sequence of actions. Action i^j means that the i^{th} instruction of the program is executed in the j^{th} step of the execution. The length of EH (i.e. the total number of the steps executed) is denoted by J. Furthermore, the notations $i(i^j) = i$, $j(i^j) = j$, $EH(j) = i^j$, and $EHI(j) = i(EH(j))$ will also be used.

The static representation of the program is called the *D/U program representation*. It captures local definition-use relationships between the variable occurrences within each instruction. For simplicity, we will assume that each instruction defines exactly one variable and uses zero or more variables. An instruction i of a program is represented as $(i.\ d : U)$. The defined variable at i is d, while U is denotes the set of variables used to compute the value of d. The whole program is represented as $DU = \langle (1.\ d_1 : U_1), (2.\ d_2 : U_2), \ldots, (i.\ d_i : U_i), \ldots, (I.\ d_I : U_I) \rangle$ where i is an instruction serial number and I is the total number of instructions in the program. We will use the following shorthand notations: $DU(i) = (i.\ d_i : U_i)$, $d(i) = d_i$ and $U(i) = U_i$.

In the D/U representation control dependences are handled similarly to data dependences using *predicate variables*. For each predicate instruction i in the program (like *if* or *for*), we define a predicate variable p_i denoting the decision value computed in the predicate such as $d(i) = p_i$, and for each instruction k that is control dependent on i, $p_i \in U(k)$ will hold.

For the formalization of the dynamic slicing algorithms some more notations will be used. The *last definition* of a variable v before action i^j will be denoted by $LD(v, j)$, which is a function returning the action at which v was defined last before the j^{th} step in the execution history. We will also use the shorthand notations for the last defining step $LD(v) = j(LD(v, j))$, statement $LS(v) = i(LD(v, j))$ and action $LA(v) = LD(v, j)$ for an actual step j that is being processed during the execution of the slicing algorithm. After processing a step i^j, $LD(d(i)) = j$, $LS(d(i)) = i$ and $LA(d(i)) = i^j$ will hold for each subsequent action until $d(i)$ is defined next time.

We will use the notation $CB = (\mathbf{x}, i^j, V)$ for the backward, and $CF = (\mathbf{x}, i^j)$ for the forward slicing criterion, where $\mathbf{x}$ is a program input corresponding to a specific execution of

the program, i^j is the action for which the dynamic slice needs to be computed and $V \subseteq U_i$ is a subset of the *used* variables of i for which we compute the slices. The forward dynamic slice of i^j is computed for d_i.

2.2.2 Overview

The slicing algorithms operating on a given D/U representation and execution history of the program can be categorized according to three kinds of properties:

- **Slice direction.** If we search for program locations whose earlier execution affected the value computed at the criterion, we speak of a *backward slice*. A *forward slice* is a set of program locations whose later execution depends on the values computed at the slicing criterion.
- **Global or Demand-driven.** The traditional approach is to compute one slice at a time (in a single pass of the trace) for a single criterion. This is what we call *demand-driven slicing*. However, there is an opportunity to compute multiple slices for different criteria during a single pass over the trace. This is what we call *global slicing*.
- **Processing direction.** Given a trace, we may process it in both directions. *Forward processing* of the trace seems to be the natural one (and the only feasible in some applications), however traversing the trace *backwards* can also be applied in some situations.

In Table 2.1, we list all the 8 possibilities resulting from the above classification. As can be seen, some of them lead to practical algorithms, while others are impractical. In the followings, the number in the first column will be used to identify the algorithms.

No.	Global/Demand-driven	Slice direction	Processing direction	Usefulness
0	Demand-driven	Backward	Backward	Practical
1	Demand-driven	Backward	Forward	Impractical
2	Demand-driven	Forward	Backward	Impractical
3	Demand-driven	Forward	Forward	Practical
4	Global	Backward	Backward	Parallel
5	Global	Backward	Forward	Practical
6	Global	Forward	Backward	Practical
7	Global	Forward	Forward	Parallel

Table 2.1: Overview of dynamic slicing algorithms

The last column of the table is used for a preliminary classification of the algorithm according to its usefulness. In Section 2.2.3 the two practical demand-driven algorithms, in Section 2.2.4 the two practical global algorithms, and finally in Section 2.2.5 the two parallel algorithms will be discussed. The remaining two types of demand-driven algorithms are impractical, since to compute a demand driven slice in a reverse direction would virtually mean performing global dependence tracking.

2.2.3 Demand-driven algorithms

Given a slicing criterion, demand-driven algorithms will produce a single dynamic slice. The algorithms traverse the execution trace starting with the action of the dynamic slicing criterion, and follow the dynamic dependences with the help of the D/U representation going towards the beginning or the end of the trace, depending on the slice direction.

Backward Slice – *Algorithm 0*

This algorithm processes the trace starting with the action of the criterion and traces back the dependences towards the very first action. The algorithm (formalized in Figure 2.1) has a worklist with actions on which *CB* is (transitively) dependent but which are not yet processed. When an action from the worklist is processed, its instruction is added to the slice, and the last defining actions of its used variables are inserted in the worklist. When the worklist becomes empty the algorithm terminates by providing the slice. (A similar algorithm was sketched by Korel as well [91, 92].)

program Algorithm-0(P, CB)
input: P : a program
$CB = (\mathbf{x}, i^j, U(i))$: dynamic slicing criterion
output: S : dynamic slice of P for CB

begin
1 Read and store EH up to i^j
2 $S := \emptyset$
3 $worklist \leftarrow i^j$
4 **while** $worklist \neq \emptyset$
5 $k^l :=$ remove element with biggest l from $worklist$
6 **if** $l \neq j$ **then** $S := S \cup \{k\}$
7 **for** $\forall u \in U(k)$ **do** $worklist \leftarrow LD(u, l)$
8 Output S as the backward dynamic slice for criterion CB
end

Figure 2.1: Demand driven algorithm for backward slices

Due to the reverse processing of the trace, the *EH* needs to be stored (at least up to i^j), and the last defining action (in line 7) is not directly accessible, which are drawbacks of the algorithm. The *LD* problem can be mitigated while the *EH* is stored by constructing the so-called execution history table (*EHT*), which stores the defining actions in a way it can be effectively searched for a variable (u) and then by maximum step number under a limit (l). Note, that when removing the action in line 5, removing the one with the highest step is not required, but helps to avoid multiple processing of the same actions and can also be useful for *EHT* implementation.

Forward Slice – *Algorithm 3*

Computing forward dynamic slices starting from the slicing criterion means traversing the execution trace in a natural way. The basic idea of the algorithm (given in Figure 2.2) is to mark those variables during the processing of the trace that (transitively) dynamically depend on *CF*. If the processed action uses a marked variable (line 8), its defined variable will be marked and the instruction will be added to the slice (lines 9-10), otherwise we unmark that variable (line 11) as it is redefined without using the value computed in *CF*. The algorithm terminates if all variables gets unmarked or we reach the end of the trace.

2.2.4 Practical global algorithms

In a number of applications more than one slice may be needed at a time. It is possible to compute many slices during one pass of the trace by executing the demand driven methods in parallel for multiple criteria. Furthermore, a clever parallelization can reduce the

beszedes_242_24

program Algorithm-3(P, CF)
input: P : a program
$CF = (\mathbf{x}, i^j)$: dynamic slicing criterion
output: S : dynamic slice of P for CF

begin
1 Read EH
2 $mark(d(i))$
3 $S := \emptyset$
4 $k := j$
5 **while** $\exists\big(v$ is variable **and** $marked(v)\big)$ **and** $k < J$
6 $k := k + 1$
7 $l := EHI(k)$
8 **if** $\exists\big(u \in U(l)$ **and** $marked(u)\big)$
9 $mark(d(l))$
10 $S := S \cup \{l\}$
else
11 $unmark(d(l))$
12 Output S as the forward dynamic slice for criterion CF
end

Figure 2.2: Demand driven algorithm for forward slices

overall computational costs due to reusing several intermediate results. The global parallel algorithms incorporating this kind of operation are described in Section 2.2.5.

However, in those approaches the data structures for *all slicing criteria* need to be maintained throughout the whole execution history. Fortunately, it is possible to construct more practical algorithms in which only the actual dependence sets of the variables of the program needs to be maintained, which significantly reduces the space requirements. An interesting duality in these approaches is that the trace processing direction is the reverse of the slice direction. In this section we will describe the two basic dynamic slicing algorithms, which are able to produce *all* dynamic slices for a given execution of a program.

Backward Slice – *Algorithm 5*

The global backward slicing algorithm (in Figure 2.3) requires a forward processing of the execution history. It keeps track of the actual transitive dynamic backward dependences (*DynDep*) and the last defining statement (*LS*) of all variables. These information are updated for the variable defined in the actually processed action (i^j) using the last defining statement and the dynamic dependences of the variables used in the action. Right after this update, $DynDep(d(i))$ will contain the backward slice for CB_j.

This is the most practically usable algorithm among the 6 presented ones, as it allows on-the-fly processing of the trace. It has been presented in previous publications, and was implemented in different contexts [2, 6, 14, 123].

Forward Slice – *Algorithm 6*

The global forward slicing algorithm (in Figure 2.4) requires a backward processing of the execution history. For each variable v it maintains a "live" set (*LiveAt*), which holds statements of processed actions with defined variables (transitively) dependent on the latest previous

beszedes_242_24

program Algorithm-5($P, \mathbf{x}$)
input: P : a program
$\mathbf{x}$: a program input
output: backward slices for all $CB_j = (\mathbf{x}, i^j, U(i))$ criteria ($j = 1 \ldots J$)

begin
1 Read EH
2 **for** $j = 1$ **to** J
3 $i := EHI(j)$
4 $DynDep(d(i)) := \bigcup_{u_k \in U(i)} \Big(DynDep(u_k) \cup \{LS(u_k)\} \Big)$
5 $LS(d(i)) := i$
6 Output $DynDep(d(i))$ as the backward dynamic slice for criterion CB_j
end

Figure 2.3: Global algorithm for backward slices

(not yet processed) definition of v. Thus, just before action i^j is processed, the *LiveAt* set of the defined variable (d_i) contains the forward slice of i^j. Then, the *LiveAt* set of the variables used in instruction i must be extended with i and the *LiveAt* set of d_i, as their values have influence on the later values of d_i. If d_i is not dependent on itself, then the influence of the previous definition of d_i ends here, thus $LiveAt(d_i)$ must be set empty (line 6).

program Algorithm-6($P, \mathbf{x}$)
input: P : a program
$\mathbf{x}$: a program input
output: forward slices for all $CF_j = (\mathbf{x}, i^j)$ criteria ($j = 1 \ldots J$)

begin
1 Read and store EH
2 **for** $j = J$ **downto** 1
3 $i := EHI(j)$
4 Output $LiveAt(d(i))$ as the forward dynamic slice for criterion CF_j
5 **for** $u_k \in U(i)$ **do** $LiveAt(u_k) := LiveAt(u_k) \cup LiveAt(d(i)) \cup \{i\}$
6 **if** $d(i) \notin U(i)$ **then** $LiveAt(d(i)) := \emptyset$
end

Figure 2.4: Global algorithm for forward slices

Backward processing of the trace is not as straightforward as forward processing, but otherwise this algorithm is a usable counterpart of *Algorithm 5* to compute all dynamic forward slices of an execution of the program.

2.2.5 Parallel global algorithms

In this section, two global algorithms are presented that use the slice direction for processing the trace and compute the slices for all criteria. We called these *parallel global* algorithms because they compute all slices in parallel – virtually having many parallel demand-driven algorithms. However, they still have the advantage over computing all the slices with the demand-driven algorithms: the dependences arising from a specific action are computed only once.

Forward Slice – *Algorithm 7*

Figure 2.5 shows our parallel algorithm for computing forward slices. As the trace is processed forward, the algorithm tracks dependences of the defined variable $d(i)$ of the processed action i^j. This is similar to *Algorithm 5* except that these dependences are actions, not instructions. Then the slices of actions on which $d(i)$ depends must be extended with instruction i. The slices of all actions will be ready when we reach the end of the trace.

program	Algorithm-7($P, \mathbf{x}$)
input:	P : a program
	$\mathbf{x}$: a program input
output:	forward slices for all $CF_j = (\mathbf{x}, i^j)$ criteria ($j = 1 \dots J$)

begin
1 Read EH
2 **for** $j = 1$ **to** J
3 $\quad i := EHI(j)$
4 $\quad DynDep(d(i)) := \bigcup_{u_k \in U(i)} \Big(DynDep(u_k) \cup \{LA(u_k)\} \Big)$
5 $\quad LA(d(i)) := i^j$
6 $\quad$ **for** $a_k \in DynDep(d(i))$ **do** $S(a_k) := S(a_k) \cup \{i\}$
7 **for** $i^j \in EH$ **do** Output $S(i^j)$ as the forward dynamic slice for criterion CF_j
end

Figure 2.5: Forward algorithm for forward slices

Backward Slice – *Algorithm 4*

Just as in the case of *Algorithm 7* and *5*, the parallel backward slice algorithm (in Figure 2.6) is similar to the practical forward slice algorithm (*Algorithm 6*) that processes the trace backwards. The main difference here is that we track actions instead of instructions in the *LiveAt* sets of the variables. Slices of actions at which $d(i)$ is live must be extended with instruction i before the *LiveAt* set updates. The slices of all actions will be ready when we reach the beginning of the trace.

2.3 Conclusions

In this chapter, six algorithms for computing backward and forward dynamic slices were presented. All of them are based on computing the dynamic dependences by traversing the execution history, and using a simple static representation of the program containing local definition-use information, which uniformly incorporates both data and control dependences.

Algorithms 5 and 0 were implemented for the C language [1, 6], for which we had to solve some special problems, like handling dynamic variables and composite data types. Converting all variable accesses to memory locations simplified all data flow problems. The trace generation problem was solved by source code instrumentation. For the Java language, algorithms 5 and 7 have been implemented [123]. This implementation works on Java byte code, but the result can be converted to the source. To handle some problems (like multithreading and reflection) correctly, the slicer simulates some necessary tasks of the virtual machine while processing the trace. To produce the trace an instrumented Java Virtual Machine was used.

beszedes_242_24

program Algorithm-4($P, \mathbf{x}$)
input: P : a program
$\mathbf{x}$: a program input
output: backward slices for all $CB_j = (\mathbf{x}, i^j, U(i))$ criteria ($j = 1 \ldots J$)

begin
1 Read and store EH
2 **for** $j = J$ **downto** 1
3 $i := EHI(j)$
4 **for** $a_k \in LiveAt(d(i))$ **do** $S(a_k) := S(a_k) \cup \{i\}$
5 **for** $u_k \in U(i)$ **do** $LiveAt(u_k) := LiveAt(u_k) \cup LiveAt(d(i)) \cup \{i^j\}$
6 **if** $d(i) \notin U(i)$ **then** $LiveAt(d(i)) := \emptyset$
7 **for** $i^j \in EH$ **do** Output $S(i^j)$ as the backward dynamic slice for criterion CB_j
end

Figure 2.6: Backward algorithm for backward slices

A question arises in which contexts is each of the presented algorithms most useful. Choosing between a global or demand-driven algorithm depends on the number of criteria for which slices are needed. Based on our experience, if more than a few dozens of slices are needed, the global algorithms have a better overall performance – the exact number depends on various features of the actual program dependences. The choice between the global algorithms has two dimensions. The practical algorithms maintain less data structures and use less memory. However, the reverse processing of the trace might be problematic for storage reasons if the length of the *EH* is large. So, in this case the global algorithms that process the trace in a forward way might be better choices than their backward processing counterparts.

Contribution

This chapter is based on the publication:

[4] **Árpád Beszédes**, Tamás Gergely, and Tibor Gyimóthy. *Graph-less dynamic dependence-based dynamic slicing algorithms.* In Proceedings of the 6th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2006), pages 21–30, Philadelphia, PA, USA, September 2006.

The paper received 14 independent citations so far, one of which is an international and the other one a USA patent.

The basic concept and framework for defining the graph-less dynamic slicing algorithms is mostly my contribution, while the details of the exact algorithms and their evaluation from the practicality point of view and their possible applications are joint work. Some of my other notable papers that have contributed to this result are: [1, 2, 7, 28].

3 Dynamic Function Coupling

3.1 Introduction

Impact analysis [60] is an important software engineering activity which is used for change propagation [110], regression testing [116] and debugging [84], among others. In general, an *impact set* of a program element x (such as a statement or a procedure) is a set of other program elements which, when executing or changing x, may be impacted and hence should be considered in a follow-up activity. In a debugging scenario, investigating the impact set of a variable being observed at a particular program point gives us important information about how a potential fault could be propagated from that variable to other parts of the system. Different kinds of impact set computation methods have been proposed in literature, including static and dynamic code analysis-based ones. In this chapter, we deal with dynamic code analysis and discuss an approach for impact analysis which is based on the relative sequence and distance of procedure calls (we will use 'functions' interchangeably with 'procedures' in the following to refer to the same programming concept).

Apiwattanapong *et al.* [48] introduced a simple dynamic approach for impact set computation at function level, which is based on Execute After (EA) sequences computed from function entry and exit events. The basic approach is that a specific function f will potentially have an impact on all functions that are executed sometime *after it* in any of the executions. This approach is safe but very imprecise: all functions that appear after f in the execution trace will be in the impact set, thus no potential dependences will be missed.

Here, we further develop the notion of EA sequences by providing a method for computing more precise impact sets with the trade-off of losing the safety of the approach by allowing potentially missed dependences. The basic idea for refining EA relations is based on the intuition that the "closer" the execution of two functions, the more likely they are dependent on each other.

Our approach is the following. The measure *Dynamic Function Coupling* (DFC) is defined between two functions as the minimal level of indirection between all possible occurrences of the two functions in the execution traces. Informally, the level of indirection is the "closeness" of the two functions taking into account the number of other intervening functions. Once we have the DFC metric for every pair of functions, we compute the impact set of a function f by taking those functions that have a DFC of at most some fixed cut-off value d.

Based on this heuristic, we introduce a method for computing the impact sets using a

fixed cut-off value and discuss its variations for potential applications. Such impact sets are proposed for the mentioned activities instead of (1) the imprecise Execute After relations, (2) the precise but expensive dynamic slices, and (3) imprecise and/or unsafe static dependency sets. The indirection level d serves as a parameter to balance between precision and recall. For example, the original EA relation can be computed with an infinite d (safe but imprecise), and only directly coupled functions can also be retrieved (more precise but unsafe).

3.2 Computation of Dynamic Function Coupling

Apiwattanapong *et al.* [48] defined the *EA* relation between functions f and g as: $(f, g) \in EA$ iff f calls g, or f returns into g, or f returns into a function h and function h later calls g (all calls and returns can be direct or transitive).

To simplify the formal definitions of our DFC relation, we will use the concept of *dynamic call trees*. For a program P and its execution with the trace T we define the dynamic call tree as a rooted tree G with the ordering of the neighbors at each vertex. A vertex $v \in G$ represents an instance of a function $f \in P$, the *root* vertex represent the *main* function. An edge $v \to u$ in G represents that instance v of f directly calls instance u of g. Edges $v \to u$ and $v \to w$ such that u precedes w as the child of v represents v calls u before v calls w. We will also use the term *f-to-g call path* ($f \rightsquigarrow g$) in G, which is a path from instance v of f to instance u of g such that v transitively calls u through d number of calls (in other words, the length of this call path is $|f \rightsquigarrow g| = d$).

The *Execute After* ($EA^{(d)}$), *Execute Before* ($EB^{(d)}$), and *Execute Round* ($ER^{(d)}$) relations with indirection level d are defined as follows:

$$
\begin{aligned}
(f,g) \in EA^{(d)}_{call} &\overset{\text{def}}{\iff} \exists (f \rightsquigarrow g) : |f \rightsquigarrow g| = d,\\
(f,g) \in EA^{(d)}_{ret} &\overset{\text{def}}{\iff} \exists (g \rightsquigarrow f) : |g \rightsquigarrow f| = d,\\
(f,g) \in EA^{(d)}_{seq} &\overset{\text{def}}{\iff} \exists h \in P : (h \rightsquigarrow f), (h \rightsquigarrow g) \in G \text{ and}\\
&\qquad h \rightsquigarrow f \text{ and } h \rightsquigarrow g \text{ having a common instance vertex for } h \text{ where}\\
&\qquad f \text{ is called before } g \text{ and } d = |h \rightsquigarrow f| + |h \rightsquigarrow g| - 1,\\
(f,g) \in EA^{(d)} &\overset{\text{def}}{\iff} \exists\, d' \leq d \,:\, (f,g) \in EA^{(d')}_{call} \cup EA^{(d')}_{ret} \cup EA^{(d')}_{seq},\\
(f,g) \in EB^{(d)} &\overset{\text{def}}{\iff} (g,f) \in EA^{(d)},\\
(f,g) \in ER^{(d)} &\overset{\text{def}}{\iff} (f,g) \in EB^{(d)} \cup EA^{(d)}.
\end{aligned}
$$

As can be seen, the $EB^{(d)}$ relation is simply defined by reversing the roles of the two functions, and $ER^{(d)}$ is defined by combining $EB^{(d)}$ and $EA^{(d)}$. Observe that $EA^{(\infty)}$ corresponds to Apiwattanapong *et al.*'s definition of the EA relation, while $ER^{(\infty)}$ gives the complete graph with the covered functions. It is also easy to verify that the *Execute Round* relation will be symmetric, while the other two are not. Also note that only one of EA_{call} and EA_{ret} need to be actually computed as they are the inverse of each other.

We define the *Dynamic Function Coupling* measure as the lowest d for which the two functions are connected by ER:

$$
DFC(f,g) = \begin{cases} \min\{d \mid (f,g) \in ER^{(d)}\} & \text{if such } d \text{ exists,} \\ \infty & \text{otherwise.} \end{cases}
$$

Observe that $DFC(f, g) = DFC(g, f)$ and $DFC(f, f) = 0$ for any two functions f and g.

The definitions above are given for one specific execution of a program, however they can be easily extended to multiple executions. Namely, the *EA*, *EB* and *ER* relations for a set of executions can be obtained by computing the respective unions of the individual relations, while the *DFC* metric for any two functions will be the minimal such value from the executions. In the following, we will assume that the computed data are obtained from a fixed set of test cases.

Based on the previous, we may now define the impact sets themselves. We will use three related concepts, *change impact*, *forward computation impact* and *backward computation impact*. The difference is if we follow the dependences in both directions, only forward or only backward, respectively. These impact sets can have applications in different tasks: in determining the impact of changes on the rest of the system a bidirectional version would be used, the forward approach is suitable, e.g., in regression test selection, while the backward approach is typically a debugging-related concept.

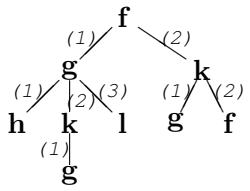
For a program, a set of test cases, a fixed indirection cut-off value d, and a set of changed functions C, the impact sets are defined as:

$$\begin{aligned} ImpactSet^{(d)}(C) &= \{g \mid \exists f \in C : (f,g) \in ER^{(d)}\}, \\ ImpactSetF^{(d)}(C) &= \{g \mid \exists f \in C : (f,g) \in EA^{(d)}\}, \\ ImpactSetB^{(d)}(C) &= \{g \mid \exists f \in C : (f,g) \in EB^{(d)}\}. \end{aligned}$$

3.2.1 Example

In Figure 3.1, an example trace T with function entry and return events (e and r subscripts, respectively), the corresponding call tree, the minimal d values of function pairs for EA_{call} and EA_{seq} subrelations, and the DFC values between functions are shown. The DFC values can be obtained by transposing the EA subrelation matrices and for each cell selecting the minimal value of the same cell from the two EA and two transposed matrices.

$$\begin{aligned} T = \langle & f_e,\ g_e,\ h_e,\ h_r,\ k_e,\ g_e,\ g_r,\ k_r,\ l_e, \\ & l_r,\ g_r,\ k_e,\ g_e,\ g_r,\ f_e,\ f_r,\ k_r,\ f_r \rangle \end{aligned}$$



$EA^{(d)}_{call}$

d	f	g	h	k	l
f	0	1	2	1	2
g	∞	0	1	1	1
h	∞	∞	0	∞	∞
k	1	1	∞	0	∞
l	∞	∞	∞	∞	0

$EA^{(d)}_{seq}$

d	f	g	h	k	l
f	∞	∞	∞	∞	∞
g	1	2	∞	1	2
h	3	2	∞	1	1
k	3	1	∞	2	1
l	3	3	∞	2	∞

DFC	f	g	h	k	l
f	0	1	2	1	2
g	1	0	1	1	1
h	2	1	0	1	1
k	1	1	1	0	1
l	2	1	1	1	0

Figure 3.1: An example trace, its dynamic call tree, minimal d, and DFC values.

In the following, algorithms for computing ER relations (and the corresponding *ImpactSet* sets) are given. We will present two variations depending on if we need impact sets for multiple functions at the same time (Section 3.2.2) or only for one (Section 3.2.3). Algorithms for computing the other types of relations and impact sets are straightforward modifications of these.

beszedes_242_24

3.2.2 Global algorithm for DFC

The global DFC algorithm computes the DFC metric for each function pair. The recursive algorithm shown in Figure 3.2 works on a trace T of length t, which is essentially a stream containing function entry and exit events. The maximal depth of the dynamic call tree will be denoted by m, while the number of functions covered will be n. For an actual subtree of the dynamic call tree rooted at function h, the algorithm works as follows. For all subtrees of h it first computes minimal h-to-f call path lengths recursively (line 8), and then the sequence-indirection levels while h being the "root" between any two f and g functions (lines 9–13). It then updates call-indirection levels with h (lines 16–19) and returns with updated call path length information.

```
program          ComputeDFC(T)
input:           T : trace
output:          D[f,g] : DFC between all functions f and g
begin
1   init all elements in D with ∞
2   E := Read T
3   ComputeDistances(E.function)
4   Output D
end
procedure ComputeDistances(function h)
local:           P[f] : array of previous values
                 N[f] : array of next values
begin
5   init all elements in P with ∞
6   E := Read T
7   while E is an ENTRY event
8         N := ComputeDistances(E.function)
9         forall f functions
10              forall g functions
11                  D[f,g] := min(D[f,g], P[f] + N[g] − 1)
12                  D[g,f] := D[f,g]
13        P := min(P, N)
14        E := Read T
15  P[h] := 0
16  forall f functions
17        D[h,f] := min(D[h,f], P[f])
18        D[f,h] := D[h,f]
19        P[f] := P[f] + 1
20  return P
end
```

Figure 3.2: Global DFC algorithm

Per trace element, this algorithm requires $O(n^2)$ time, while its memory requirement is $O(n \cdot m)$ not counting the DFC matrix itself.

3.2.3 On-demand algorithm for ER

The global algorithm presented in the previous section is less efficient in practice to compute impact sets since we are generally interested in the impact of only a small number of functions. In Figure 3.3, we present an algorithm that computes the set of impacted functions for a given changed function set. Processing the trace, the algorithm maintains $n+2$ stacks. The tops of $CALL$ and RET stacks show the indirection levels from the last changed function according to EA_{call} and EA_{ret}, respectively, while the top of $SEQ[g]$ shows the indirection from g, which corresponds to EA_{seq}.

On an entry event, new values are pushed onto the stacks depending on whether the entered function is a changed one or not (lines 6–8). The resulting impact set is also updated according to the stack tops (lines 11–16). On return events the stacks are updated by popping them, and the new tops of RET and SEQ stacks are updated using the popped value based on whether the returned function is a changed one or not (lines 19–24).

The time requirement of this algorithm is $O(n)$ per trace element, if appropriate data structures are used. The memory requirement is $O(n \cdot m)$.

Since our method will mostly be used with a small indirection levels in practice, it is useful to investigate more specialized versions of it for fixed d values, which can result in significantly better complexity requirements. For example, if $d = 1$, worst case complexities can be reduced to $O(m \cdot log(n))$ for time and $O(n \cdot m)$ for space, and average costs can be even better. Furthermore, a reduced version of the method incorporating only call-indirections can be used, which can be implemented in $O(1)$ time with respect to each step of the trace.

3.3 Experiments

We evaluate the DFC metric and the impact sets computed based on it in terms of the sets' size and accuracy. The set sizes are important since they indicate the amount of reduction that can be achieved compared to the conservative method (Section 3.3.2). The accuracy of the sets, on the other hand, is assessed through measuring the precision and recall rates with respect to precisely computed dynamic dependences among functions using dynamic program slicing (Section 3.3.1).

We performed our experiments on three medium size Java programs with their sets of test cases. The programs were JSubtitles (15 classes, 460 lines), NanoXML (27 classes, 1156 lines) and java2html (55 classes, 2290 lines) with 95–100 test cases each.

For computing precision and recall, we computed fine-grained, instruction level dynamic dependences with the Java dynamic slicer called Jadys [123]. The dynamic slices were lifted to function level and used as a golden standard for actual dependences. We used our global algorithm to produce the EA_{call} and EA_{seq} relations for different d values. Other relations were derived from these utilizing the symmetries between the relations.

3.3.1 Precision and Recall

Precision shows the rate of true positives in the resulting impact sets, which may contain false positives as well. Recall measures the rate of true positives over the total amount of actual dependences, which may include false negatives. There is a trade-off between these two measures and our parameter d provides the way to set the desired type of accuracy.

We measured the precision and recall for $EA_{call} \cup EA_{ret}$, EA as the original Execute After algorithm and ER for DFC itself. Figure 3.4 shows the overall results for ER for the subject programs. The other relation types showed similar shapes of the curves but with different values. Also, the three programs produced very similar results, although with different key

beszedes_242_24

```
program          ComputeER(T, C, d)
input:           T : trace
                 C : set of changed functions
                 d : cut-off indirection level
output:          IMP : change impact set of C
data:            CALL, RET : stacks of values
                 SEQ[] : vector of stacks with values
begin
1   IMP := C
2   init all stacks by pushing ∞ in them
3   while T is not empty
4         E := Read T
5         f := E.function
6         if E is an ENTRY event
7               if f ∈ C then push(0, CALL))
8               else push(top(CALL) + 1, CALL))
9               push(RET, ∞)
10              forall g functions do push(SEQ[g], top(SEQ[g]) + 1)
11              if top(CALL) < d then insert f into IMP
12              forall g ∈ C functions
13                    if top(SEQ[g]) < d then insert f into IMP
14              if f ∈ C
15                    forall g ∉ IMP functions
16                          if top(SEQ[g]) < d then insert g into IMP
17        else
18           pop(CALL)
19           u := min(pop(RET) + 1,pop(RET))
20           if f ∈ C then push(RET, 1)
21           else push(RET, u)
22           forall g functions
23                 if f ∈ C then push(SEQ[g], 0)
24                 else push(SEQ[g], u)
25           if top(RET) < d then insert f into IMP
end
```

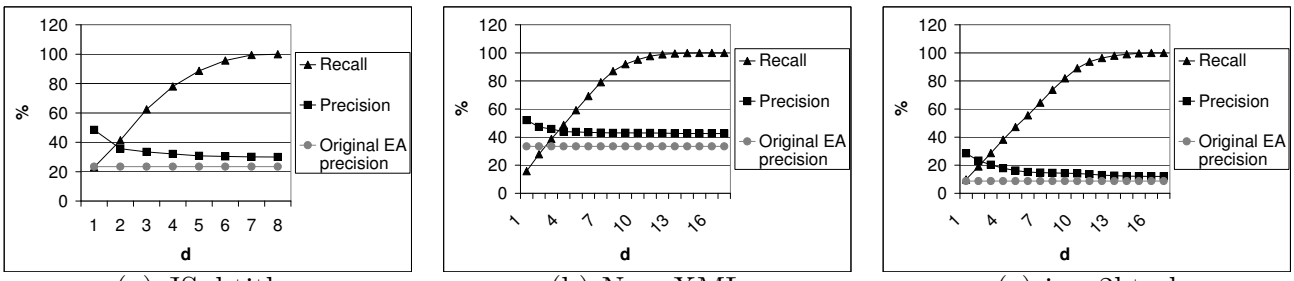
Figure 3.3: Algorithm for computing ER

values. It can be observed that as d grows recall also steadily grows with a relatively long linear phase, and at some point it reaches 100%. This suggests that there is a threshold level of d for every program with which our algorithm can be used with safety. On the other hand, precision starts at a higher value and rapidly decreases towards the precision of the original Execute After algorithm (but remaining somewhere above it because of *EB* relations).

3.3.2 Impact set size

Precision and recall are important, but when the method is applied the sizes of the impact sets will determine the efficiency of the software engineering task in question. We computed the set sizes for all d levels but since the tendencies of their change were very similar to the recall curves, we will present only the interesting values ($d = 1, 2$) for *ER* in a table

beszedes_242_24



(a) JSubtitles (b) NanoXML (c) java2html

Figure 3.4: Precision and recall for ER (%)

below. The mentioned similarity can be attributed to an interesting relation. Namely, $impact = \frac{recall}{precision} \cdot C$, where C is a constant value. As after a few steps of increasing d precision becomes nearly constant, the impact set size will grow proportionally to recall and will eventually reach a certain value when recall reaches 100%. Unfortunately, the sizes of the impact sets at 100% recall are not significantly smaller than that produced by the original conservative method.

	$d = 1$			$d = 2$		
	set size	orig. p.	prec.	set size	orig. p.	prec.
JSubtitles	14.3	23.4	48.4	34.5	23.4	35.6
NanoXML	13.5	33.4	52.1	25.5	33.4	47.2
java2html	4.2	8.7	28.5	9.9	8.7	23.2

Table 3.1: Impact set sizes. The set sizes are shown as percentage values relative to the respective set sizes of the conservative method, whose precision values are shown in the second and fifth column. The third and the sixth column are the respective precisions for ER.

In Table 3.1, the average sizes and precisions over all functions' impact sets are shown. At $d = 1$ the precision of *ER* is at least twice as good as of the original EA method, and it is still very good with $d = 2$. But, the corresponding impact set sizes are much smaller, although the recall is not very good compared to the 100% of the original Execute After.

3.4 Conclusions

Our initial expectation was that by reaching a good enough recall with a given d value, the precision will not start to decline significantly. Measurements proved the opposite: precision declines very fast, while recall steadily rises for some more indirection levels. Hence, we conclude that one should not aim at very high recall values using this approach since the gain in terms of impact set size and precision is minor with respect to the safe and simpler Execute After method. However, with small d values (1 or 2), the gain is notable: the average impact set size is much smaller than the one with the safe method and a double precision can be obtained using DFC.

A small DFC value between two functions indicates that there may be an actual coupling between them with higher probability. According to our measurements, all actual couplings can be identified by the level of maximum 5–15. The recall rates show that DFC levels 1 or 2 indicate significantly more actual dependences than at higher levels. The cut-off value of parameter d around 5–15 produces recall near 100%, having a similar precision at this point to the safe method (20–30%). The impact set sizes increase in a similar rate to the recall,

namely a small d will produce small sets with proportionally smaller recall as well. The relative gain compared to the conservative approach is also scalable with a characteristic similar to the recall values. Namely, the closest level 1 produces impact sets that are on average 13–15% of the set sizes of the safe method, while level 2 covers about 25–35%.

The possibility for parameterizing the computation algorithm with the cut-off value d enables its flexible application in various applications, which we list in the following.

Change impact analysis. In change impact analysis [110], when a change is made to a part of the system, the other parts of the system that need to be investigated in order to propagate the change may be computed using $ImpactSet^{(d)}$ with a fixed d. The method can be as follows. The initial impact sets are computed for all functions during a regular all-inclusive testing process. For subsequent changes, the actual database is used to determine the impact of the change, and during regression testing of the changes, the database can be updated by the newly computed impact sets. Some change propagation methods rely on high recall, in which case we can use a large d value (according to our experiments the smallest values where it reached 100% were around 8–15) for the updates, while other approaches benefit from better precision, in which case a close coupling ($d = 1$) should be chosen.

Regression testing. As far as regression testing is concerned, $ImpactSetF^{(d)}$ can be used with all traditional modification-based test selection strategies, again with a fixed d. We may maintain a database of impact sets as outlined above, and the test cases to be retested may be selected using these impact sets. Testing firewalls [116] are typically defined to involve only the closest dependents, in which case our impact sets with $d = 1$ can be a good alternative. Other regression testing approaches require the impact sets to be safer, in which case larger cut-offs may be chosen.

Debugging. Consider a scenario where a faulty value is observed at a specific program point, which is then marked with a breakpoint. Debugging the program step-by-step after it is not the optimal strategy. Additional breakpoints in the code could help but the programmer is not always certain where to put them. The $ImpactSetB^{(d)}$ set can be used here to determine the set of functions probably having effect on the erroneous value, and breakpoints can be put at the exit points of these functions to check the correctness of their return values. Varying the d value gives us flexibility: if a small value does not uncover the place of the bug we can use a higher value until the bug has been found.

Contribution

This chapter is based on the publication:

[3] **Árpád Beszédes**, Tamás Gergely, Szabolcs Faragó, and Tibor Gyimóthy. *The Dynamic Function Coupling Metric and Its Use in Software Evolution.* In Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR 2007), pages 103-112, Amsterdam, the Netherlands, March 2007.

The paper received 32 independent citations so far. Among others, it has been included in survey papers investigating impact analysis and program coupling literatures.

The concept of dynamic coupling based on function call sequences and distance with the analysis of possible applications of the method in debugging and software maintenance are mostly my contribution, while the details of computing the DFC metric with the related sets and the empirical evaluation of the method are joint work.

Part II

Dependence Clusters

beszedes_242_24

4

Background on Dependence Computation

Dependence Cluster analysis relies on underlying program dependence relations which are calculated using code analysis algorithms. In particular, we deal with two static analysis algorithms that provide the basic code dependences at two levels of granularity. Program Slicing (PS) is a code analysis technique of finer granularity and is defined at the level of statements. In this section, we first overview this concept, and then we introduce a coarser granularity algorithm called Static Execute After (SEA) that computes the dependences at the level of procedures, and enables a more efficient analysis of larger software systems. We also introduce some basic notations that will be used throughout Chapter 5.

Program Slicing

Program slicing [124, 129] is a classical code analysis technique, which aims to determine a subset of a program, called the *program slice*, by omitting the irrelevant code elements, such as statements, with respect to a specific calculation and from a specific perspective. A slice is computed with respect to a *slicing criterion*, which is a combination of a program variable and a location. The slice of the program includes only computations that are related to the variable at the program point defined in the criterion. Slicing approaches can be categorized by the direction of computation, i.e., *forward* or *backward*, depending on whether the result contains those instructions which depend on the value of the criterion variable, or affect it, respectively. A *static slice* includes information computed for all possible executions of the program, while *dynamic slices* are computed for a specific program input and execution.

Computing slices at the level of statements enables us to express the dependence relations at coarser level granularities as well – by lifting the dependences based on syntactical code structure, in particular, from statements to procedures. In the following, we will distinguish four different slice computations, referred to as *operators* based on what granularity level are the criterion and the resulting slice defined. Also, we will limit our investigation to *backward static slicing*, except where noted otherwise.

The slicing operators compute slices as the solution to a reachability problem over a program's *System Dependence Graph* (SDG) [80]. An SDG is comprised of vertices, which essentially represent the statements of the program and two kinds of edges: data dependence edges and control dependence edges. A data dependence connects a definition of a variable with each use of the variable reached by the definition [65]. Control dependence connects a predicate p to a vertex v when p has at least two control-flow-graph successors, one of which

can lead to the exit vertex without encountering v and the other always leads eventually to v [65]. Thus p controls the possible future execution of v. When slicing an SDG, a slicing criterion is a vertex from the SDG, and also the resulting slice includes a set of vertices. Hence, in the following we will use the terms 'vertices' and 'statements', as well as 'procedures' and 'functions' representing the same concepts, respectively.

The four SDG-based slicing operators $Slice^{VV}$, $Slice^{VF}$, $Slice^{FV}$, and $Slice^{FF}$ differ on the slicing criteria considered and on the set of elements returned as the result of the slice. The first one, $Slice^{VV}$, is the traditional vertex-level slicing [75] where the slicing criterion is an SDG vertex and the result is a set of vertices, V. With such a slice the criteria is dependent on each of the vertices found in V. The second, $Slice^{VF}$, has the same slicing criteria, a vertex, but produces a set of functions F rather than a set of vertices. In this case, the criteria is dependent on the (entry-point vertices of the) functions in F.

Parallel to the first two, the output of the final two operators, $Slice^{FV}$ and $Slice^{FF}$, is a set of vertices V and a set of functions F, respectively. These two differ in that the slice is taken with respect to an entire function instead of a single SDG vertex. For function f, this is done by taking the union of the slices for each vertex that represents source code from f.

Static Execute After

The concept of *Static Execute After (SEA)* and its counterpart *Static Execute Before (SEB)* have been introduced by Jász *et al.* [18] as a procedure-level alternative to instruction-based dependences captured by the SDG, which are easier to compute but only slightly more imprecise. Another good property of SEA/SEB is that they are conservative in the sense that they include all dependences computed by SDG-based slices. SEA/SEB does not require the computation or use of data dependences. Rather it uses only the possible control-flow paths and call-structures inside functions.

For functions f and g, we say that $(f, g) \in SEA$ if and only if it is possible that any part of g is executed after any part of f in any one of the executions of the program. Similarly, functions f and g are in SEB relation if and only if, it is possible that any part of g is executed before any part of f. It can be observed that these relations are inverse to each other just as is the case with backward and forward slices. A dynamic counterpart of this relation has been defined by Apiwattanapong *et al.* [48]. Following the notation of this publication and that for DFC from Chapter 3, we define the SEA relation involving (f, g) as follows (SEB can be defined in an analogous way):

$$\text{SEA} = \text{CALL} \cup \text{SEQ} \cup \text{RET} \text{ , where}$$

$$\left.\begin{array}{lll} (f,g) & \in & \text{CALL} \\ (g,f) & \in & \text{RET} \end{array}\right\} \iff \begin{array}{l} f \text{ calls } g \\ (g \text{ returns into } f) \end{array}$$

$$(f,g) \in \text{SEQ} \iff \begin{array}{l} \exists h : f \text{ returns into } h, \text{ then } h \text{ calls} \\ g \text{ (through a control-flow path)} \end{array}$$

Usually, the reflexive closure of this relation is considered, since any change in a particular function can (conservatively) affect any other part of it. Computing the SEA relation means following all possible control flow paths from a function to the rest of the system, and it is based on the Interprocedural Component Control Flow Graph (ICCFG) [5], a program representation that contains sufficient information to extract the required relations, while being much smaller and simpler than other graphs such as the System Dependence Graph [80] used for slicing.

Following the notation introduced for slice operators, we will use $Slice^{SEA}$ and $Slice^{SEB}$ to refer to function-level dependence determined by SEA and SEB relations, respectively.

5 Computation and Analysis of Dependence Clusters

5.1 Introduction

Dependences in computer programs are natural and inevitable. We can talk about dependences among any kind of artifacts such as requirements, design elements, program code or test cases, but the physical structure of the system as implemented are best captured by dependences within the source code. A dependence between two program elements (*e.g.* statements or procedures) basically means that the execution of one element can influence that of the other, hence the software engineer should be aware of this connection in virtually any software engineering task involving the two elements. One of the fundamental tasks of program analysis is to deal with source code entities and the dependences between them [52].

Dependences cannot be avoided, but they do not always reflect the original complexity of the problem. Sometimes unnecessary complexity is injected into the implementation, which may cause significant problems. *Dependence clusters* in program code are defined as maximal sets of program elements that each depend on the other [54], but other – from a computational point of view – more practical definitions exist as well. Large dependence clusters are detrimental to the software development process; in particular, they hinder many different activities including maintenance, testing and comprehension [41, 55, 56, 75, 59]. The primary problem is that in any dependence-related examination, encountering any member of a cluster forces us to enumerate all other cluster members. If large clusters covering much of the program code exist in a system, then it is very likely that one cluster member is encountered and consequently a large portion of the program code has to be considered.

The root causes of this phenomenon are not well understood yet; it seems to be an inherent property of program code dependence relationships. Sometimes, dependence clusters are avoidable because they actually introduce unnecessary complexity to the implementation; this is what Binkley and Harman call "dependence pollution" [54]. In such cases the program can be refactored using reasonable effort, but this is not always the case.

However, dependence clusters cannot be easily avoided in the majority of cases, so research should be focused on understanding the causes for the formation of clusters, and the possibilities for their removal or reduction. Previous work revealed that in many cases a highly focused part of the software can be deemed responsible for the formation of depen-

dence clusters [55, 56, 53]. Namely, program elements called *linchpins* are seen as central in terms of dependence relations, and are often holding together the whole program. If the linchpin is ignored when following dependences, clusters will vanish, or at least decrease considerably. The general approach to define a linchpin is to find a program element whose removal results in the largest decrease of clusters according to a given metric, but more efficient heuristic techniques are preferable.

In this chapter, we first introduce our dependence cluster concept based on Static Execute After and its calculation possibilities (Section 5.2). Section 5.3 presents our results related to the analysis of SEA-based dependence cluster properties using visual and metric-based approaches. In Section 5.4, we discuss the relationship of SEA-based dependence clusters and clusters based on SDG-based program slices. The problem of identifying linchpins in dependence clusters is overviewed in Section 5.5, in which we discuss our manual and metric-based identification approaches. Finally, we present our results of applying SEA-based dependence clusters in impact analysis and related applications (Section 5.6).

5.2 Static Execute After-based dependence clusters

Dependence clusters have originally been introduced by Binkley and Harman using static backward slices as the underlying program dependence [54]. This means that the set of dependent code elements that form a cluster are statements, or equivalently, vertices from the SDG program representation.

The notion of Static Execute After (SEA) relations is analogous to instruction level program slices, with the difference that the code elements that constitute a dependence set are procedures (functions or methods) instead of statements. This enables us to define a concept of dependence clusters based on SEA relations instead of slices in a straightforward way. There are two main benefits to this approach: (1) the computation of SEA is much more efficient and (2) in the case of real size software systems, statement granularity is too fine for general high level investigation.

We will use different definitions for the SEA-based clusters, starting from the theoretical concept and then working towards practical computability. (We will use SEA for illustration but all concepts can be defined in an analogous way for SEB dependences as well.)

Mutual dependence. This concept is based on the original definition given for slice-based clusters, according to which a dependence cluster is a *maximal set of procedures that each depend on the other based on the SEA relation.* This kind of dependence clusters is prohibitively expensive to compute because it is a form of the clique identification problem. Hence, in practical applications dependence clusters are defined based on the coincidence of dependence sets, as follows.

Same set. According to this definition, a SEA-based dependence cluster of a program is a *maximal set of procedures in which any two procedures have the same reflexively closed SEA sets.* The elements of these types of clusters are mutually dependent on each other (due to reflexivity), but the associated SEA sets may contain additional dependences which are not members of the cluster. In other words, a cluster can have at most as many elements as the common SEA sets. This is a reasonable approximation of the mutual dependence-based definition and is easier to compute since the dependence sets only need to be compared in a pairwise fashion. This definition has the additional good property that it gives a partitioning of the procedures into clusters. This still can be expensive to compute in certain situations, so an even simpler approach is used in many applications, which is the following.

Same size. This definition differs from the previous one only in that it does not compare the SEA sets themselves but only *their sizes*. It has been empirically shown that (in the case of non-trivial clusters) it is sufficient to test if two dependence sets have the same size with a good error margin [11, 24, 54, 75], so in most of the following discussion we will rely on this kind of cluster analysis.

We define our SEA-based dependence clusters more formally as follows. Let $P = \{p_1, p_2, \ldots, p_n\}$ be the set of procedures in a program X (for simplicity, we assume $n \geq 2$). The SEA relation of program X is the reflexive closure of the relation defined on its set of procedures, *i.e.* $SEA \subseteq P \times P$, according to the definition in Chapter 4. With $\bar{n} = \{0, 1, \ldots, n\}$, we give the following auxiliary definitions.

For any procedure p, two sets of procedures are associated with it: the set of procedures that are dependent on p (*i.e.* the procedures that are successors of p according to SEA), and the set of procedures on which p depends (*i.e.* the procedures that are predecessors of p according to SEA). The former is referred to as the *SEA-set of* p, while the latter is its *SEB-set* (as mentioned, these notions are analogous to forward and backward slices, respectively). Because the SEB relation is defined as the inverse of SEA, in the following we will use the notations SEA and SEA^{-1} for simplicity. Additionally, we use a notation that emphasizes the direction of the dependences as follows:

$$\vec{D}(p) = \{q \in P \mid SEA(p, q)\}$$

for forward dependences, and

$$\overleftarrow{D}(p) = SEA^{-1}(p) = \{q \in P \mid SEA(q, p)\}$$

for backward dependences.

Based on the different cluster notions introduced above, we define two types of dependence clusterings, one with set coincidence ($\mathcal{I}$) and one with size comparison ($\mathcal{S}$), both having backward and forward versions. For the latter we will need an additional definition of the dependence set size, the weight functions $\vec{w}$ and $\overleftarrow{w}$:

$$\vec{w} : P \to \bar{n}, \quad \vec{w}(p) = \left|\vec{D}(p)\right| \qquad \text{and} \qquad \overleftarrow{w} : P \to \bar{n}, \quad \overleftarrow{w}(p) = \left|\overleftarrow{D}(p)\right|$$

The formation of dependence clusters of a program based on SEA dependences is in fact a partitioning of the procedure set P (not being transitive, the SEA relation itself does not exhibit partitions). For forward dependence sets, we define the set of dependence clusters (*i.e.*, partitioning or clusterization) of program X as:

$$\vec{\mathcal{I}} = \left\{ \left\{ q \in P \mid \vec{D}(q) = \vec{D}(p) \right\} \,\middle|\, p \in P \right\}$$

$$\vec{\mathcal{S}} = \left\{ \left\{ q \in P \mid \vec{w}(q) = \vec{w}(p) \right\} \,\middle|\, p \in P \right\}$$

For any $c \in \vec{\mathcal{S}}$ the weights of its members are equal, so we can assign the same weight to cluster c itself. Clearly $\vec{D}(q) = \vec{D}(p)$ implies $\vec{w}(q) = \vec{w}(p)$, so $\vec{\mathcal{I}}$ is a refinement of $\vec{\mathcal{S}}$. This also means that the weight function can be extended naturally to $\vec{\mathcal{I}}$ as well. Note, however, that the weight and the size of a cluster are different notions, the former may also be referred to as dependence set size. The corresponding backward definitions can be derived from these easily, and the same considerations apply.

For the discussion that follows, we will use the *same size* dependence cluster concept ($\vec{\mathcal{S}}$ and $\overleftarrow{\mathcal{S}}$), and together with the different slicing operators defined in Chapter 4, the following notations will be used to denote the different clusterizations, respectively: $\mathcal{C}^{\mathcal{VV}}$, $\mathcal{C}^{\mathcal{VF}}$, $\mathcal{C}^{\mathcal{FV}}$, $\mathcal{C}^{\mathcal{FF}}$, $\mathcal{C}^{\mathcal{SEA}}$, and $\mathcal{C}^{\mathcal{SEB}}$.

beszedes_242_24

5.3 Analysis of dependence cluster properties

We present the results of our empirical investigation of SEA-based dependence clusters in a range of C/C++ subject programs of moderate (up to 200k lines) and large (millions of lines) sizes. The main aim of this research was to find out how common are large SEA-based dependence clusters in a variety of programs of different sizes and how can we categorize the programs more objectively in terms of their degree of clusterization?

We introduce the term *clusterization* to indicate the extent programs exhibit dependence clustering, and define novel metrics that characterize this property quantitatively. We follow two approaches to express and analyze clusterization levels:

1. By a manual classification based on the *visual inspection* of the Monotone Size Graphs of the subject program [54], which are composed of SEA dependence sets for all procedures in the system. An MSG of a program is a graphical representation of all its dependence sets by drawing the sizes of the sets in monotonically increasing order along the x axis from left to right (see examples in Figure 5.1). Three levels will be used to express the clusterization level for a subject program: *low*, *medium* and *high*.

2. By defining different *clusterization metrics*, which are designed to express clusterization in easily quantifiable numerical form (values from $[0, 1]$).

5.3.1 Subject programs and tool setup

We started with the collection of programs Harman *et al.* used in their experiments [75]. We could reuse 60% of these programs but also extended this set to finally arrive at 29 programs written in C (we will refer to this set as the *moderate size* programs, and they can be observed in Table 5.1). The second part of our data set consisted of two large industrial software systems from the open source domain. The first one was the WebKit system, which is a popular open source web browser engine integrated into several leading browsers [127]. It consists of about 2.2 million lines of code, written mostly in C++, JavaScript and Python. In this research we concentrated on C++ components only, which attributes to about 86% (1.9 million lines) of the code. In our measurements we used the Qt port of WebKit called QtWebKit on x86_64 Linux platform. We performed the analysis on revision 91555, which contained 91,193 C++ functions and methods as the basic entities for our analysis.

The other large system we used was the GNU Compiler Collection (GCC), the well-known open source compiler system [68]. The GCC system is large and complex, and its different components are written in various languages. It consists of approximately 200,000 source files, of which 28,768 files are in C, which was the target of our analysis. In terms of lines of code, this attributes to about 13% of the code, 3.8 million lines in total (note that in C, the size of individual functions is usually larger than that of an average C++ method, hence this difference in lines of code compared to WebKit). We chose revision 188449 (configured for C and C++ languages only) for our experiments, in which there were 36,023 C functions as the basic entities for our analysis.

We used our custom built tools as well as some existing components. To extract base program representations, as parser front ends we used Grammatech CodeSurfer [71] in the case of moderate size programs and Columbus [13] for the big programs. For the SEA dependence computation, our existing implementation of the SEA algorithm using ICCFG graphs [5] was applied. Since we needed to process and store a large number of dependence sets, we implemented additional tools (for MSG computation, cluster metrics computation, etc.) that employ efficient specialized data structures and algorithms. Specifically, we used the SoDA library [120] to store and process the dependence sets.

To obtain the dependence clusters and investigate the level of clusterization we computed the SEA-based dependence sets for both forward and backward directions for all procedures in our subject programs. The structure of our dependence sets is fortunately simple: for each procedure in a program we compute the corresponding set of procedures it is in SEA relation with. Hence, the total number of dependence sets equals the number of procedures in a program, and this number is also the maximal dependence set size. Altogether we computed 23,970 dependence sets for the moderate size programs, 182,386 for WebKit and 72,046 for GCC.

5.3.2 Manual classification

A cluster reveals itself as a wide plateau in the MSG visualization consisting of a number of equal-sized dependence sets. Typically, in a low class we cannot identify any plateaus, while for the high class there are one or two big ones, the rest being medium. Three typical programs from the moderate size category, one for each clusterization level, are shown in Figure 5.1. Visual classification reveals that the first program (`epwic`) does not show any plateaus and the landscape ascends in small increases, the second one (`findutils`) contains some moderately wide plateaus which are not significant individually but altogether cover much of the width of the landscape, and that in the last program (`gnubg`) there is a single plateau occupying nearly the whole width of the landscape.

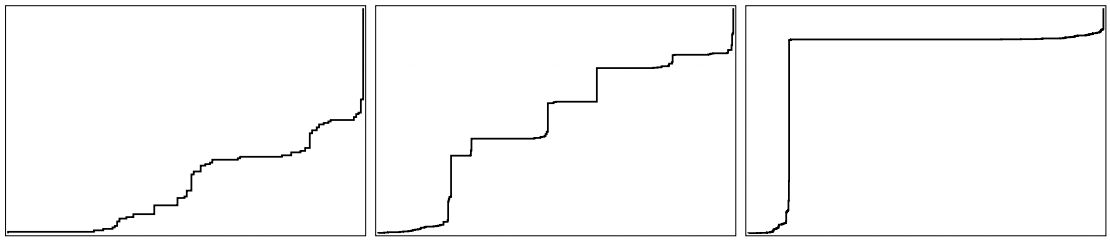


Figure 5.1: Example MSGs for the visual classification (`epwic`: *low*, `findutils`: *medium*, `gnubg`: *high*)

The MSGs for the two big programs in our dataset, GCC and WebKit, can be seen in Figure 5.2. The differences between the two programs are clear. GCC belongs to the low level clusterization category, while WebKit exhibits a medium level clusterization in the visual ranking.

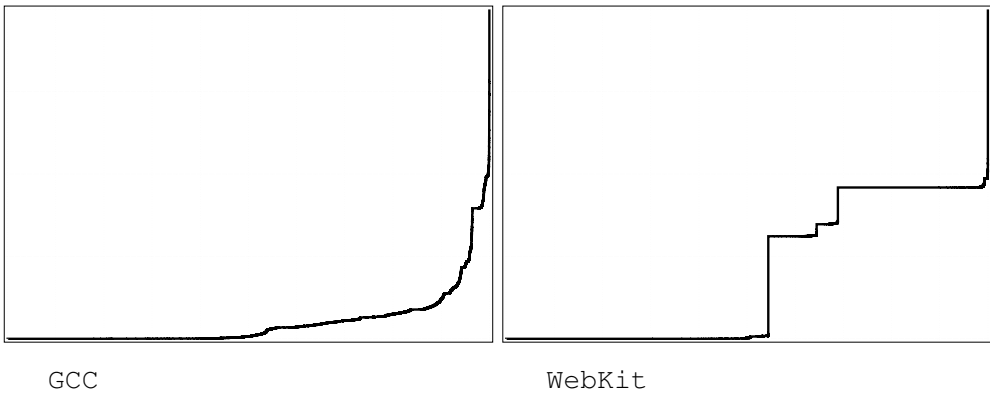


Figure 5.2: MSGs for `GCC` and `WebKit`

5.3.3 Metric-based analysis

We define all metrics for measuring clusterization so that they are comparable to each other for a given program and to programs with different sizes, so we normalize them to the interval $[0, 1]$, where 0 means no clusterization and 1 indicates maximum clusterization (we assume that the number of procedures, $n > 1$). All metrics can be defined in the same way for both forward and backward dependences, but we will omit the direction notation in this section for simplicity. Also, the metrics have to be interpreted in the context of a given program.

The first metric we used for the numerical expression of the level of clusterization is based on Binkley and Harman's work [54], who measured the area under MSG (referred to as AREA in the following). The apparent weakness of this metric is that it increases if all dependence sets are increased by the same amount, although – intuitively – clusterization should not be different in such cases. Programs with no dependence clusters can have both small and large dependence sets, and vice versa. The following is a formal definition of this metric. Normalization is done using the maximum possible area of the MSG, which also means that AREA is equivalent to the average dependence set size for a program:

$$\text{AREA} = \frac{1}{n^2} \sum_{c \in \mathcal{I}} |c| \cdot w(c)$$

Our next metric is based on an analogy of entropy and measures the "(dis)order" in the system of dependence sets in terms of their sizes (called ENTR). We consider a program more clusterized in this respect if there is a greater number of equal-sized dependence sets, *i.e.* when the entropy is lower (note, that this inverse relationship is required to obtain comparable metric intervals with the other metrics). This metric is defined as:

$$\text{ENTR} = \frac{\sum_{c \in \mathcal{S}} |c| \cdot \log_2 |c|}{n \log_2 n}$$

Our concept of regularity metrics is based on the number of partitions (REGU and REGX). The idea is that the fewer partitions there are, the larger their size must be, so there have to be more large clusters among them. Inversely, more partitions have to take more "regular" different sizes hence they will represent low clusterization. This metric has two variants: REGU is based on the *same size*, and REGX on the *same set* dependence cluster concepts, and they are normalized over the largest possible number of dependence sets:

$$\text{REGU} = \frac{n - |\mathcal{S}|}{n - 1} \qquad \text{REGX} = \frac{n - |\mathcal{I}|}{n - 1}$$

Moderate size subjects

Table 5.1 shows the clusterization analysis results for the moderate size programs, including the visual classification, and the clusterization metric values as well. Since the metrics are normalized to the range $[0, 1]$, their values could be visualized as small horizontal bars in the last four columns of the table.

Visual analysis produced three groups with 5 (low), 11 (medium), and 13 (high) elements, respectively. We would expect an ideal clusterization metric to yield values in such a way that the 5 smallest would be assigned to the "low" level, the middle 11 would be assigned to "medium", and the largest 13 to the "high" level group. Based on these criteria, the clusterization metrics can be characterized by counting how many programs they fail to assign to the group given by visual ranking. The counts are as follows: AREA $\rightarrow$ 10, ENTR $\rightarrow$ 7, REGU $\rightarrow$ 15, REGX $\rightarrow$ 8. The differences in these counts can also be observed by visual inspection of the metric values. It can clearly be seen that AREA and REGU are significantly

beszedes_242_24

Program name	**LOC**	**NP: # of procedures**	**Visual class**	**Clusterization metrics** AREA	ENTR	REGU	REGX
lambda	1766	104	▼ low				
epwic	9597	153	▼ low				
tile-forth	4510	287	▼ low				
a2ps	64590	1040	▼ low				
gnugo	197067	2990	▼ low				
time	2321	12	■ med				
nascar	1674	23	■ med				
wdiff	3936	29	■ med				
acct	7170	54	■ med				
termutils	4684	59	■ med				
flex	22200	153	■ med				
byacc	8728	178	■ med				
diffutils	17491	220	■ med				
li	7597	359	■ med				
espresso	22050	366	■ med				
findutils	51267	609	■ med				
compress	1937	24	▲ high				
sudoku	1983	38	▲ high				
barcode	5164	70	▲ high				
indent	36839	116	▲ high				
ed	3052	120	▲ high				
bc	14370	215	▲ high				
copia	1168	242	▲ high				
userv	8009	255	▲ high				
ftpd	31551	264	▲ high				
gnuchess	18120	270	▲ high				
go	29246	372	▲ high				
ctags	18663	535	▲ high				
gnubg	148944	1592	▲ high				

Table 5.1: Moderate size subject programs with clusterization information, sorted by Visual class and NP

worse than the other two metrics, while the difference is not so great regarding ENTR and REGX. ENTR is more precise on low and medium clusterization levels, while REGX performs better on highly clustered programs.

Large subjects

The ENTR values are 0.4347 for GCC and 0.6980 for WebKit, while REGX is 0.3134 and 0.3552, respectively, which supports our initial (visual) classification for these two systems. While ENTR shows a notable difference, in the case of REGX it is not so significant, which may also reflect our finding from above that ENTR was better for low or medium clusterization.

A question arises what makes GCC not having significant dependence clusters as opposed to WebKit: can we identify any properties that justify such a classification? After consulting with some key WebKit developers and showing them the members of the clusters, we came to the conclusion that clusterization is related to architectural concepts in the system. The most notable difference between the two systems in this respect is that while WebKit is essentially a library consisting of highly coupled elements for the distinct functional areas, GCC is a complex application but with much clear behavioral paths that are independent of each other. In WebKit, most complex functionalities are implemented in a set of highly interacting procedures (for example, webpage rendering is performed by several hundred procedures calling each other recursively). On the other hand, GCC implements functionalities like compiler optimization passes that are more isolated from each other. In addition, the two systems are written in different programming paradigms (C vs. C++) which may influence their internal structure.

5.4 SEA and slice-based dependence clusters

In this section, we present an empirical investigation and comparison of the dependence clusters identified by SEA-based and program slice-based dependence relations using a collection of 20 subject programs written in the C language. The goal of the experiment was to find out how well can SEA-based cluster analysis be used as a proxy for the more expensive slice-based cluster analysis (function-level vs. vertex-level)? In particular, we wanted to investigate how do the slice sets compare to SEA sets, what are the differences between cluster structures produced by slice sets and SEA sets, and how well do the different clusterization metrics perform on the clusterization of the two types?

For the different dependence types (defined in Chapter 4) to be maximally comparable we used a common analysis tool setup to the last point where the different dependence analyses diverge. Particularly, we used GrammaTech's CodeSurfer [71] to compute the common internal program representation, the System Dependence Graph (SDG). The SDG was used to compute the four SDG-based slice types using the two pass traversal algorithm by Horwitz et al. [80]. To compute $Slice^{SEA}$ and $Slice^{SEB}$, we created the ICCFG from the same SDGs and applied a reachability algorithm [18].

5.4.1 Comparison of dependence sets

The first experiment seeks to verify the relationship among the different dependence types in terms of their relative precision and recall. By definition, slice types with the same kind of criteria (vertices or functions) differ only in how we determine the elements of the slice since they are computed by the same underlying algorithm. Hence, $Slice^{VV}(c) \subseteq Slice^{VF}(c)$ and $Slice^{FV}(c) \subseteq Slice^{FF}(c)$ for any slicing criterion c (with vertices aggregated to functions

beszedes_242_24

in the results). $Slice^{FF}(c) \subseteq Slice^{SEB}(c)$ also holds due to the definition of SEB relation and the fact that we used the same underlying program representation in all cases.

The comparison of the different dependence sets showed that their sizes were comparable with the different slice types. This experiment confirmed our earlier findings that the $Slice^{SEB}$ sets are not much larger than $Slice^{FF}$ slices: the difference is between 3%–48%, the average being 11%, and the outliers are all in the very small programs. The difference between the two function-level slices $Slice^{FF}$ and $Slice^{FV}$ are also very similar: on average it was 12%. The other interesting observation from the data is that the difference between pairs of slices that differ in the criteria used, but not the counting granularity ($Slice^{VV}$ vs. $Slice^{FV}$ and $Slice^{VF}$ vs. $Slice^{FF}$, respectively) was small, about 2% on average for both cases. Based on this observation and to reduce the complexity of further analyses we limit our further investigations to the three slice types that take functions as their criteria, namely $Slice^{FF}$, $Slice^{FV}$ and $Slice^{SEB}$.

Subject	Clusterization level			AREA			REGX		
	$\mathcal{C}^{\mathcal{FV}}$	$\mathcal{C}^{\mathcal{FF}}$	$\mathcal{C}^{\mathcal{SEB}}$	$\mathcal{C}^{\mathcal{FV}}$	$\mathcal{C}^{\mathcal{FF}}$	$\mathcal{C}^{\mathcal{SEB}}$	$\mathcal{C}^{\mathcal{FV}}$	$\mathcal{C}^{\mathcal{FF}}$	$\mathcal{C}^{\mathcal{SEB}}$
acct	none	none	small	0.18	0.28	0.39	0.02	0.19	0.52
barcode	high	high	high	0.60	0.79	0.82	0.00	0.91	0.94
bc	high	high	high	0.65	0.77	0.93	0.04	0.77	0.92
byacc	small	small	medium	0.29	0.31	0.55	0.09	0.27	0.57
compress	small	small	medium	0.43	0.45	0.50	0.22	0.57	0.70
copia	huge	huge	huge	0.49	0.99	1.00	0.94	0.99	1.00
ctags	large	large	large	0.62	0.80	0.88	0.02	0.84	0.93
diffutils	medium	medium	medium	0.26	0.24	0.29	0.10	0.59	0.75
ed	large	large	large	0.66	0.78	0.80	0.08	0.88	0.90
epwic	none	none	small	0.11	0.11	0.13	0.30	0.27	0.37
flex	small	medium	medium	0.51	0.61	0.78	0.05	0.56	0.76
ftpd	medium	medium	medium	0.39	0.49	0.51	0.03	0.69	0.74
gnuchess	large	large	large	0.54	0.62	0.72	0.13	0.70	0.82
go	huge	huge	huge	0.90	0.95	0.96	0.01	0.96	0.98
indent	large	large	large	0.43	0.62	0.68	0.00	0.78	0.88
sudoku	small	large	huge	0.41	0.50	0.98	0.00	0.41	0.97
time	none	none	none	0.25	0.43	0.69	0.00	0.08	0.50
userv	large	large	large	0.41	0.54	0.60	0.10	0.72	0.85
wdiff	none	none	medium	0.26	0.37	0.68	0.07	0.07	0.68
wu-ftpd	none	none	none	0.07	0.13	0.16	0.03	0.22	0.21
Average				0.4230	0.539	0.6525	0.1115	0.5735	0.7495

Table 5.2: Clusterization metrics and dependence cluster classification

5.4.2 Manual analysis of dependence clusters

For this set of experiments, we used a combined MSG in which all three dependence types are shown on the same graph (see Figure 5.3). Note, that since these slice types share the same slicing criteria, the x-axis of the MSG is common, but due to the different granularity of the slice elements, the slices of $Slice^{FV}$ were scaled on the y-axis. The first observation one can make about the graphs in Figure 5.3 is that the three slice types typically produce MSGs with very similar shape. In only a few cases is there a significant difference (e.g., sudoku).

The result of the manual classification is shown in the columns 2–4 of Table 5.2. For this experiment, we used the five-level Likert scale ("none", "small", "medium", "large" and "huge"), which allowed us a systematic and relatively fine grained analysis of the cluster

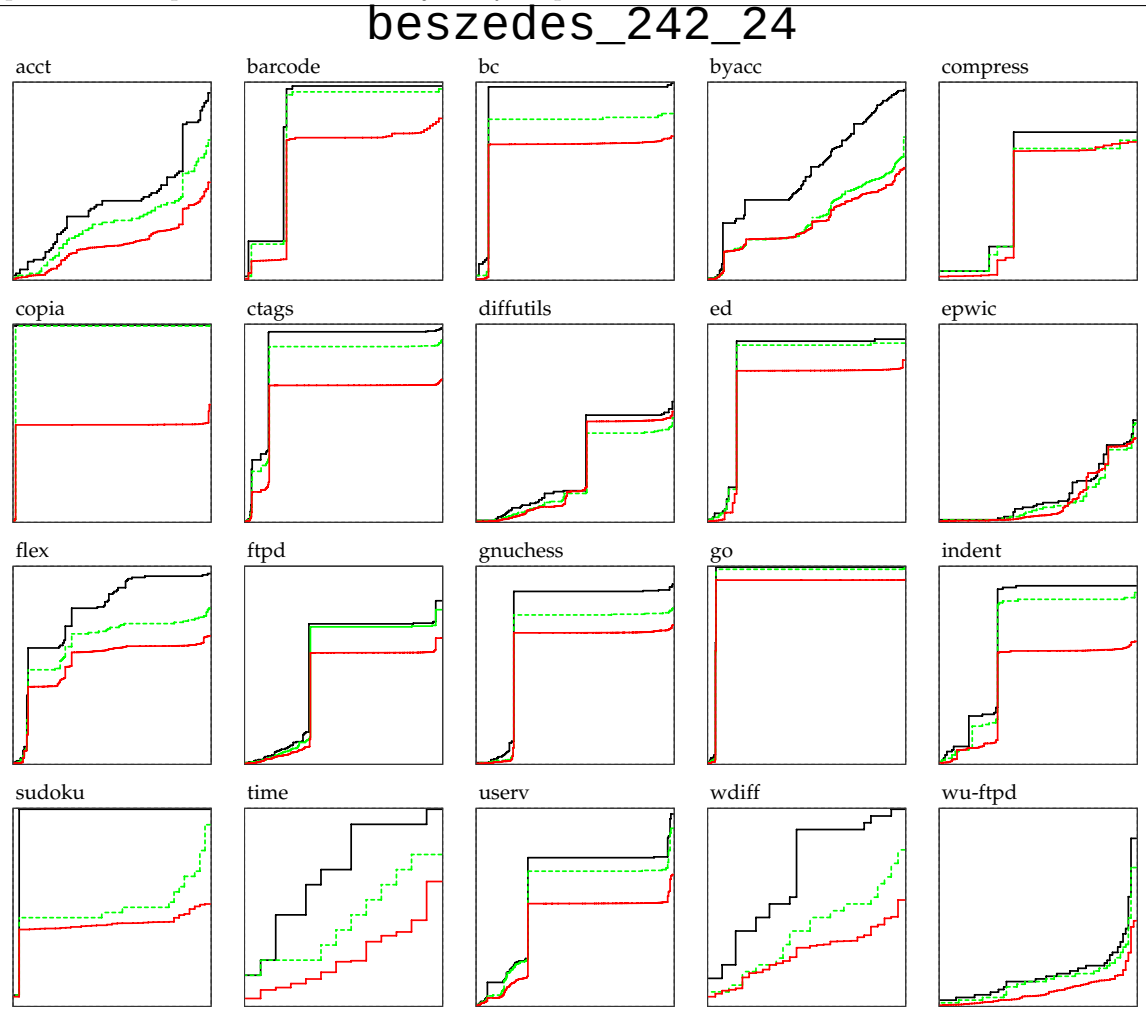


Figure 5.3: Subject program MSGs. The lowest solid red line shows $Slice^{FV}$, the middle dotted green line shows $Slice^{FF}$, and the upper black line shows $Slice^{SEB}$. Both axes on the graphs follow the number of program elements (vertices or functions, depending on the slice type) relative to all elements.

structures. As mentioned, clusterization was quite similar in many cases, but there are significant differences as well. For instance, while slice-based clusters for wdiff cannot be identified, a clearly visible cluster can be found with SEB.

5.4.3 Metric-based analysis of dependence clusters

In this section, we will rely on two clusterization metrics, AREA and REGX, adapted to be applicable to all slice types, not only SEA/SEB, as presented in previous section. We do not provide formal definitions for all the different variations of these metric since their adaptation is straightforward.

Table 5.2 provides the AREA and REGX metrics for each subject program and the three investigated dependence types. Comparing the manual cluster classifications to the metric values, generally it is not obvious which metric best reflects the level of clusterization. There are obvious cases such as programs copia and go, where large dependence clusters can be easily identified by large AREA values. However, there are cases, such as wdiff, byacc, and ctags, where AREA alone is not enough to determine clusterization. Here, REGX could provide additional information about the regularity of different set sizes. For instance, in the case of wdiff we observe a medium SEB-based cluster, which is reflected by the relative high value of REGX while the same metric for the other two slice types is low, indicating the absence of

clusters. Another example is ctags, which is classified as highly clustered. In this case the REGX metrics are bigger than AREA metrics, which indicates that in these cases REGX may be a better indicator than AREA.

5.5 Linchpin identification

Experience shows that in many cases, a highly focused part of the software can be deemed responsible for the formation of dependence clusters [53, 55, 56]. Namely, program elements called *linchpins* are seen as central in terms of dependence relations, and are often holding together the whole program. If the linchpin is ignored when following dependences, clusters will vanish, or at least decrease considerably. It is useful if one is aware of such linchpins, let alone be able to remove them by refactoring the program. However, currently even the first step (identifying linchpins) is largely an unexplored area. We still do not understand fully what makes a particular program point a linchpin, how they can be identified, or whether there is always a single element to be made responsible in the first place. The possibilities for linchpin removal by program refactoring are even harder to assess.

In this section, we present our experiments in relation to the identification of linchpin program elements. The general approach to locate a linchpin is to find a program element whose removal results in the largest decrease of clusters according to a given (objective) evaluation, but this approach is not practical due to the large number of trials, so other, more scalable techniques are required.

We identified the linchpins for the 20 C subject programs used in the experiments from Section 5.4. We used the mentioned brute-force method that checks all program elements (statements or procedures) for the amount of reduction they can produce in clusterization, and relied on both a visual inspection of the MSG graphs and a metric-based analysis. As the biggest challenge in this topic is how to locate linchpins using more efficient methods, we also investigated approximate heuristic methods for this task (on the subjects from Section 5.3) and compared their results to the exact results of the brute-force method.

5.5.1 Linchpin identification by brute-force

The simplest way to identify a possible linchpin in a program is to remove code elements one by one and see which one brings the biggest reduction in clusterization. However, this is not so simple because the level of clusterization is not easy to objectively measure in the first place – as we discussed in previous sections.

One possibility is *manual identification* by observing the resulting MSG graphs and visually determining the biggest differences in their shapes before and after removing the code element. As this is a very laborious process, we can use a *metric-based approach* as well, in which case we calculate the biggest gain (reduction in the clusterization) according to a given metric (metrics defined in Section 5.3 can be used for this purpose). In the following, *gain* will mean the amount the respective metric is reduced in percentage: $\frac{m-m'}{m}$ [%], m being the original metric value and m' the value after linchpin removal.

Specifically, we computed all dependence sets for a program by removing one code element at a time, *i.e.* by ignoring the candidate element and all of its dependences during dependence set calculations. We then determined the clusters and compared the clusterization metrics (we used ENTR and REGX in these experiments) of the reduced versions of the program to the corresponding metrics of the original program. This calculation was then repeated for all code elements in the program.

beszedes_242_24

Manual Linchpin Identification

First, we visually investigated the series of cluster structures represented in form of MSG graphs. To make the classification of linchpins more structured we applied a 5-level scheme: 5 (cluster broken), 4 (almost a 5), 3 (a bit of breaking is evident), 2 (almost a 1) and 1 (clearly no breaking or just a drop in slice size). For each program we then identified all potential linchpin elements, separately for each dependence type. We then further classified the linchpins into what we call a *"gold standard"* and a *"silver standard,"* which form the basis for further statistical analysis.

These two classifications capture the authors' intuition as to the level of clusterization with the gold standard being more rigorous and representing very obvious cases, while silver standard is more relaxed (and is a superset of the gold standard). We decided not to include any of linchpin candidates falling into categories 1–2, thus sliver standard included candidates of categories 3–5, and gold standard consisted of only category 5 linchpins.

In Figures 5.4 and 5.5 we present examples of gold and silver standard functions, respectively. If we compare the MSGs shown with their unreduced versions from Figure 5.3, we can clearly see the effect of linchpin removal: in the case of gold standard it is significant, while it is less pronounced with silver standard. For example, in the case of barcode all slice types produce significant cluster break, however with ed only $\mathcal{C}^{\mathcal{SEB}}$ produces a break, the slice-based clusters do not vanish, they are just reduced. Considering example silver standard reductions (Figure 5.5), the reduction for go is significant, however, a big cluster remains. Program byacc is the least evident: in fact, with $\mathcal{C}^{\mathcal{FV}}$ and $\mathcal{C}^{\mathcal{FF}}$ different linchpins could be identified than with $\mathcal{C}^{\mathcal{SEB}}$-based ones. The example shows a function that resulted in the slight break of clusters with $\mathcal{C}^{\mathcal{SEB}}$ but slice-based clusters were unaffected.

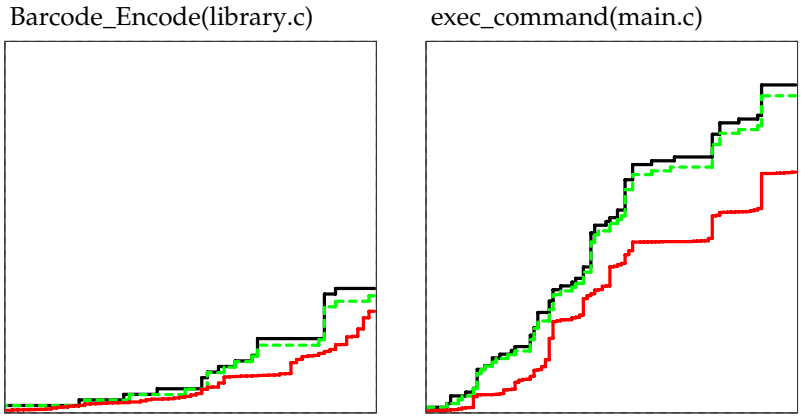


Figure 5.4: Gold Standard Patterns. Functions from barcode (left) and ed (right)

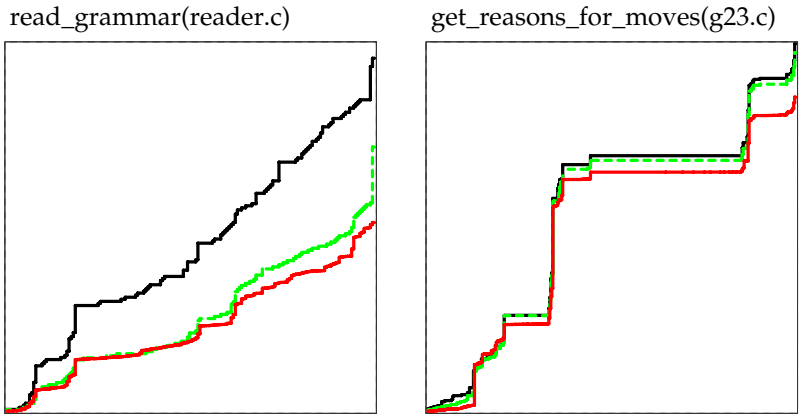


Figure 5.5: Silver Standard Patterns. Functions from byacc (left) and go (right)

beszedes_242_24

Metric-Based Linchpin Identification

The clusterization reduction gain measured by the metric values can be used to identify the potential linchpins by simply taking the code elements with the highest gain. However, in many cases the situation is not that evident because the gains are similar for multiple elements, and we could also use different metrics and different dependence types. Figure 5.6 shows examples of how different the rankings can be. The series of bars are given in decreasing order of a relative decrease of the metric compared to the unreduced program. Note that since the rankings are computed individually, the bars at the same rank position may represent different code elements. From the results for program bc we can clearly observe that the first element in the rank list is much higher than the rest in all three AREA metrics and in one of the REGX metrics (in this case the first element was the same), which clearly indicates a linchpin. However, the other example in this figure (from program ctags) exemplifies a different pattern. Here, just by looking at the ranks we could not clearly say if the first one or more elements might be linchpins. In fact, in the case of this program it turned out that the ranking according to REGX was closer to manual assessment than the AREA-based.

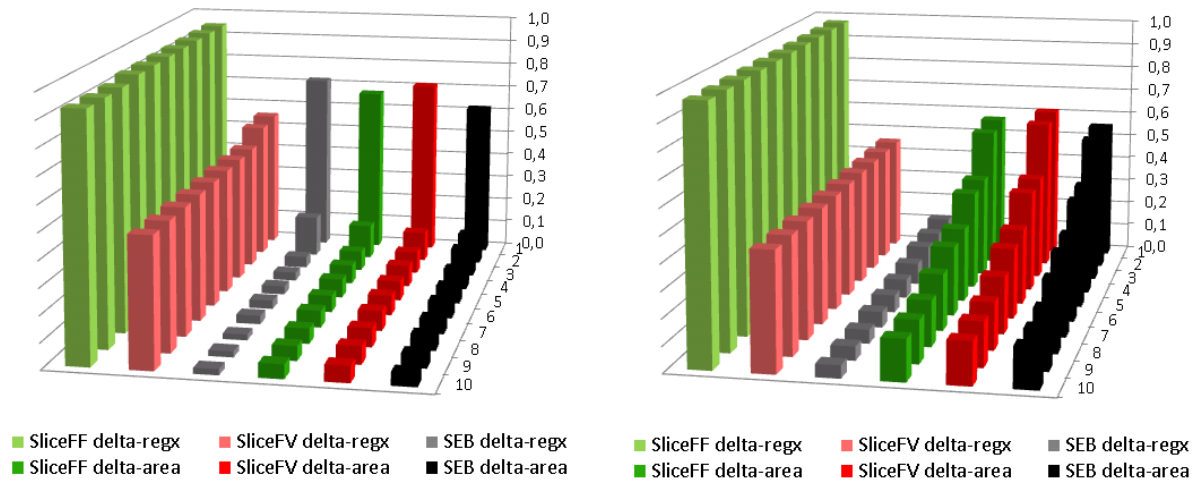


Figure 5.6: Linchpin rankings by different metrics for bc (left) and ctags (right)

For comparing the metric-based rankings to the manually identified linchpins we used the Average Precision (AP) and Mean Average Precision (MAP) measures, typically used in Information Retrieval for similar purposes [100]. Average Precision is more appropriate for ranked information retrieval than traditional precision and recall, which do not take the ranks into account. AP is precision (at each rank position) at each relevant document (linchpin in our case) averaged over the number of relevant documents, while the MAP value is the mean of the Average Precision values over all sets of queries (programs in our case). We computed AP values for each combination of program and ranking (determined by the different metrics and slice types), using both the gold and silver standard linchpins as the relevant documents.

Table 5.3 shows the Average Precision values for the gold standard linchpins. An obvious thing to observe from the data is that the metric based ranking performs exceptionally well. In other words, if the existence of linchpins is evident, it can be found by metric-based approach with high success. Particularly, AP is 1.00 for most of the programs in the gold standard category with the AREA metrics. Typically, the REGX metric is also a good indicator in these cases as it usually "follows" AREA in the case of high clusterization and evident linchpins. For silver standard linchpins, the results were similar, just slightly worse.

beszedes_242_24

	AREA			*REGX*		
Subject	$\mathcal{C}^{\mathcal{FV}}$	$\mathcal{C}^{\mathcal{FF}}$	$\mathcal{C}^{\mathcal{SEB}}$	$\mathcal{C}^{\mathcal{FV}}$	$\mathcal{C}^{\mathcal{FF}}$	$\mathcal{C}^{\mathcal{SEB}}$
barcode	1.00	1.00	1.00	1.00	1.00	1.00
bc	1.00	1.00	1.00	1.00	1.00	1.00
copia	1.00	1.00	1.00	1.00	0.32	1.00
ctags	1.00	1.00	–	1.00	0.27	–
ed	1.00	1.00	1.00	1.00	0.25	1.00
ftpd	1.00	1.00	1.00	1.00	0.50	1.00
gnuchess	1.00	1.00	1.00	1.00	1.00	1.00
indent	1.00	1.00	1.00	1.00	1.00	0.81
userv	0.50	0.50	–	0.33	0.00	–
MAP	**0.94**	**0.94**	**1.00**	**0.93**	**0.59**	**0.97**

Table 5.3: Gold Standard Average Precision

5.5.2 Heuristic determination of linchpins

Although being a brute-force method, we could successfully implement it in the experiment from the previous section because the subject systems were relatively small (1–29 thousand lines of code). However, it is not feasible in the case of real size systems, so we should look for alternative methods to approximate the linchpins to enable practical application of this dependence cluster related research.

The existence of dependence clusters and any related linchpins are determined by the structure of the dependences under investigation. Therefore, it is to be expected that by investigating the topology of the underlying dependence graph (ICCFG in the case of SEA dependence) one could gain insight into what makes a program point a potential linchpin.

The problem does not have an obvious solution, so we wanted to investigate whether local properties of the dependence graph nodes (procedures) could be leveraged to approximate linchpins. We used the following heuristic metrics as potential indicators: NOI (Number of Outgoing Invocations from the procedure), NII (Number of Incoming Invocations to the procedure), sum of the former two (SOI=NOI+NII), and their product (POI=NOI·NII). We tried the sum and the product because we expected that in linchpin formation both incoming and outgoing dependences could be important.

We used the moderate size subjects from Section 5.3 to experiment with these heuristics because we could apply the brute-force method on these programs. To compare the actual linchpins identified by the brute-force method to the performance of the heuristic metrics, we related two values for each procedure in the programs: a clusterization metric (ENTR or REGX) after removing the procedure and one of the heuristic metrics (NOI, NII, SOI, POI) associated with the procedure We then used Pearson and Kendall correlation checks between the corresponding vectors of these values.

In Table 5.4, we show Pearson correlation results. We marked the strongest correlation values for each program underlined; the last two rows show the average correlation values and the counts of strongest cases for each metric. It can clearly be seen that the NOI metric (Number of Outgoing Invocations) is the best estimator for both ENTR and REGX. The best values are negative in the NOI columns, which means that for the procedures of a program there is a high correlation between a high NOI value and a low clusterization value resulting from the removal of that procedure. In other words, the higher NOI value a procedure has, the more likely it is that its removal would decrease the clusterization considerably, *i.e.* the more likely it is that the procedure is a linchpin.

beszedes_242_24

	ENTR				REGX			
Program	**NOI**	**NII**	**SOI**	**POI**	**NOI**	**NII**	**SOI**	**POI**
lambda	0.30	0.53	0.50	0.58	-0.61	-0.49	-0.64	-0.57
epwic	0.28	0.12	0.32	0.32	-0.50	-0.02	-0.48	-0.32
tile	0.48	0.46	0.62	0.63	-0.27	-0.18	-0.29	-0.28
a2ps	-0.27	0.03	-0.16	-0.04	-0.57	-0.01	-0.39	-0.40
gnugo	-0.45	0.04	-0.06	0.01	-0.53	-0.01	-0.13	-0.05
time	0.70	-0.29	0.47	0.70	-0.55	0.08	-0.47	-0.12
nascar	-0.13	-0.18	-0.23	-0.41	-0.77	0.15	-0.76	-0.33
wdiff	0.04	-0.23	-0.02	-0.50	-0.89	0.18	-0.89	-0.66
acct	-0.67	0.21	-0.52	-0.53	-0.67	0.13	-0.57	-0.46
termutils	-0.35	0.18	-0.21	-0.13	-0.46	0.17	-0.33	-0.20
flex	-0.79	0.07	-0.70	-0.54	-0.88	0.08	-0.78	-0.62
byacc	-0.11	-0.01	-0.08	-0.20	-0.72	0.05	-0.42	-0.40
diffutils	-0.42	-0.02	-0.36	-0.51	-0.66	-0.02	-0.56	-0.57
li	-0.07	-0.17	-0.18	-0.18	-0.09	-0.15	-0.17	-0.18
espresso	-0.55	0.03	-0.33	-0.46	-0.70	0.04	-0.42	-0.43
findutils	-0.25	0.07	-0.20	-0.04	-0.34	0.07	-0.29	-0.01
compress	-0.72	0.04	-0.63	-0.49	-0.89	-0.09	-0.85	-0.63
sudoku	-0.69	0.22	-0.26	-0.40	-0.79	0.20	-0.35	-0.52
barcode	-0.59	0.07	-0.55	-0.65	-0.71	0.06	-0.66	-0.74
indent	-0.64	0.04	-0.45	-0.17	-0.69	0.05	-0.48	-0.16
ed	-0.67	0.03	-0.49	-0.56	-0.82	0.04	-0.59	-0.62
bc	-0.72	0.04	-0.56	-0.57	-0.75	0.05	-0.58	-0.59
copia	-0.72	-0.66	-0.98	-1.00	-0.70	-0.68	-0.98	-1.00
userv	-0.49	0.02	-0.35	-0.40	-0.57	0.04	-0.39	-0.39
ftpd	-0.74	0.03	-0.53	-0.40	-0.78	0.02	-0.57	-0.42
gnuchess	-0.54	0.07	-0.47	-0.31	-0.55	0.06	-0.48	-0.29
go	-0.49	0.03	-0.16	-0.31	-0.58	0.04	-0.18	-0.33
ctags	-0.42	0.03	-0.18	-0.23	-0.53	0.04	-0.23	-0.24
gnubg	-0.66	-0.07	-0.55	-0.68	-0.69	-0.07	-0.57	-0.71
average	-0.36	0.03	-0.25	-0.26	-0.63	-0.01	-0.50	-0.42
strongest	17	0	1	11	23	0	2	4

Table 5.4: Pearson correlation between heuristic metrics and the ENTR and REGX metric. Underlined numbers indicate strongest correlation in the corresponding block.

In the case of ENTR and REGX metrics, in 59% and 79% of the cases NOI showed the strongest correlation; the average correlation was -0.36 and -0.63 (with standard deviations 0.4 and 0.18), respectively. The second best was POI showing strongest correlation in 38% and 14% of the programs with average correlation values -0.26 and -0.42. NII performed poorly, which was surprising because we expected NOI and NII will perform similarly. The promising results for NOI are strengthened by the fact that the highest NOI value predicts a linchpin correctly in most of the cases: in the highly clustered group in 12 out of 13 programs, in the medium group in 7 out of 11 programs the procedure with the highest NOI value turned out to be a linchpin. Another interesting observation we made about the data is that for smaller programs the agreement between the NOI metric and both clusterization metrics was slightly better, suggesting that this heuristic will perform better for smaller programs.

5.6 SEA dependence clusters in Impact Analysis

Impact analysis deals with the problem of identifying those parts (the impact set) of a software system that are affected by a change in the system. The motivation behind the analysis is that developers can concentrate their efforts to the impact set when they want to evaluate the effects of a change. Often the developers are interested in those parts of a program that they have to (re)test when they want to ensure that the modifications did not break existing behavior, or in parts to (re)examine if a change turns out to cause their program to misbehave.

beszedes_242_24

A practical impact analysis method should be as accurate as possible without losing any dependence and sufficiently fast at the same time. Traditional dependence-based methods, such as those based on program slicing, may be accurate but inefficient for real size systems. Using SEA relations for impact analysis is a viable alternative: it is efficient and can be used in practical situations but we need to cope with a certain loss of precision. In this section, we present a set of experiments in which we wanted to verify if (1) the sizes of impact sets computed from SEA dependences are reasonably small, and (2) despite the inaccuracy of the impact sets, they can serve as a basis for determining the effects of a change.

5.6.1 Experiment setup

Figure 5.7 depicts our approach to evaluate the usefulness of SEA dependences for impact analysis applications. It is based on comparing the SEA-based impact sets of fault inducing changes to the modifications made in a software system for fixing these faults. In the figure, the revisions of the system under examination are denoted along horizontal lines. We can examine the differences in subsequent revisions to arrive at a set of procedures that were modified from one revision to the next. We depict these sets between two pairs of revisions, $revision_m$ and $revision_{m+1}$, then later between $revision_n$ and $revision_{n+1}$.

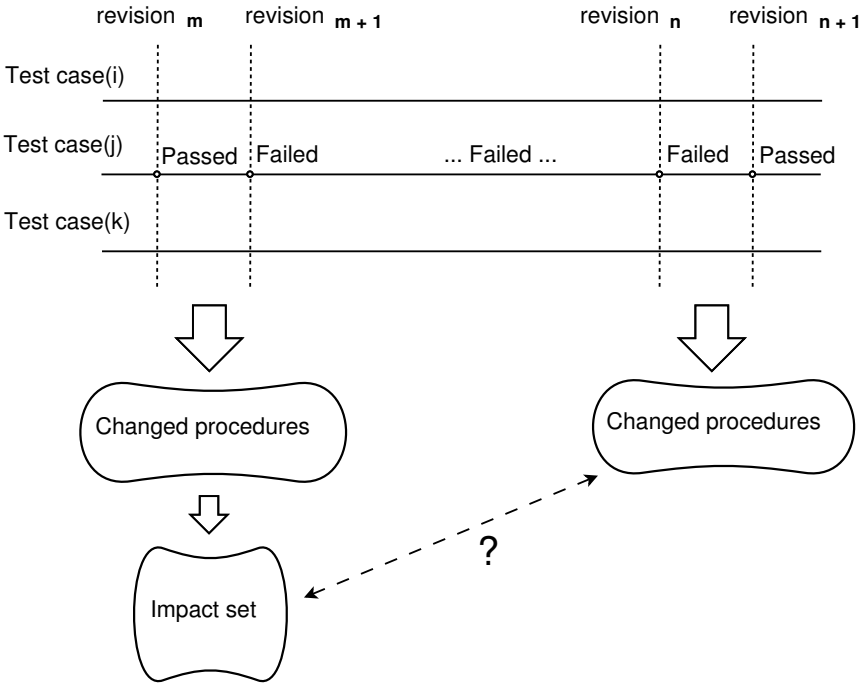


Figure 5.7: Verifying impact analysis prediction capability using real defects

The set of test cases of the system under examination can be seen vertically. All test cases are run on every revision to find out whether any regression errors have been introduced by the latest modifications. The outcome of running a test case can be either *Passed* or *Failed*. Let us consider a scenario in which there is a test case tc_j that produces the following outcomes: *Passed* in $revision_m$, then *Failed* a number of times from $revision_{m+1}$ up to $revision_n$, then *Passed* again in $revision_{n+1}$. In this scenario we can assume that the changes made between $revision_m$ and $revision_{m+1}$ are responsible for the failed test case tc_j. The error that was introduced in $revision_{m+1}$ is worked on by the programmers, then it is corrected in $revision_{n+1}$, when test case tc_j passes again. Our hypothesis is that the impact set of the modified procedures at the time the error was introduced in $revision_{m+1}$ contains the procedures that were modified between $revision_n$ and $revision_{n+1}$.

We evaluated this approach on the WebKit system which was used in previous sections for other experiments as well. The regression test suite of WebKit consists of nearly 25 thousand active test cases; its purpose is to maintain compatibility, standards compliance gains, and check some stability, performance and other issues. WebKit has a large development community geographically spread around the world with very lively development activity. The development environment is a typical one for such a big, distributed open source team which includes serious configuration management and strict integration rules. There is a huge body of version information and historical defect data available in the version control repository, automatic test execution logs and the issue management database. As of January, 2013 there were 140800 revisions, and about 94 revisions were created on average each day, all of which are followed by either a full or selective regression test execution. The issue management database contained nearly 108 thousand entries.

5.6.2 Impact analysis on WebKit

The first step to perform the impact analysis experiments was to determine the potentially interesting defect introducing and defect correcting revision pairs. To arrive at a suitable number of such pairs, we examined a wide range of revisions (r79171–r112713) and used two different methods:

- An automatic method searched through the full range of these revisions and extracted change information from the version control system, as well as examined the execution logs of the regression test suite. The execution logs contain the $Passed/Failed$ status of every test case, so it is possible to identify defect introducing/correcting revision pairs this way. We examined 33542 revisions and found 477 candidate revision pairs.

- We examined the WebKit issue management database, which is based on Bugzilla [128], by manually looking for entries that reported the successful elimination of bugs. Such entries identify a defect correcting revision, then searching backwards from that point, we tried to find the defect introducing revision. There are a lot of uncertainties with this method, primarily because Bugzilla entries are often incomplete or unreliable. Nevertheless, we examined 370 bug entries, from which we managed to identify 275 candidate revision pairs in the same interval as above.

The initial set of revision pairs were filtered in a final step to exclude components of no interest, modules in other language than C/C++, among others. Finally, we combined the results of the two searches and used the 240 identified revision pairs as a basis for determining the prediction capability of the Static Execute After relation. For the final revision pairs we performed ICCFG computation on the first element of each pair (the failure inducing revision). SEA sets were then computed for each changed procedure at these revisions, which resulted in the total of 3792 sets. We computed individual impact sets for the changed procedures in order to be able to compute the prediction ratios for individual procedures, but we also computed a union of these SEA sets to show overall percentages for the revisions.

We needed a series of two measures: the sizes of the impact sets and the corresponding prediction capabilities. Both measures can be expressed in percentage, relative to the program size and the ratio of correctly identified procedures, respectively.

Our analysis framework identified about 92000 procedures (C/C++ function and methods) in the WebKit system (this number varies from revision to revision but stays around this number for the investigated revisions). In the investigated interval approximately 10 procedures per revision were modified on average. This is a relatively low number but it can be justified by the fact that the development process in this system involves frequent modifications.

beszedes_242_24

We present the data about set sizes and prediction on an XY-plot in Figure 5.8. This plot contains 240 data points which correspond to the revision pairs identified in the initial step (the x axis is the prediction, y is SEA set size). Here, the impact sets are determined together for all procedures in a change set by computing their union. This is a more informative presentation of the data compared to looking at individual procedures at the failure revisions, because that way we would not be able to identify which procedure(s) in a change set actually caused the new failure, nor the fixing procedure(s).

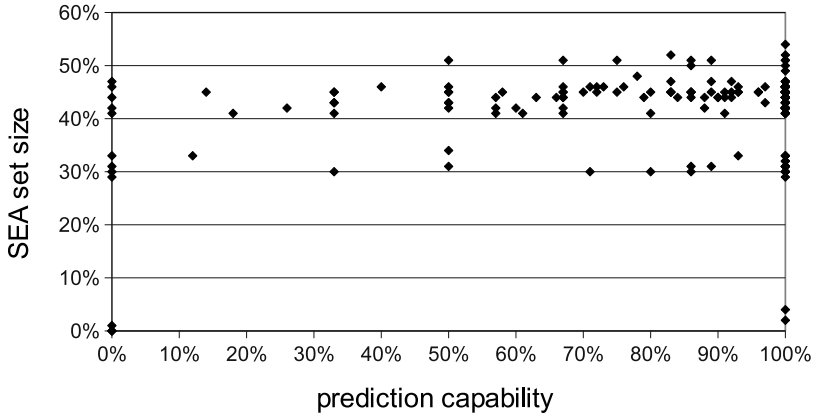


Figure 5.8: Relating prediction capability and impact set sizes for complete change sets.

An expectation about this graph is that it would be beneficial to have many small impact sets with high prediction (towards the lower right corner). If we produced a large number of randomly selected subsets of the procedures in different sizes, theoretically the expected values of this data would be by the diagonal (assuming uniform distribution of the selection and the defects). Therefore anything below the diagonal is good, and it can be determined that the majority of the data points are below the diagonal, so we can conclude that the results are promising as far as prediction capability is concerned.

The prediction numbers vary greatly within the whole range 0-100%; the average is **83.9%** with a deviation of 27.7%. Figure 5.9 shows the distribution for this data with one hundred value ranges. There are several cases where the prediction is 0% or close to it, and we can observe different values at all of the ranges, but in most of the cases the prediction was high, mostly 100%. Low prediction values, even complete misses are expected to occur in a few cases for a number of reasons such as the inaccuracy of static analysis, and deletion or addition of new procedures in the impacts sets, among others.

5.6.3 Dependence clusters in WebKit

We investigated the dependence Clusters in WebKit using the MSG visualization of the subject program based on computing SEA impact sets for all procedures in the system. Figure 5.10 shows this graph for WebKit revision r91555. By visual inspection, we identified three clearly distinguishable clusters that we marked by, from left to right, Cluster_1, Cluster_2 and Cluster_3. Considering the sizes of the SEA-based impact sets, more than half of them were below 1% of the system size, *i.e.* the number of all procedures in the system. The average size of impact sets was 17203, which is just below 19% of the system size.

Apart from the three clusters, we also considered the four regions surrounding the clusters in the MSG. The first two columns of Table 5.5 show these sizes both in the number of impact sets of these regions and in percentage relative to all impact sets. The largest cluster takes

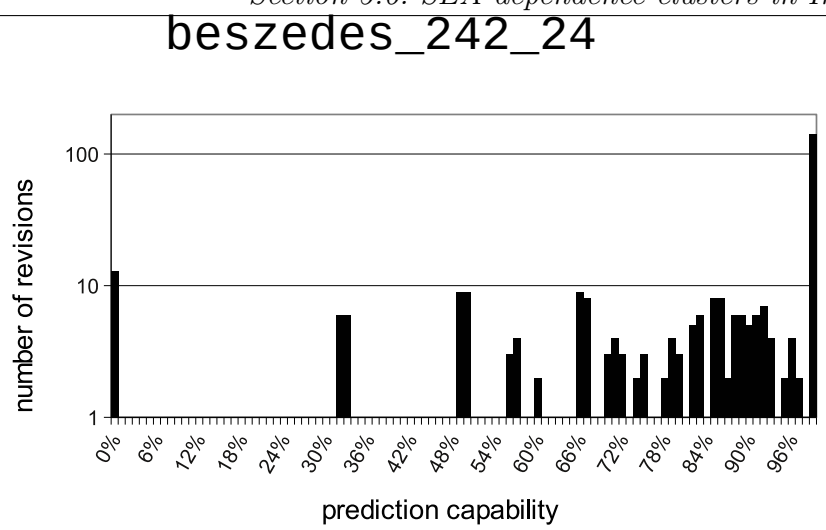


Figure 5.9: Prediction histogram (number of revisions shown on logarithmic scale)

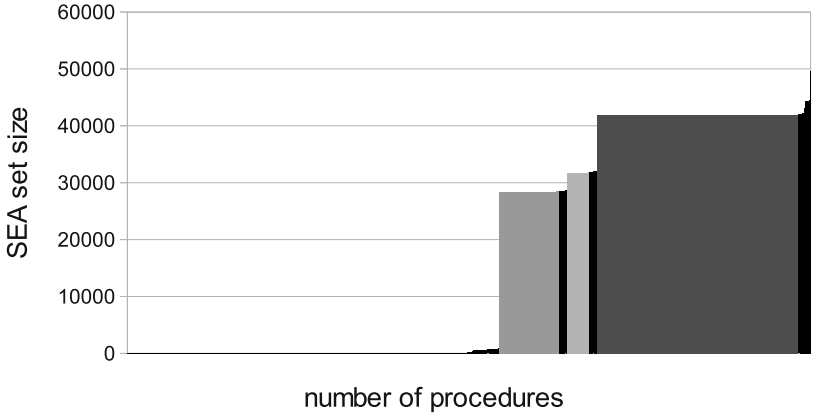


Figure 5.10: MSG graph of SEA impact set sizes. Grey boxes represent the three clusters, black regions are non-cluster sets.

almost 1/3 of all impact sets while the sum of all three is 41.53%, which means that almost half of all impact sets reside in some of the large clusters. The last column of the table shows the impact set sizes for these three clusters. Since there are about 92000 procedures in WebKit, we can conclude that although there are a lot of large impact sets, the largest one contains only about half of the whole program.

The next question we considered was about the distribution of change sets that contain defect-correcting procedures. In particular, we wanted to find out if these occur more frequently in large clusters than in other parts of the system? For this, we filtered the list of all impact sets to those which captured at least one fixing change at the corresponding failure fixing revision. There were 2021 such procedures altogether. The third and fourth data columns of Table 5.5 show how many of these filtered impact sets belonged to the identified regions of the MSG. We can observe that in terms of percentage all clusters contained more impact sets from the filtered set, most notably Cluster_3, which doubled. Altogether, 77.93% of defect predicting impact sets belong to some of the big clusters. This difference to the overall percentage of 41.53% is mostly due to the relatively few small impact sets kept by this filtering (there are more large predicting sets than small ones).

To summarize, almost half of all of the impact sets belong to big clusters while significantly more, over 3/4 of failure predicting impact sets belong to this category.

beszedes_242_24

	All procedures	Percent	Predicted procedures	Percent	SEA size
$Region_1$	49623	54.42%	360	17.81%	–
$Cluster_1$	8028	8.80%	180	8.91%	28395
$Region_2$	1127	1.24%	28	1.39%	–
$Cluster_2$	2981	3.27%	156	7.72%	31720
$Region_3$	1040	1.14%	15	0.74%	–
$Cluster_3$	26864	29.46%	1239	61.31%	41892
$Region_4$	1530	1.68%	43	2.13%	–
Sum	91193	100.00%	2021	100.00%	–

Table 5.5: Dependence cluster sizes in WebKit

5.7 Conclusions

In this chapter, we presented a set of empirical studies related to SEA-based dependence clusters. Our findings towards better understanding of dependence clusters – their formation, detection, analysis and potential elimination – raised a number of additional questions. We think that further research is needed about the connection of dependence clusters and the internal structures of the program, as it would help us understand how we can avoid the formation of clusters, more reliably detect linchpin elements, and design suitable refactoring approaches if eliminating dependence clusters is necessary. Studies involving human evaluation would also be required to gain more insight into the benefits and risks related to dependence clusters in specific applications beyond impact analysis.

Contribution

This chapter is based on the publications:

[10] **Árpád Beszédes**, Lajos Schrettner, Béla Csaba, Tamás Gergely, Judit Jász, and Tibor Gyimóthy. *Empirical Investigation of SEA-Based Dependence Cluster Properties.* Science of Computer Programming, Volume: 105, pages: 3-25, 2015, Publisher: Elsevier B.V. (conference version: [9])

[11] David Binkley, **Árpád Beszédes**, Syed Islam, Judit Jász and Béla Vancsics. *Uncovering Dependence Clusters and Linchpin Functions.* In Proceedings of the 31th IEEE International Conference on Software Maintenance and Evolution (ICSME 2015), pages: 141-150, Bremen, Germany, September 2015.

[25] Lajos Schrettner, Judit Jász, Tamás Gergely, **Árpád Beszédes**, and Tibor Gyimóthy. *Impact Analysis in the Presence of Dependence Clusters Using Static Execute After in WebKit.* Journal of Software: Evolution and Process, Volume: 26, Number: 6, pages: 569-588, 2014, Publisher: John Wiley & Sons, Ltd. (conference version: [24])

These papers received 12 independent citations so far, and contribution [9] was awarded the best paper of the conference.

The concept of the dependence clusters based on Static Execute After relations is mostly my contribution, while the visual and metric-based analysis of dependence cluster properties, the identification of linchpin code elements and the application of SEA sets and dependence clusters in impact analysis are joint work. Some of my other notable papers that have contributed to this result are: [5, 18, 19].

Part III

Spectrum-Based Fault Localization

beszedes_242_24

6

Background on Spectrum-Based Fault Localization

Debugging and related activities are among the most difficult and time-consuming ones in software development [142]. This activity involves human participation to a large degree, and many of its sub-task are difficult to automate. A relevant debugging sub-task is *fault localization* (FL), in which the root causes of an observed failure are sought. Fault localization is notoriously difficult, and any (semi)automated method, which can help the developers and testers in this task, is welcome.

There exists a class of approaches to aid FL which are popular among researchers, but have not yet been widely adopted by the industry: Spectrum-Based Fault Localization (SBFL) [63, 107, 109, 115, 132]. The basic intuition behind SBFL is that code elements (statements, blocks, paths, functions, etc.) exercised by comparably more failing test cases than passing ones are considered as "suspicious" (i.e., likely to contain a fault), while non-suspicious elements are traversed mostly by passing tests.

The concept of *spectrum* in SBFL is a record of a set of program executions, *i.e.* test cases, and their relationship to the code elements. In addition, it relies on pass/fail status of the test case executions. Several types of spectra have been defined over the past decades [76, 132], but the most common approach is to use the so-called "hit-based" spectrum. This refers to the simple binary information if a code element is covered during the execution of a test case or not (also called *coverage-based spectrum*).

Let P denote the program under investigation, T the set of test cases that test P, and E the set of code elements in P according to the chosen granularity level (most often statements or procedures). In the SBFL approach, the dynamic information from running test cases consists of two parts, the *spectrum matrix* M of size $|T| \times |E|$ and the *results vector* R of size $|T|$. Columns of the spectrum matrix represent elements of E while the rows contain elements of T. In the coverage-based spectrum matrix, $m_{i,j} = 1$ if the i-th test covers the j-th element and 0 otherwise. Elements of the results vector R are defined as $r_i = 0$ if the i-th test was completed without failure and 1 otherwise.

The next step in the fault localization process is calculating the four *spectrum metrics* on the matrix [63, 79], which count the number of passing and failing test cases that do or do not include the code element e in question, in various combinations. The following four sets provide the basis for these numbers:

$$
\begin{aligned}
ef(e) &= \{t \in T \mid M(t,e) = 1 \wedge R(t) = 1\} \\
nf(e) &= \{t \in T \mid M(t,e) = 0 \wedge R(t) = 1\} \\
ep(e) &= \{t \in T \mid M(t,e) = 1 \wedge R(t) = 0\} \\
np(e) &= \{t \in T \mid M(t,e) = 0 \wedge R(t) = 0\}
\end{aligned}
$$

For simplicity, in most of the following discussions we will use the notations ef, nf, ep, and np to denote the sizes of these sets, respectively.

Dedicated *risk formulae* are then used to calculate the *suspiciousness* levels by combining the spectrum metrics, which in turn *rank* the code elements to provide a debugging aid to the developer (by convention, a bigger score means a higher rank). When this ranked list is given to the developer for investigation, it is hoped that the fault will be found near the beginning of the list, hence providing useful advice in the debugging process. There is a plethora of different risk formulae proposed by researchers that use hit-based spectrum metrics (good summaries can be found in [79, 104]). Also, some researchers experimented with automatically deriving new formulae [21, 105, 140]. Some of the more important formulae are listed in Table 6.1, which we used in our SBFL-related research.

Barinel [40]: $\frac{ef}{ef+ep}$	*DStar* [131]: $\frac{ef^2}{ep+nf}$
GP13 [140]: $ef \cdot \left(1 + \frac{1}{2 \cdot ep + ef}\right)$	*Jaccard* [39]: $\frac{ef}{ef+nf+ep}$
Naish2 [104]: $ef - \frac{ep}{ep+np+1}$	*Ochiai* [39]: $\frac{ef}{\sqrt{(ef+nf)\cdot(ef+ep)}}$
Russell-Rao [38]: $\frac{ef}{ef+nf+ep+np}$	*Sørensen-Dice* [99]: $\frac{2 \cdot ef}{2 \cdot ef+nf+ep}$
Tarantula [85]: $\frac{\frac{ef}{ef+nf}}{\frac{ef}{ef+nf}+\frac{ep}{ep+np}}$	

Table 6.1: SBFL formulae

There have been several surveys written [42, 107, 130, 132] on fault localization, and various empirical studies performed [38, 109, 147] to compare the effectiveness of various methods. Despite the immense literature, there are still challenges for adopting SBFL in every day practice [12, 20, 38, 94, 122]. Often the faulty element is placed far from the top of the rank-list [108, 135], professionals question the applicability of theoretical results in practice [90, 94], there are little experimental results with real faults [109], and validity issues of empirical research have been raised [122], among others.

In particular, the traditional hit-based methods are generally seen as providing modest performance in terms of ranking precision [89, 108, 135, 136], which contributes to the fact that automated fault localization is still ignored by industry for the most part. Consequently, researchers proposed different approaches that go beyond the hit-based spectrum and utilize other information available that could help improve the overall ranking performance [64, 96, 98, 143, 145]. The presented methods in this part of the dissertation aim at bringing closer the SBFL technique to practical applicability by addressing some of these issues.

7 Reliable Code Coverage Measurement

7.1 Introduction

Code coverage refers to a measurement used in software development and maintenance to analyze the extent to which the source code of a program has been tested. It is a metric that helps assess the effectiveness of testing by indicating the percentage of code elements that have been executed during testing and whether a code element was executed during the test run or not. Code coverage measurement is the basis of several software testing and quality assurance practices including white-box testing [106], test suite reduction [117] and most importantly, for the purposes of our discussion, Spectrum-Based Fault Localization.

Testers have developed both the theory and practice surrounding code coverage measurement, establishing various coverage criteria such as statement and branch coverage [58], along with technical solutions like different instrumentation methods [138]. However, even in straightforward scenarios (e.g., procedure-level analysis of medium size software using popular and stable tools), we observed significant discrepancies in the results produced by different tools applied to the same task. These differences in computed coverages can have serious consequences across various applications, such as generating false confidence in white-box testing and leading to inefficient fault localization, among others.

Various factors might contribute to these differences, and tool developers face specific challenges. In this research, we concentrated on the Java language and the challenges in Java code coverage measurement because Java is frequently used in SBFL-related research. In the Java environment, the most significant challenge we identified is the method of *code instrumentation*. Code instrumentation involves inserting "probes" into the program, which are triggered during runtime to gather essential data on code coverage. There are two primary approaches to instrumentation: at the *source code* level and at the *bytecode* level.

In both instrumentation approaches, the probes which are placed within the system at specific points enable the collection of runtime data but do not alter the behavior of the system. Source code instrumentation means that the original code is modified by inserting the probes, then this version is built and executed during testing. The second method instruments the compiled version of the system, the bytecode. Here, two further approaches exist. First, the probes may be inserted right after the build, which effectively produces modified versions of the bytecode files (called *offline* bytecode instrumentation). Second, the instrumentation may take place during runtime upon loading a class for execution (*online*

bytecode instrumentation). One benefit of bytecode instrumentation is that it does not require the source code, thus it can be used *e.g.* on third party code as well. On the other hand, it is dependent on the bytecode version and the Java VM, thus it is not as universal as source code instrumentation. Implementing bytecode instrumentation is usually easier than inserting proper and syntactically correct probes into the source code. However, source code instrumentation also allows full control over what is instrumented, and in certain applications, including SBFL, it is a better choice due to direct connection to the source code, contrary to bytecode-based approach which is sometimes less accurate in this respect [138].

In this chapter, we present an evaluation method for assessing code coverage measurement tools, and a corresponding empirical study that compares the code coverage results generated by two popular tools for Java, one representative for each instrumentation approach. We evaluated the tools on 8 open source Java projects, and systematically investigated the differences both quantitatively (how much the outputs differ) and qualitatively (what the causes for the differences are). We also investigate the impact of discrepancies on possible applications, among others Spectrum-Based Fault Localization.

We concentrated on *procedure-level* measurements, meaning that the primary focus of coverage information is on whether a specific Java method is invoked by the tests, without considering which statements or branches within the method are executed. In many practical situations, coverage analysis is conducted hierarchically, beginning with higher-level code components like classes and methods. If the coverage results are inaccurate at this level, they will also be inaccurate at lower levels. Across different applications, unreliable results at the method level are likely to lead to similar or even worse outcomes at the statement or branch level. Previous research has demonstrated significant discrepancies between bytecode and source code coverage measurements at the statement and branch levels [95], and notable differences in overall coverage values at the method and branch levels [46].

7.2 Evaluation method

We conducted an empirical study on eight open source systems with code coverage tools for Java employing both instrumentation approaches. Apart from the coverage measurement tools, our measurement framework consisted of some additional utility tools. The main tool we relied on was the SoDA framework [29, 120]. For the representation of the coverage data in SoDA, the data generated in different forms by the coverage tools were converted into the common SoDA representation, the coverage matrix. Later, this representation was used to perform the additional analyses. This framework also contains tools to calculate statistics, produce graphical results, etc. Apart from this, only general helping shell scripts and spreadsheet editors were used.

7.2.1 Benchmark programs

To establish our benchmark programs, we adhered to the following criteria. We aimed to compare bytecode and source code instrumentation, so the source code had to be accessible. Consequently, we selected open-source projects, which also facilitates the replication of our experiments. We chose projects that could be compiled with Maven, as this framework allows for straightforward integration of code coverage measurement tools. Finally, it was essential that the subject programs included a practical set of realistic test cases based on the JUnit framework [86] (preferably version 4).

We looked for potential projects on GitHub [69], favoring those that had been utilized in prior studies. We identified eight subject programs from various domains, each with a

substantial size (see Table 7.1). These systems exhibit varying proportions of tests and overall coverage, adding diversity to our benchmark.

Program	version	LOC	Methods	All tests	Domain
checkstyle	6.11.1	114K	2 655	1 589	static analysis
commons-lang	#00fafe77	69K	2 796	3 683	java library
commons-math	#2aa4681c	177K	7 167	5 842	java library
joda-time	2.9	85K	3 898	4 177	java library
mapdb	1.0.8	53K	1 608	1 786	database
netty	4.0.29	140K	8 230	4 066	networking
orientdb	2.0.10	229K	13 118	950	database
oryx	1.1.0	31K	1 562	208	mach. learning

Table 7.1: Subject programs. Metrics were calculated from the source code (generated code was excluded).

7.2.2 Selection of coverage tools

Our goal in this research was to compare the code coverage results generated by tools utilizing the two different instrumentation approaches. To achieve this, we aimed to ensure that the tools chosen for the analysis accurately represent the instrumentation methods and that our results are less affected by tool-specific characteristics. The list of candidate tools identified for our study is shown in Table 7.2.

Tool	Approach	Supported Java/JRE version	Active	Licence
Clover	source	1.3+	present	commercial/free
Cobertura	bytecode	1.5–1.7	2015	free
JaCoCo	bytecode	1.5+	present	free
Jcov	bytecode	1.0+	present	free
SD Test Coverage tools	source	1.1+	present	commercial

Table 7.2: Tools for Java code coverage measurement

Among from the source code instrumentation-based tools we selected Clover by Attlassian [61] (version 4.0.6) to be used in later parts of the experiments. It supports Java 8 constructs, integrates seamlessly with the Maven build system, and can measure coverage on a per-test basis. We selected Clover for our detailed bytecode-source code measurements due to its superior Maven integration and per-test coverage support, which facilitated its incorporation into our experiments. To establish the source code instrumentation results as a baseline for our experiments, we manually verified Clover's results through selective manual instrumentation. We chose a subset of methods, up to 300 per subject system, and manually instrumented these methods before running the test suite. We then analyzed the results in terms of actual test executions and program behavior at the source code level. Upon review, we found no discrepancies between the methods covered as reported by the manual instrumentation and Clover.

From the three candidate tools in the bytecode-based instrumentation category, we chose JaCoCo [81] (version 0.7.5.201505241946) due to its popularity, greater visibility and easier integration compared to the other options. This free Java code coverage library, developed by

the EclEmma team, integrates seamlessly with Maven-based build systems. JaCoCo has plug-ins for most of the popular IDEs (*e.g.* NetBeans), for CI- and build systems (*e.g.* Jenkins) and also for quality assessment tools (*e.g.* SonarQube). These plug-ins have about 20k installations/downloads per month in total. JaCoCo has up-to-date releases and an active community.

7.2.3 Measurement process

To compare the code coverage results and examine the differences in detail, we needed to calculate the coverages using different settings and variations of the tools. We wanted to make sure that we could gather *per-test case* and *per-method* coverage results from the tools as well (*i.e.* which test cases covered each method and the opposite), which is essential for assessing some applications including SBFL.

Clover could be easily integrated in the Maven build process, and we had no issues obtaining the per-test case coverage data we needed. JaCoCo could also be incorporated into a Maven-based build system; however, it initially lacked the ability to measure coverage for individual test cases. To address this, we developed a custom listener to capture this information. We then configured each program's test execution environment to interact with this listener, allowing us to identify the start and end of each test case's execution. The two kinds of JaCoCo measurements may produce differences in the results, but these do not affected most of our further experiments.

We needed to address another technical detail related to the Clover tool. Specifically, when dealing with multiple modules in projects, we had two options: either integrate the measurement at a global level for the entire project, or configure it individually for each sub-module. Since JaCoCo adopts the latter approach, we decided to configure Clover individually for the sub-modules as well. In the following, we will make note when these variations to the tool configuration influence the results of the experiments.

Using the per-test case coverage data, we generated a *coverage matrix* for each program, which is compatible with the spectrum matrix used in Spectrum-Based Fault Localization. From this matrix, we were able to easily compute various coverage statistics, including per-test case and per-method coverage.

7.3 Results

In this section, we present the results related to code coverage differences between the two tools. As the source code-based instrumentation is more suitable for applications involving the source code, including SBFL, we will treat Clover results as the ground truth and reference point, and JaCoCo results will be compared to it.

7.3.1 Quantitative analysis

We compared the overall method-level coverage values obtained for our subject programs (see Table 7.3). The results for JaCoCo and Clover are shown for each program, along with the difference in coverage percentages. Coverage ratios are expressed as the percentage of methods covered, out of the total methods identified by the corresponding tool.

Excluding the outlier program checkstyle, the differences between the tools range in a relatively small interval, from -1.42% to 0.76%. In the following sections, we seek for the reasons of the differences, and we will explain the outlier as well (in Section 7.3.2).

While Table 7.3 presents the overall coverage values generated by the entire test suite, the coverage ratios achieved by individual test cases may reveal a different set of specific

beszedes_242_24

Program	JaCoCo	Clover	difference
checkstyle	53.77%	93.82%	-40.05%
commons-lang	92.92%	93.28%	-0.36%
commons-math	84.92%	84.65%	+0.27%
joda-time	89.52%	89.94%	-0.42%
mapdb	74.64%	76.06%	-1.42%
netty	40.92%	40.18%	+0.74%
orientdb	27.01%	28.01%	-1.00%
oryx	29.51%	28.75%	+0.76%
average	**61.65%**	**66.84%**	**-5.19%**

Table 7.3: Overall coverage values for the unmodified tools

variations. Table 7.4 provides statistics on the coverage for individual test cases for both JaCoCo and Clover. This includes minimum, maximum, and average values (representing the proportion of covered methods to total methods) as well as the difference in average values (with positive values indicating higher average coverage for Clover). It is notable that checkstyle exhibits a high global difference between Clover and JaCoCo in the per-test case results as well, though this difference is less pronounced than in the global case. Interestingly, for mapdb, netty, and oryx, the average individual differences between Clover and JaCoCo are opposite in sign to the global differences.

Program	JaCoCo results			Clover results			avg diff
	min	max	avg	min	max	avg	
checkstyle	0.00%	15.87%	3.02%	0.00%	30.13%	4.66%	+1.64%
commons-lang	0.00%	3.10%	0.61%	0.00%	2.93%	0.64%	+0.03%
commons-math	0.00%	4.34%	0.47%	0.00%	4.34%	0.47%	0.00%
joda-time	0.00%	8.84%	1.62%	0.00%	9.93%	1.64%	+0.02%
mapdb	0.00%	20.86%	7.67%	0.00%	22.08%	7.19%	-0.48%
netty	0.00%	3.43%	0.32%	0.00%	3.69%	0.37%	+0.05%
orientdb	0.00%	9.40%	0.58%	0.00%	9.88%	0.62%	+0.04%
oryx	0.00%	2.05%	0.45%	0.00%	1.79%	0.48%	+0.03%

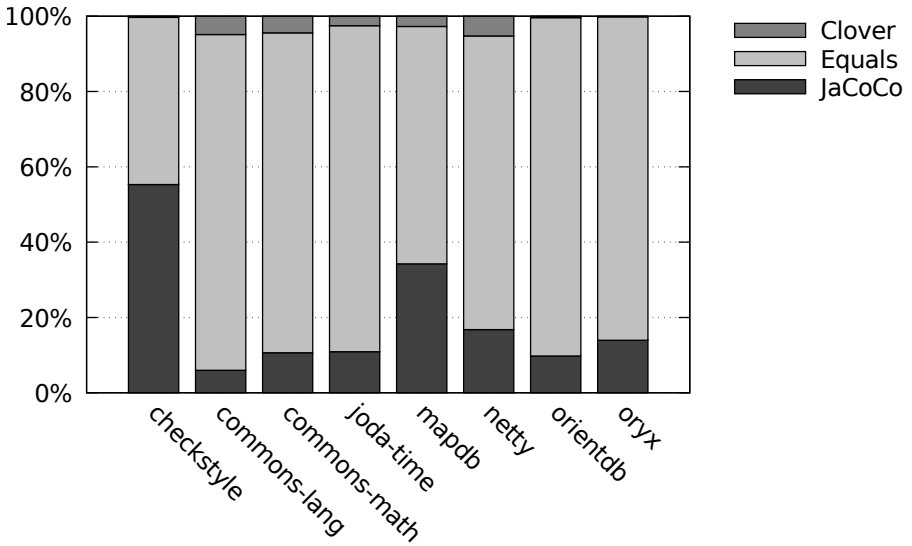
Table 7.4: Per-test case coverages

It is not obvious how individual coverage differences imply global coverage difference and vice versa. Significant variations in individual coverages might not affect the overall coverage. For instance, a method is considered covered if even a single test case covers it. So, if one instrumentation technique identifies a hundred covering test cases, while another identifies only one, the overall coverage remains unchanged, though the individual coverages differ. Conversely, small differences at the individual level can lead to a significant overall difference. If many test cases each have one method that is reported differently, and these methods are exclusively covered by those test cases, the minor individual differences can accumulate, resulting in a substantial global coverage difference.

When we compared the number of covering test cases per method, we observed three types of differences. First, JaCoCo and Clover identified different sets of methods, the reasons for which will be explained in Section 7.3.2. Second, in some instances where both approaches recognized the same methods, Clover reported at least one covering test case, while JaCoCo did not, and vice versa. The third type of difference occurred when both tools indicated

beszedes_242_24

that a method was covered, but by a differing number of test cases. Figure 7.1 illustrates the corresponding results. Specifically, it shows the percentage of methods for each program (relative to the total number of methods recognized by either tool) under the following conditions: there is no difference in the covering sets of test cases, or either Clover or JaCoCo reports more covering test cases.



Clover: Clover reports more covering TCs; Equals: both tools report the same covering TCs; JaCoCo: JaCoCo reports more covering TCs.

Figure 7.1: Summary of differences in the per-method coverage

As can be seen, the tools do not completely agree on the coverage. First, many methods are not detected by the Clover tool, which can be attributed to various factors, mainly related to the generated code. A significant outlier is checkstyle, with 55% of methods falling into this category, while the others are below 15%. Next, as shown in Table 7.5, there are only a few methods where Clover and JaCoCo disagree on whether a method is covered by at least one test case, while both tools recognize the method (as indicated by the *Czero* and *Jzero* columns). We manually examined all 220 of these methods to determine the causes of the discrepancies (see Section 7.3.2). The other two columns report cases where the number of covering test cases differed. The *CltJ* column indicates cases where "Clover reports fewer than JaCoCo", while the *JltC* column indicates cases where "JaCoCo reports fewer than Clover". A significant portion of methods in the subjects were affected by this discrepancy to some degree (nearly 30% for mapdb and over 11% for joda-time, for example).

Program	Czero	CltJ	JltC	Jzero
checkstyle	1	9	16	0
commons-lang	0	21	131	5
commons-math	19	297	239	7
joda-time	0	358	86	2
mapdb	7	450	25	2
netty	91	300	466	76
orientdb	1	104	32	5
oryx	4	8	1	0

Table 7.5: Differences in per-method coverages of code elements of JaCoCo and Clover

In summary, the detailed measurements for both test case and method centered data can differ significantly from the overall coverage ratios. While there are instances where the overall ratios align with the detailed data, this is not always the case. A large overall discrepancy may be due to small variations at a detailed level, or vice versa. Therefore, observing inaccuracies at the detailed level is required in order to be able to assess the impacts on various applications and, in particular, on Spectrum-Based Fault Localization.

7.3.2 Qualitative analysis

In this section, we explore the potential causes of the differences noted in the previous section. We manually inspected and analyzed the discrepancies between the coverage results reported by JaCoCo and Clover. We selectively investigated representative cases to ensure that each system and module was adequately covered. We also focused on the most problematic cases highlighted in columns *Czero* and *Jzero* of Table 7.5.

Our investigation included both the original and instrumented versions of the source code and bytecode, as well as other artifacts such as build configuration files to uncover additional contributing factors. In total, we manually examined several hundred individual methods and test cases, and identified several common causes of the discrepancies, which we will summarize in the following.

Cross-submodule coverage. In projects with multiple sub-modules, Clover and JaCoCo handle coverage reporting differently. Clover instruments all source code, allowing it to report cross-module coverage, while JaCoCo only instruments the currently tested module, missing coverage from other modules. JaCoCo would only consider methods covered if they are tested directly within their own module, whereas Clover would aggregate coverage across all modules. This distinction can result in varying coverage reports. The examples provided include both multi-module projects like netty, orientdb oryx, and single-module projects.

Untested sub-modules. In the case of JaCoCo, if a module does not have any tests its methods will not be recognized. Consequently, the methods of sub-modules will not be recognized and they will be missed from the set of all methods of the project. Clover, on the other hand, correctly determines the set of all methods across all sub-modules.

Test case preparation and cleanup. In testing frameworks, setup and teardown methods prepare or clean up before and after tests, often marked with annotations like `@Before` and `@After`. In JaCoCo , these methods are counted as part of the test cases and reported as covered. Conversely, Clover does not include setup and teardown methods as part of the test cases, so coverage of these methods is not attributed to any specific test case.

Recognized method sets. There is a discrepancy between the results from JaCoCo and Clover regarding the method sets they are working with, as already indicated in Figure 7.1. This discrepancy arises because the sets of methods identified from the source code and the bytecode can differ. We observed that several methods are identified only by Clover or JaCoCo. The second group is not really surprising because we expected in advance a relatively large number of generated methods in the bytecode (due to the necessary mechanisms of the Java language). However, we were somewhat surprised to find that some methods were recognized solely by Clover. The causes of the differing method sets include, among others: (1) different handling of test methods, (2) compiler generated code which is impossible to be included by a source code instrumentation approach and (3) inclusion or omission of generated code.

Instrumentation. We discovered that in some instances, the instrumentation itself modified the behavior of the tests, potentially affecting the list of executed methods. For example, in the joda-time program, two specific test cases failed after being instrumented by Clover. This occurred because the tests use Java reflection to determine the number of subclasses of the tested class. Since Clover implements coverage measurement and test case detection by adding subclasses to the examined class, these two tests failed on assertions at the very beginning of the test case. A similar issue arose in the checkstyle project, where two test cases verify that the classes being tested contain a fixed number of fields. However, due to the additional fields inserted by Clover in these classes, the assertions failed.

Exception handling during coverage measurement. When JaCoCo instruments bytecode, it places probes at key points by analyzing the control flow of all methods within a class. If the control flow is interrupted by an exception between two probes, JaCoCo will not consider the instructions between the probes to be covered. This happens because if a method throws an exception early in the caller method, JaCoCo marks the caller method as not covered since it misses the instrumentation probe at the method's exit point. In contrast, Clover's instrumentation strategy can handle this scenario, marking the caller method as covered by considering the probe at the method's entry point. Additionally, JaCoCo tends to report lower coverage for tests expected to throw exceptions (*i.e.* annotated with `@Test(expected=SomeException)`), which is related to its exception handling approach and is a known limitation of JaCoCo.

Name encoding. A common cause of the discrepancies involved enums, anonymous classes, and nested classes. The issue arises because, in some instances, when these classes are compiled into bytecode, a method may receive additional parameters to access members of its enclosing class. In other cases, the methods might lose some of their original parameters from the source code. As a result, the signatures of the same method in the source code and bytecode can differ. These missing or additional parameters in the bytecode lead to differing method signatures between JaCoCo and Clover measurements. This discrepancy prevented the automatic mapping of methods between the two measurements, resulting in lower JaCoCo coverage counts in our experiments.

Other. We also identified several occasional reasons for the deviations. The first was the differing handling of certain built-in methods in the `Object` class (such as `equals`, `finalize`, or `hashcode`). When these methods were redefined across multiple levels of inheritance, both tools sometimes produced incorrect results for these methods. This discrepancy could lead to both JaCoCo and Clover reporting lower coverage for the same project. Another issue was that Clover sometimes struggled to detect test cases that were invoked from within other test cases (as seen, for example, in the class c in commons-math), resulting in incorrect coverage elements being reported. While it is possible to avoid calling test cases from within other test cases (even transitively), if this does occur, the resulting detailed coverage data might be unreliable. However, this issue does not affect the overall coverage of the test suite, as the coverage will still be recorded, just at a different point in the program.

7.4 Impact on software maintenance applications

In this section, we elaborate on the possible implications of the inaccuracies of different code coverage measurement tools. We collected some of the most important applications of code

coverage measurement, which are summarized in Table 7.6. Note, that this analysis is not about deciding if a source code instrumentation-based approach such as Clover or a bytecode one such as JaCoCo is better since both tools can produce false results in both directions.

Application	*Falsely not covering*	*Falsely covering*
White-box testing, quality, traceability	Increased effort	False confidence
Fault localization	Impossible localizability	Moderate impact on scores
Test selection and prioritization	Suboptimal priority	Suboptimal priority
Test case generation	Inability to check success	Reduction in success
Mutation analysis	Minor	Inability to kill mutants

Table 7.6: Summary of the impacts of inaccurate code coverage

If we compare the different cases we can observe that the level of impact is typically not the same for falsely covering and falsely not covering. In the following, we elaborate on the different applications in more detail.

White-box testing, quality, traceability. White-box testing may suffer from coverage inaccuracies in two ways. The falsely covering case is a more serious one because it may give a false confidence in the completeness of the testing. Fortunately, our results show that bytecode instrumentation rarely results in this kind of error. The other case is much more frequent, and falsely not covering program elements will usually result in more effort required to action on the coverage results. Namely, it will mean more program elements to investigate during testing.

Fault localization. Spectrum-Based Fault Localization fundamentally relies on code coverage. An imprecision in the spectrum matrix will impact the suspiciousness score and hence the chances of localizing the fault. In particular, if the fault is in the program element which is erroneously reported as not covered it will never be localized using the standard algorithms (most scores such as Tarantula [85] will be set to 0 in this case). Even in the case when the coverage is not totally missing but there are fewer covering test cases reported, it will decrease the chances for fault localization. The falsely covering case will also impact the localization scores, though moderately.

Test selection and prioritization. Test selection and prioritization methods that rely on code coverage [33, 72] may also be severely impacted by inaccuracies in the coverage. Algorithms that give preference to highly covering test cases basically prioritize them either globally according to the coverage ratio or to how much additional coverage a test case provides [139]. Test selection methods then select the first given number of test cases from this list. This means that any difference in the per-test case coverage will have a high impact on the performance of the algorithms. This problem affects both the falsely not covering and falsely covering cases.

Test case generation. The impact of inaccurate code coverage on test case generation algorithms [66, 111] is similar to the impact on general white-box test design, except that in

this case there is no human involved who can understand (with increased effort) that there is a missing or superfluous coverage. Consequently, a missing coverage will result in the algorithm not being able to check the success of a generated test case because it will always observe that it failed to cover a program element. A superfluous coverage on the other hand, will reduce the successfulness of the generation.

Mutation analysis. In mutation analysis [83, 125], the program is modified to insert artificial faults (creating mutants) and verify if any of the test cases can detect the fault (killing the mutant). If the mutant is not killed then a new test case is created or generated to kill it. Here, the falsely covering case will have the effect that the algorithm will be unable to kill the mutant because, due to actual non-coverage, the fault will not be detected. A falsely not covering case will have a minor impact because the fault will be detected regardless of code coverage information, but it will still be confusing. Also, mutation testing will suffer from the same difficulties as with test case generation since it uses this technique to augment the test cases in order to kill the mutants.

7.5 Conclusions

Results presented in this chapter indicate that significant differences may occur between the bytecode and the source code instrumentation coverage approaches for Java. Some of the differences can be eliminated, but some cannot or their elimination would not be practical. The kind and level of influence of these differences on various applications is difficult to predict as it depends on the subject program and the application itself. The list of possible reasons we identified for the differences may be used as a guideline on how to avoid and workaround the inaccuracies of the tools. Despite its disadvantages, source code-based instrumentation proved to be beneficial both in terms of coverage accuracy and superiority in some applications, including Spectrum-Based Fault Localization.

Contribution

This chapter is based on the publications:

[17] Ferenc Horváth, Tamás Gergely, **Árpád Beszédes**, Dávid Tengeri, Gergő Balogh, and Tibor Gyimóthy. *Code Coverage Differences of Java Bytecode and Source Code Instrumentation Tools.* Software Quality Journal, Volume: 27, Number: 1, pages: 79-123, 2019, publisher: Springer.

[30] Dávid Tengeri, Ferenc Horváth, **Árpád Beszédes**, Tamás Gergely, and Tibor Gyimóthy. *Negative Effects of Bytecode Instrumentation on Java Source Code Coverage.* In Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016), pages: 225-235, Osaka, Japan, September 2016.

These papers received 24 independent citations so far, mostly from the application side of code coverage measurement, which demonstrates the usefulness of our results.

The evaluation process with which the different code coverage measurement tools can be objectively assessed is mostly my contribution, while the execution of the comparison of actual tools on a benchmark, the quantitative and qualitative evaluation of the differences, and the impact of code coverage inaccuracies in different applications are joint work.

8

Use of Call Chains in Spectrum-Based Fault Localization

8.1 Introduction

Researchers proposed many different Spectrum-Based Fault Localization (SBFL) formulae and scoring mechanisms, but these are essentially all based on counts of passing/failing and traversing/non-traversing test cases in different combinations (see Chapter 6). Based on the vast amount of research performed in the field, it seems that variations to these basic approaches may yield only marginal improvements, and that other approaches to the problem are required in order to achieve more significant gains. For example, by combining conceptually different techniques [147], or by involving additional information to the process.

This additional information can either be feedback from the user [16] or should go beyond the simple hit-based spectrum on basic code elements. This can then serve as some kind of a *context* for the suspicious elements. Early attempts to incorporate control or data flow information, for instance [76, 115], have not been further developed because it soon became apparent that they are difficult to scale to large programs and real defects.

One specific reason why an SBFL formula may fail is what is referred as *coincidental correctness* [50, 102, 126]. This is the situation when a test case traverses a faulty element without failing. This can happen often since not all exercised elements may have an impact on the computation performed by a test case [103], and if there are more such cases than traversing and failing ones, the suspiciousness score will be negatively affected [102].

Motivated by the need for adding contextual information to the process, and specifically addressing the issue of coincidental correctness, we propose to enhance traditional SBFL with *function call chains* on which the FL is performed (in the following, we will use the term 'function' to refer to any kind of procedure, *i.e.* also method for object-oriented languages like Java). Function call chains are snapshots of the call stack occurring during execution and as such can provide valuable context to the fault being traced. Call chains (and call stack traces) are artifacts occurring during program execution which are well-known to programmers who perform debugging, and can show, for instance, that a function may fail if called from one place and perform successfully when called from another. There is empirical evidence that stack traces help developers fix bugs [118], and Zou *et al.* [147] showed that stack traces can be used to locate *crash-faults*.

More specifically, in this chapter we describe a novel SBFL algorithm that computes ranking on all occurring call chains during execution, and then selects the suspicious functions from these ranked chains using a function-level spectrum-based algorithm, Ochiai in particular [38]. An example of the overall approach is presented in Section 8.2.

Our approach works at a lower granularity than statement-level approaches (previous work suggests that function-level is a suitable granularity for the users [49, 147]). At the same time, we provide more context in the form of the call chains, and therefore have the potential to show better fault localization performance.

8.2 Fault localization on call chains

Figure 8.1 provides a high-level overview of our approach. Using a given set of test cases T, the subject program P is executed while collecting the necessary execution trace information. This is used to produce the function call chains, as well as the test case pass/fail outcomes (more on this in Section 8.2.1). Based on that, we compute the call chain level program spectrum information, which is used to calculate the ranking of the chains according to their suspiciousness levels (discussed in more detail in Section 8.2.2). In the next step, two algorithms are applied to compute the ranking of the functions for FL, which are then merged to produce the final ranking (presented in Section 8.2.3).

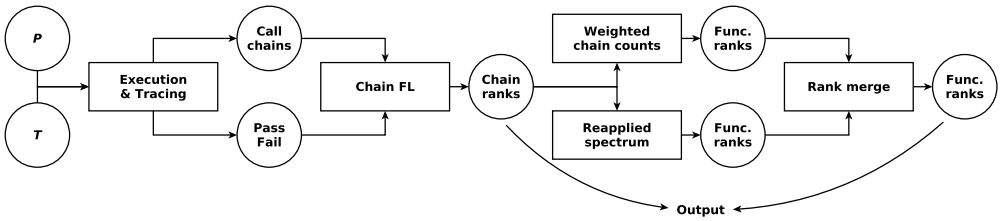


Figure 8.1: Call chain-based SBFL

8.2.1 Function call chains

Let F be the set of functions in a program P, and T a set of test cases used to test P. Then, a *call chain* c is a sequence of functions $f_1 \to f_2 \to \cdots \to f_n$ ($f_i \in F$), which occur during the execution of some test case $t \in T$, and for which:

- f_1 is the entry point called by t,
- each f_i directly calls f_{i+1} ($0 < i < n$), and
- f_n returns without calling further functions in that sequence.

In other words, c is one of the possible deepest *call stack* states occurring during the execution of t. Call stacks and the associated stack traces are well-known structures used in everyday work by programmers during debugging. They describe a particular state during program execution and help understand the context that led to that state. At the same time, they are very concise as well because no previous state is maintained, and typically the function call nesting levels are not very deep. Statement-level control or data flow information is much more complex and more difficult to produce. Call chains can be efficiently produced from test case executions because only the function entry and exit events need to be recorded and stored in a stack structure.

In our method, we collect all distinct call chains occurring during the execution of T, which will be referred to as the call chain set C. We also maintain a set of chains $C(t)$ occurring for each individual test case t (we say that t *executes* c if $c \in C(t)$). Finally, the set of functions occurring in a chain c will be denoted by $F(c)$.

beszedes_242_24

Figure 8.2a contains a simple code snippet for illustrating these concepts with the associated test cases in Fig. 8.2b. In it, we can identify four test cases `t1`...`t4`. `t1` and `t2` are passing, while the other two fail due to an error in function `g`. These test cases produce altogether five different call chains: $\texttt{a} \rightarrow \texttt{f}$, $\texttt{a} \rightarrow \texttt{g}$, $\texttt{b} \rightarrow \texttt{a} \rightarrow \texttt{f}$, $\texttt{b} \rightarrow \texttt{g}$ and `b`, which will constitute the set C. Then, $C(\texttt{t1}) = \{\texttt{a} \rightarrow \texttt{f}, \texttt{a} \rightarrow \texttt{g}, \texttt{b} \rightarrow \texttt{g}\}$, $C(\texttt{t2}) = \{\texttt{a} \rightarrow \texttt{g}, \texttt{b} \rightarrow \texttt{g}\}$, $C(\texttt{t3}) = \{\texttt{a} \rightarrow \texttt{g}, \texttt{b}\}$ and $C(\texttt{t4}) = \{\texttt{a} \rightarrow \texttt{f}, \texttt{a} \rightarrow \texttt{g}, \texttt{b} \rightarrow \texttt{a} \rightarrow \texttt{f}\}$.

This example is constructed so that the benefits of our method are visible. In particular, we set the fault to be in function `g`, and it will be manifested if invoked directly from `b` but not when invoked from `a`. This way, both elements, the caller and the callee, are important from the localization point of view, and this is what the call chains will capture. As we will see, the fault will be located at the first ranking position with our approach, while a function-level hit-based suspiciousness score (*e.g.* Ochiai) will give priority to some other code element. The call chain information is also useful because it will allow the programmer to find other possible fixes for the failure such as modifying the call site if that is more appropriate. A more realistic example is provided in Section 8.5, which is an actual fault from our benchmark.

8.2.2 Chain-based SBFL

The first phase of our approach is fault localization on the call chains. This takes as inputs the test case execution outcomes (pass/fail) and uses a program spectrum representation with the chains as code elements. The output is a ranked list of call chains with the associated suspiciousness scores.

```
public class ChainFLExample {
  private int _x = 0;
  private int _s = 0;
  public int x() {return _x;}

  public void a(int i) {
    _s = 0;
    if (i==0) return;
    if (i<0)
      f(i);
    else
      g(i);
  }

  public void b(int i) {
    _s = 1;
    if (i==0) return;
    if (i<0)
      a(i);
    else
      g(i);
  }

  private void f(int i) {
    _x -= i;
  }

  private void g(int i) {
    _x += (i+_s); // error: should be _x += i;
  }
}
```

(a) Example for illustrating call chains.

```
public class ChainFLExampleTest {
  @Test public void t1() {
    ChainFLExample tester = new ChainFLExample();
    tester.a(-1);
    tester.a(1);
    tester.b(1);
    assertEquals(3, tester.x());
  }

  @Test public void t2() {
    ChainFLExample tester = new ChainFLExample();
    tester.a(1);
    tester.b(1);
    assertEquals(2, tester.x());
  }

  @Test public void t3() {
    ChainFLExample tester = new ChainFLExample();
    tester.a(1);
    tester.b(0);
    assertEquals(1, tester.x());
  }

  @Test public void t4() {
    ChainFLExample tester = new ChainFLExample();
    tester.a(-1);
    tester.a(1);
    tester.b(-1);
    assertEquals(3, tester.x());
  }
}
```

(b) Test cases for the example.

Figure 8.2: Function call chain running example

We apply a traditional program spectrum representation based on binary matrices (see Chapter 6). Let $\mathbf{S}^{ch}$ denote the chain based spectrum, whose rows represent test cases (elements of T), and columns contain the call chains (elements of C):

beszedes_242_24

$$\mathbf{S}^{ch} = \begin{matrix} \\ t_i \end{matrix} \overbrace{\begin{matrix} \\ \end{matrix}}^{c_j} \begin{Bmatrix} \begin{bmatrix} 0/1 & 0/1 & \cdots & 0/1 & 0/1 \\ 0/1 & 0/1 & \cdots & 0/1 & 0/1 \\ & & \ddots & & \\ 0/1 & 0/1 & \cdots & 0/1 & 0/1 \\ 0/1 & 0/1 & \cdots & 0/1 & 0/1 \end{bmatrix} \end{Bmatrix} \quad \mathbf{R}^{ch} = \begin{bmatrix} 0/1 \\ 0/1 \\ \vdots \\ 0/1 \\ 0/1 \end{bmatrix},$$

where $\mathbf{S}^{ch}(i,j) = 1$ means that the call chain c_j will occur at least once in the execution of test case t_i, and vector $\mathbf{R}^{ch}$ denotes the test case execution results vector. It is a record of the outcomes of test case runs, namely *pass* (0) or *fail* (1). Figure 8.3 shows the spectrum with the matrix and the result vector for our example from Figures 8.2a and 8.2b ($c_1 \dots c_5$ denote the chains `a` $\rightarrow$ `f`, `a` $\rightarrow$ `g`, `b` $\rightarrow$ `a` $\rightarrow$ `f`, `b` $\rightarrow$ `g` and `b`, respectively).

$$\mathbf{S}^{ch} = \begin{bmatrix} & c_1 & c_2 & c_3 & c_4 & c_5 \\ \texttt{t1} & 1 & 1 & 0 & 1 & 0 \\ \texttt{t2} & 0 & 1 & 0 & 1 & 0 \\ \texttt{t3} & 0 & 1 & 0 & 0 & 1 \\ \texttt{t4} & 1 & 1 & 1 & 0 & 0 \end{bmatrix}, \quad \mathbf{R}^{ch} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

Figure 8.3: Chain-based spectrum for the example

For the call chains, any basic SBFL suspiciousness score could be used. In this work we used the Ochiai score [39] which is one of the more popular formulae and is proven to outperform other popular formulae in many situations [49, 109, 147]. In the following, we will refer to the Ochiai score of a call chain c as $\mathcal{O}(c)$ (for the definition refer to Chapter 6).

Each call chain c will be assigned a suspiciousness score between $[0, 1]$. For our example, $\mathcal{O}(\texttt{a} \rightarrow \texttt{f}) = \frac{1}{2}$, $\mathcal{O}(\texttt{a} \rightarrow \texttt{g}) = \frac{1}{\sqrt{2}}$, $\mathcal{O}(\texttt{b} \rightarrow \texttt{a} \rightarrow \texttt{f}) = 0$, $\mathcal{O}(\texttt{b} \rightarrow \texttt{g}) = 1$ and $\mathcal{O}(\texttt{b}) = 0$. This, in itself, might be a useful output for the programmer seeking the faulty code element because the high ranked chains could lead their attention to the faulty element and the context in which it was invoked (in our case the call chain `b` $\rightarrow$ `g`). However, we proceed to compute also the most suspicious functions, as described in the following.

8.2.3 Locating functions

A trivial approach for the user to locate the defective function (and statement, respectively) is to consider the highest-ranked call chains and investigate the functions occurring in them (according to our experimentation, this can be successful quite often). But, we also propose an approach to produce a ranked list for functions as well based on the call chain scores. We experimented with various algorithms for this purpose and eventually found out that different strategies may produce good results in different cases. Hence, we decided to use the two best performing strategies and then combine their results, as explained in the following.

Weighted Chain Counts

The basic idea with this strategy is to count the number of occurrences of each function in the chains weighted by the respective chain scores from the previous phase. The intuition behind

this is that functions frequently occurring in highly ranked chains will be more suspicious. More precisely, for each function $f \in F$ we compute the score $\mathcal{W}$ as:

$$\mathcal{W}(f) = \sum_{c \in C(f)} \mathcal{O}(c)\,, \text{where } C(f) = \{c \mid f \in F(c)\}\,.$$

Note that this score will not fall in the interval $[0, 1]$ which is typical for many other scoring mechanisms. However, this does not affect other parts of the approach since only the relative ranks are subsequently used. For our example, the scores will be the following: $\mathcal{W}(\texttt{a}) = \frac{1}{2} + \frac{1}{\sqrt{2}}$, $\mathcal{W}(\texttt{b}) = 1$, $\mathcal{W}(\texttt{f}) = \frac{1}{2}$ and $\mathcal{W}(\texttt{g}) = 1 + \frac{1}{\sqrt{2}}$. This leads to the defective function with the highest score.

Reapplied Spectrum

The second idea for computing function-level scores is to re-apply the spectrum-based approach, but this time on the functions using the call chains in place of the test cases. For this purpose, we treat a call chain as a proxy to a test case in the following manner. If its score is greater than a threshold $\texttt{z} \in [0, 1)$ it is treated as "failing" otherwise as "passing." Our function-level spectrum has the call chains in its rows and the functions in the columns:

$$\mathbf{S}^{fn} = \begin{matrix} \\ c_i \end{matrix} \overbrace{\begin{matrix} \\ \left\{ \begin{bmatrix} 0/1 & 0/1 & \cdots & 0/1 & 0/1 \\ 0/1 & 0/1 & \cdots & 0/1 & 0/1 \\ & & \ddots & & \\ 0/1 & 0/1 & \cdots & 0/1 & 0/1 \\ 0/1 & 0/1 & \cdots & 0/1 & 0/1 \end{bmatrix} \right. \end{matrix}}^{f_j} \quad \mathbf{R}^{fn} = \begin{bmatrix} 0/1 \\ 0/1 \\ \vdots \\ 0/1 \\ 0/1 \end{bmatrix}$$

In this case, a 1 at the matrix position (i, j) means that $f_j \in F(c_i)$, and the entry in the vector $\mathbf{R}^{fn}$ for a chain c_i is 1 if $\mathcal{O}(c_i) > \texttt{z}$. By adjusting $\texttt{z}$, one can regulate how "strictly" a suspicious call chain should be considered as faulty. We experimented with different thresholds, but in the following, we will set $\texttt{z} = 0$ as it provided the best results.

The final scores in this case will be computed by re-applying the Ochiai formula to this function-level spectrum, which will be denoted by $\mathcal{R}(f)$ for a function f. In the example, $\mathcal{R}(\texttt{a}) = \frac{2}{3}$, $\mathcal{R}(\texttt{b}) = \frac{1}{3}$, $\mathcal{R}(\texttt{f}) = \frac{1}{\sqrt{6}}$ and $\mathcal{R}(\texttt{g}) = \frac{2}{\sqrt{6}}$, which again ranks `g` to the first position.

Note, that the simple function-level Ochiai formula scores function `f` with $\frac{1}{2}$ and the other three with $\frac{1}{\sqrt{2}}$, which makes this approach not very useful in this particular case. It is also interesting that the statement-level Ochiai FL will locate the call statement to `g` in function `b`, which is also informative. However, our approach provides more context in a general case because the whole call stack is presented and not only individual code elements. Also, with our second phase not only the chain ranks but the function ranks will be available as well.

Merging the ranks

The reason for the two ranking methods to behave differently can be traced back to the mentioned coincidental correctness, which can affect the chain-based approach as well. Most SBFL formulae perform poorly when the defective element f has a high $ep(f)$ value compared to $ef(f)$. In the case of our reapplied spectrum technique, this means that f is found in many chains that have $\leq \texttt{z}$ score, while in fewer chains that have $> \texttt{z}$ score. This, in turn, can happen if some passing test cases are complex enough to generate a lot of different

```
Input R1 {rank according to W scores}
Input R2 {rank according to R scores}
Output {combined rank}
Repeat
  f := next element in R1
  If f is not output yet
    Output f
  f := next element in R2
  If f is not output yet
    Output f
Until R1=empty and R2=empty
```

Figure 8.4: Rank merging algorithm

chains, contrary to the failing ones. We observed that this can happen often in the case of the reapplied spectrum, so in this case, the weighted chain counts technique will perform better because it is not affected by the many passing chains.

Since we do not know in advance which of the two function-level scores will lead to better results in a particular case, we merged the two ranked lists by alternatively selecting the next element from each of the two lists. The algorithm in Figure 8.4 depicts the approach.

The algorithm has the following property: if the rank position of the faulty element is r_1 in R_1 and r_2 in R_2, in the worst case it will be found in $2 \cdot \min(r_1, r_2)$ steps. This means that if one of the scoring mechanisms is poor compared to the other, the result will depend on the better one. If the two ranks are similar, the output will also be similar to them and the mentioned worst case will not be reached. Note that this algorithm does not explicitly handle ties, situations when elements with the same score are ranked subsequently in an arbitrary order. Also, the rank list with which the processing is started is arbitrary. Depending on how these are implemented, the algorithm could produce different final outputs.

Consider again, our running example. The weighted chain counts approach produces the following ranked list of functions: `gabf`. At the same time, the function-based spectrum results in `gafb`. The merging step outputs either `gabf` or `gafb`, depending on which rank list is the processing started on. The faulty element is in the first position in either case.

8.3 Empirical evaluation

To verify the effectiveness of our approach we conducted an empirical study in which we compared the call chain-based SBFL to a traditional coverage-based SBFL on a benchmark consisting of 404 bugs from the Defects4J suite [87].

8.3.1 Study settings

We performed the experiments on real defects from the Defects4J suite (v1.4.0). We selected this benchmark because it can be seen as the state of the art in SBFL research for Java (see *e.g.* [49, 109, 147] and many others), and it includes real defects and programs with non-trivial size and complexity. The dataset provides the fix for each bug as a patch set (called a *version*). Using the patch sets we were able to create change sets that contain data about which functions (Java methods) were affected by each bug fix.

By default, Defects4J utilizes Cobertura [62], a bytecode instrumentation-based tool to measure code coverage. However, since call chains are needed for our approach we had to use a different technique. We developed a custom bytecode instrumentation tool based on

Javassist [82] to collect execution traces. This tool uses a compact data structure which was carefully engineered to handle recursive calls and the exceptional amount of data that is generated during the execution of real life programs.

Unfortunately, some tests cases fail if the code is instrumented. These tests assert things that the instrumentation changes *e.g.* structure of an object, runtime, contents of the classpath, etc. Since the unexpectedly failing tests would affect the suspiciousness of the covered code elements, we excluded those bugs that include this kind of tests. Finally, we considered only those faults for which there is at least one failing and traversing test case. The final set of programs and defects from the Defects4J dataset we used in our experiments is reported in Table 8.1. Numbers regarding size (lines, tests, functions) vary from version to version, here data from the last versions are provided. The last column contains the number of chains generated (also for the last version).

Program	**KLOC**	**Tests**	**Bugs**	**Functions**	**Chains**
Chart	96	2 187	25	5 235	41k
Closure	91	7 867	173	8 379	889k
Lang	22	2 270	60	2 353	6k
Math	84	4 371	92	6 351	228k
Mockito	11	1 331	28	1 433	11k
Time	28	4 019	26	3 627	150k
Total	**332**	**22 045**	**404**	**27 378**	**1 325k**

Table 8.1: Main properties of the defects used in the experiments

To store the spectrum information matrices and compute the various scores and ranks, we used the SoDA framework [120]. Apart from that only various scripts and spreadsheet editors were used for the calculations.

8.3.2 Evaluation of fault localization effectiveness

Several strategies have been proposed in the literature for measuring the effectiveness of SBFL approaches, but they are practically all based on looking at the rank position of the actual faulty element within the list of all possible program elements. One strategy is to express this as the number of elements that need to be investigated by the programmer before finding the fault [132], and another is the opposite: elements that do not need to be investigated [114]. This is usually expressed in relative terms compared to the length of the rank list (program size). However, Parnin and Orso argued that absolute rankings are more helpful in practical situations [108].

Another issue with these mechanisms is the handling of ties [137], because in many cases different program elements may get assigned the same suspiciousness scores. Some approaches select the first (best case), last (worst case) or middle (expected case) element for expressing this value, while others simply treat the elements with the same values as all belonging to one position.

For computing the effectiveness of an SBFL approach, we follow the strategy to look at "elements that need to be investigated" using the "expected case" in the case of ties and express this in a set of measures called *Expense*. We use two variants of the measure: an absolute one expressed in the number of code elements (E) and a relative version compared to the length of the rank list (E'). The following formulae express precisely how to calculate

this value (following [39]):

$$E = \frac{|\{i|s_i > s_f\}| + |\{i|s_i \geq s_f\}| + 1}{2}\,,\; E' = \frac{E}{N} \cdot 100\ [\%]\,,$$

where N is the number of code elements, for $i \in \{1, \ldots, N\}$ s_i is the suspiciousness score of the ith code element and f is the index of the faulty code element.

To compare our approach to traditional SBFL techniques, we will compute the Expense metric for both approaches and compare them in terms of change relative to traditional SBFL, using both absolute values and relative improvements.

Apart from the general average change, we define the notion of *enabling improvement*, an improvement in which the traditional SBFL algorithm ranks the faulty element beyond the 10th position but the proposed approach reaches it in at most 10 steps. This way, from a practically "hopeless" localization scenario, our approach enables the user to localize the fault by inspecting only the top elements in the list.

8.4 Results

8.4.1 Call chains and faults

The last column of Table 8.1 shows the number of generated call chains of the subject programs (their last versions). This number seems to be related to the number of test cases in the respective project. The distribution of chain lengths varies greatly across the subject programs, and they can be very long as well (up to about 3,500 functions). But, our biggest program, Closure tends to have shorter chains, i.e., about 4 to 26 functions, which means that the call chain length is not related to the program size. Column 3 of Table 8.2 shows the chain lengths with the average in the last row.

Program	**Faulty in High**	**Length All**	**Length High**
Chart	19 (73%)	8.3	5.7
Closure	98 (56%)	26.0	849.3
Lang	56 (88%)	4.4	5.1
Math	70 (75%)	14.8	13.9
Mockito	20 (69%)	7.8	58.6
Time	21 (78%)	10.1	11.7
Total / Average	**284 (69%)**	**24.8**	**749.2**

Table 8.2: Faulty elements in high ranked chains and average chain lengths

We investigated what is the relationship between the faulty elements and the content of the highly-ranked chains produced in the first phase of our approach. The second column of Table 8.2 shows the number of times (and their ratio) the faulty element can be located in the call chains from the very beginning of the ranked list. In particular, we considered the chains with the highest suspiciousness scores. It is interesting to note that the highest score was in many cases 1. We can observe from the data that, for all programs, as much as 69% of the defective elements are found in the highest-ranked chains.

It is also interesting to investigate whether these fault-containing chains are any different in terms of their sizes from the general statistics. Column 4 in Table 8.2 shows the related average values. Compared to the general length of all chains, there can be variations in both directions, but not considering the outlier Closure, the average length of chains with the

highest score is 13.8. This finding indicates that the investigation of only the resulting call chains may often lead to finding the fault. However, the chain with the highest score may also be long, so this process can be supported by the ranked functions in the second phase.

8.4.2 Fault localization effectiveness

Table 8.3 reports the results for fault localization effectiveness. Columns "Ochiai" and "Combined" show the absolute and relative Expense values for function-level Ochiai and for the proposed approach, respectively. Column "Difference" reports the difference between the average rankings, while column "Relative change" expresses the same as percentage increase/decrease with respect to Ochiai. Column "Ochiai > 10" reports the number of defects in the programs for which the ranking position is more than 10. "Enabling improvement" indicates how many defects were successfully moved to the 10th or better position by our approach (the percentage is relative to bug number), and the last column shows the average absolute and relative difference of rankings for such cases.

Program	Bugs	Ochiai $E(E')$	Combined $E(E')$	Difference $E(E')$	Relative change	Ochiai > 10	Enabling improvements	Relative improvement
Chart	25	8.3 (0.19%)	10.8 (0.25%)	2.4 (0.06%)	29%	5	2 (8%)	-19.0 (-76%)
Closure	173	99.5 (1.33%)	131.4 (1.77%)	31.9 (0.44%)	32%	106	16 (9%)	-58.8 (-93%)
Lang	60	4.7 (0.23%)	3.5 (0.17%)	-1.1 (-0.05%)	-24%	7	4 (7%)	-15.4 (-66%)
Math	92	11.0 (0.29%)	7.3 (0.19%)	-3.7 (-0.10%)	-34%	27	17 (18%)	-28.1 (-87%)
Mockito	28	25.6 (2.47%)	20.6 (1.98%)	-5.0 (-0.49%)	-19%	9	3 (11%)	-92.0 (-98%)
Time	26	18.3 (0.53%)	9.5 (0.27%)	-8.8 (-0.26%)	-48%	7	2 (8%)	-49.2 (-94%)
Total / Average	**404**	**49.3 (0.89%)**	**61.1 (1.00%)**	**11.9 (0.11%)**	**24%**	**161**	**44 (11%)**	**-43.0 (-91%)**

Table 8.3: Fault localization effectiveness comparison (averages shown)

For Lang, Math, Mockito and Time, the improvement is measurable in terms of the Expense metric: this ranges from 1 to about 9 ranking positions on average with relative change of 19-48%. For Chart and Closure, the proposed algorithm yields ranking positions that are worse by 29-32% on average compared to Ochiai. Note, that the average ranking that Ochiai scores on the bugs of Closure is 99.5, which is already impractical as developers would unlikely investigate such a large number of functions. Despite the poor average performance on Closure, our approach can still deliver enabling improvements in 16 (9%) cases and the improvement is very high -58.8 (-93%) in these cases.

Our final set of experiments regarding the localization effectiveness deals with the two function localization algorithms that work on the ranked chains, which we introduced in Section 8.2.3. As described, the two techniques performed well in different situations, and it was difficult to predict which approach would be better for a particular case. Hence, we follow the described merging approach, which produces an overall better result than the two individually (in each particular case, twice the minimum is guaranteed). Table 8.4 includes the comparison of these two techniques summarized for each program, with the overall average shown in the last row.

In columns 2 and 3 of the table, we report the average absolute Expense metrics for the respective techniques, while column 4 includes the same data for the merged outcome. The last two columns include the counts when the respective technique performed better than the other. We can conclude from the data that the combined algorithm indeed is useful because there is a similar number of cases when one of the two rankings is better. We also checked the correlation between the scores produced by the two function-level techniques, and we found that it is close to zero. As expected, the combined approach produced an overall better result than any of the other two, however, both approaches are quite close to the combined.

beszedes_242_24

Program	E Weigh.	E Reapp.	E Comb.	Weigh. better	Reapp. better
Chart	13.5	10.6	10.8	12	9
Closure	143.6	149.9	131.4	59	112
Lang	3.7	4.0	3.5	23	19
Math	9.3	7.0	7.3	17	52
Mockito	21.0	36.3	20.6	13	15
Time	16.5	8.0	9.5	9	13
Total / Average	**67.5**	**70.1**	**61.1**	**133**	**220**

Table 8.4: Comparison of weighted chains vs. reapplied spectrum (averages shown)

When comparing function-level rankings, we found that the reapplied spectrum outperforms the weighted chain counts in more cases (220 vs. 133). It also yields better average scores than the combined approach in some cases, though its overall average is not as low as the scores of the combined rank.

8.5 Case study

Besides the ranking improvement, we argue that the additional information provided by the call chains (stack traces) could help the developer even in the situations when the function itself will be further in the rank. As shown, the faulty element is typically found among the highest-ranked chains. For better illustrating the support provided by call chains during FL, let us consider a real case from our benchmark. Bug number 23 from the Joda-Time Defects4J subject[1] can be located in the method `DateTimeZone.getConvertedId`. This causes one test case, `TestDateTimeZone.testForID_String_old`, to fail. The traditional function-level Ochiai SBFL approach (base) provides the localization scores as shown in Table 8.5. Apart from the mentioned faulty element, all other functions are listed that have a score > 0. It can be seen that all functions are executed by the single failing test case and several passing ones as well. However, two of them are executed by fewer passing tests, *i.e.* the faulty one and `DateTimeZone.forTimeZone`, which makes them the most suspicious but indistinguishable from each other.

Method	ef	ep	nf	np	Ochiai
forTimeZone	1	6	0	3822	0.3780
getConvertedId	1	6	0	3822	0.3780
getZone	1	131	0	3697	0.0870
getID	1	528	0	3300	0.0435
setDefault	1	3157	0	671	0.0186
getDefault	1	2884	0	944	0.0178

Table 8.5: Function-level Ochiai for the example (hit-based SBFL)

Figure 8.5 shows the relationship of the mentioned functions, which is an excerpt of a call-graph belonging to this program. `DateTimeZone.forTimeZone` is the main function called by the test case, which apart from the faulty `DateTimeZone.getConvertedId` calls `ZoneInfoProvider.getZone` as well. The other directly called functions are setup and tear-down helper functions for the test case. The reason the base algorithm cannot distinguish

[1] https://github.com/JodaOrg/joda-time/commit/14dedcb

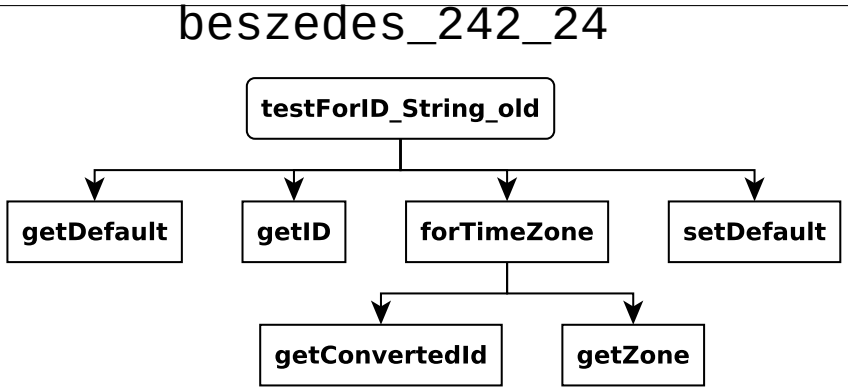


Figure 8.5: Call graph of `TestDateTimeZone.testForID_String_old`

between `forTimeZone` and `getConvertedId` is that the latter is always called (both in failing and passing test cases) by the former, and no additional information is available.

By introducing the concept of the call chains, `forTimeZone` → `getConvertedId` can be investigated separately along with all other call chains. In particular, `forTimeZone` → `getZone` is interesting as it also relates to the suspicious `forTimeZone` but represents a different context. Table 8.6 presents the localization scores calculated by the first phase of the proposed approach, namely the suspiciousness scores for the chains. Similarly, we only show those chains that have > 0 scores. We can observe that apart from the two mentioned chains, the other (one-element) chains are represented as well because they are also part of the failing test run (and also of some passing runs as well).

Chain	**ef**	**ep**	**nf**	**np**	**Ochiai**
forTimeZone→getZone	1	2	0	3826	0.5774
forTimeZone→getConvertedId	1	2	0	3826	0.5774
getID	1	9	0	3819	0.3162
setDefault	1	2882	0	946	0.0186
getDefault	1	2887	0	941	0.0186

Table 8.6: Call chain-based Ochiai for the example

Again, the two highest-ranked chains cannot be distinguished from each other because both are executed in the same situations by the failing and passing test cases. However, the next phase of our approach can pinpoint the faulty elements, because it combines the information about suspicious chains with the functions they contain. Namely, in the reapplied spectrum technique, we treat all suspicious call chains as "failing" and by counting their frequency for each function and the frequency of non-suspicious chains for the same, we can select the most suspicious function. Table 8.7 shows the statistics for this phase. As can be seen, the highest score is given to `getConvertedId`, followed by `forTimeZone`. The explanation for this can also be seen in the corresponding numbers used by the Ochiai formula. Although `forTimeZone` can be found in more suspicious chains than `getConvertedId` (2 vs. 1) it is found in much more non-suspicious chains as well (50 as opposed to 4). `forTimeZone` is a common method called by many test cases, passing and failing, and present in many different chains, but its specific branching to the faulty `getConvertedId` is less frequent and is typical to the failing test case.

This example is realistic and shows one possible benefit of the approach. However, we had to limit its complexity to be able to clearly explain it. The ranking positions 1 and 2, used in the example, are equally good in practical situations, but in more complex cases, the context provided by the call chains could be much more useful.

beszedes_242_24

Chain	ef	ep	nf	np	Ochiai
getConvertedId	1	4	4	87692	0.2000
forTimeZone	2	50	3	87646	0.1240
setDefault	1	78	4	87618	0.0503
getZone	1	77	4	87619	0.0506
getID	1	376	4	87320	0.0230

Table 8.7: Function-level Ochiai for the example (Reapplied Spectrum)

8.6 Conclusions

Our empirical results indicate that, except for outlier cases, the proposed approach can achieve a significant improvement in terms of the FL expense, about 19-48%, which is even higher in the case of worse ranking positions (over 10). The highest-ranked call chains provide useful information for a better understanding of the context of the defect (in 69% of the cases, the defective element is in the highest-ranked chains), and could even provide hints for the fixation of the bug. For instance, the call chain indicates which function invokes the defective function when the fault manifests in a failure.

The two outputs produced by our approach (*i.e.* the ranked list of most suspicious call chains in the first phase and the merged ranked list of functions in the second) can be used in different scenarios to complement hit-based approaches like Ochiai. In a first scenario, the user can start localizing the fault by observing the ranked chains. If the fault is located this way, the context of the investigated chains also informs about the possible ways to fix the defect. If there are many high ranked chains with equally high scores, the user can rely on the final result of the ranked functions from the second phase, and focus on those functions only. In a second scenario, the user starts from the ranked list of functions from the second phase, and if the defect is not easily found, looks at the highest-ranked call chains (and the functions with high ranks in them) for clues about the possible contexts leading to the failed test cases.

Contribution

This chapter is based on the publication:

[8] **Árpád Beszédes**, Ferenc Horváth, Massimiliano Di Penta, and Tibor Gyimóthy. *Leveraging Contextual Information from Function Call Chains to Improve Fault Localization.* In Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2020), pages: 468-479, February 2020, London, Ontario, Canada.

The paper received 5 independent citations so far.

The concept of function call chains and their relation to the call stacks is mostly my contribution, while the use of call chains as context in Spectrum-Based Fault Localization, and the design of call chain-based SBFL algorithms on function level with the associated empirical study are joint work.

9

Use of Call Frequencies in Spectrum-Based Fault Localization

9.1 Introduction

In this chapter, we introduce another extension to the traditional hit-based spectrum used in Spectrum-Based Fault Localization (SBFL), which is also based on the concept on function call chains, the topic of Chapter 8. First, we consider the simple *count-based* spectrum, which records the number of executions of a particular element during runtime, thus replacing the binary spectrum with one that has integer values. However, count-based spectra are relatively underexplored in the literature. Early studies by Harrold et al. [76, 77] have been conducted, and more recently, Abreu et al. [37] concluded that counts do not offer additional benefits over hits. This result could have several explanations, a common one being that the repeated execution of program elements (due to loops) can cause unwanted distortions in test case statistics of the spectrum metrics.

We first verify the mentioned weakness of the simple count-based approach (we call it the *naïve counts* approach), and empirically evaluate its fault localization capability compared to the hit-based approach. Then, we propose a method to improve hit-based spectra using a more advanced count-based approach, which we call the *unique counts* approach. Here, we do not count all occurrences of a program element during execution but only those that occur in *unique call contexts*. Our algorithm is at procedure-level granularity, meaning that the basic program element considered for fault localization is a function or a method. As a call context, we rely on *call stack instances*. In particular, we build on observing the unique deepest call stack instances upon executing a test case, and count the occurrences of methods in these. This way, repeating patterns of method invocations due to, *e.g.*, loops are excluded and only the relevant call context patterns are considered.

We applied the approach on several traditional hit-based SBFL formulae by adapting the spectrum metric calculations and the formulae to handle integer spectra. This adaptation was not trivial, and we experimented with several different approaches to this end. For the empirical assessment we relied on the popular bug benchmark, often used in SBFL research, Defects4J [87].

beszedes_242_24

9.2 Call frequency-based SBFL

We work with three kinds of program spectra for SBFL. The first is the traditional hit-based or coverage-based spectrum which is discussed in Section 9.2.1. Then, we elaborate on the naïve and unique count spectra, in Sections 9.2.2 and 9.2.3, respectively. The adaptations of the SBFL concept to the two count-based approaches are discussed in Section 9.2.4.

9.2.1 Hit-based spectra and risk formulae

Hit-based SBFL uses a binary coverage matrix (see Chapter 6, here denoted by Cov^H) and a test results vector (denoted by R) as the basic data structures to calculate the suspiciousness scores for program elements.

Figure 9.1 contains an adapted version of the example from Chapter 8. Here, $\{\texttt{a}, \texttt{b}, \texttt{f}, \texttt{g}\}$ is the set of program elements (methods), and $\{\texttt{t1}, \texttt{t2}, \texttt{t3}, \texttt{t4}\}$ are the test cases. As can be seen, the four program elements are dependent on each other: method `a` calls methods `f` and `g` directly, while method `b` calls `a` and `g` also directly. We set the bug to be in method `g`, while `a` and `b` call this method. Tests `t1` and `t2` fail due to the bug, and the other tests (`t3`, `t4`) are passing. The code is constructed so that we can emphasize the importance of caller-callee relationships, different call contexts, and the frequency of method calls. The corresponding hit-based spectrum and the spectrum metrics are presented in Table 9.1.

```
public class Example {
  private int _x = 0;
  private int _s = 0;
  public int x() {return _x;}

  public void a(int i) {
    _s = 0;
    if (i==0) return;
    if (i<0) {
      for (int y=0;y<=9;y++)
        f(i);
    } else {
      g(i);
    }
  }

  public void b(int i) {
    _s = 1;
    if (i==0) return;
    if (i<0) {
      for (int y=0;y<Math.abs(i);y++)
        a(0);
    } else {
      for (int y=0;y<=1;y++)
        g(i);
    }
  }

  private void f(int i) {
    _x -= i;
  }

  private void g(int i) {
    _x += (i+_s); //should be _x += i;
  }
}
```

(a) Running example – program

```
public class ExampleTest {
  @Test public void t1() {
    Example tester = new Example();
    tester.a(-1);
    tester.a(1);
    tester.b(1);
    tester.b(-8);
    assertEquals(13, tester.x()); // failed
  }

  @Test public void t2() {
    Example tester = new Example();
    tester.a(1);
    tester.b(1);
    assertEquals(3, tester.x()); // failed
  }

  @Test public void t3() {
    Example tester = new Example();
    tester.a(1);
    tester.b(0);
    assertEquals(1, tester.x());
  }

  @Test public void t4() {
    Example tester = new Example();
    tester.a(-1);
    tester.a(1);
    tester.b(0);
    assertEquals(11, tester.x());
  }
}
```

(b) Running example – test cases

Figure 9.1: Running example for call frequency-based SBFL

In this experiment, we use the 9 risk formulae presented in Chapter 6. The effectiveness of these formulae varies significantly, as noted in the literature. It is evident that all the selected formulae incorporate *ef* in some manner, given that the suspiciousness of a program element is largely influenced by the number of failing test cases passing through it. But, many formulae also include some or all of the other spectrum metrics.

beszedes_242_24

		Hit spectrum (Cov^H)				Spectrum metrics			
		t1	t2	t3	t4	*ef*	*ep*	*nf*	*np*
Methods	a	1	1	1	1	2	2	0	0
	b	1	1	1	1	2	2	0	0
	f	1	0	0	1	1	1	1	1
	g	1	1	1	1	2	2	0	0
Results (R)		1	1	0	0				

Table 9.1: Hit-based spectrum (Cov^H) and spectrum metrics for the running example

The suspiciousness scores for each method in our example and for each risk formula are presented in Table 9.2. Notably, the buggy method (g) is barely distinguishable from the other methods based on these scores. Two algorithms (*Barinel* and *Tarantula*) cannot differentiate between the methods, each assigning a score of 0.5. For the other algorithms, three methods (a, b, and g) share the same suspiciousness value. This means that, according to these techniques, these methods have an equal likelihood of containing a defect.

Methods	***Barinel***	***DStar***	***GP13***	***Jaccard***	***Naish2***	***Ochiai***	***Russell-Rao***	***Sørensen-Dice***	***Tarantula***
a	0.50	2.00	2.25	0.50	1.33	0.71	0.50	0.67	0.50
b	0.50	2.00	2.25	0.50	1.33	0.71	0.50	0.67	0.50
f	0.50	0.50	1.25	0.33	0.67	0.50	0.25	0.50	0.50
g	0.50	2.00	2.25	0.50	1.33	0.71	0.50	0.67	0.50

Table 9.2: Hit-based example scores for the running example

9.2.2 Naïve count-based spectra

Before we introduce our approach of a more elaborate count-based spectra, we first recall the naïve count-based method as it was proposed in previous literature, but gained little popularity due to its inefficiency [76, 77].

This technique incorporates the call frequency by simply counting the number of invocations of the methods while executing a test case. However, there is a fundamental issue with this technique, which has been raised in other research as well, but not actually investigated empirically in detail previously [37]. Namely, with this approach, in a situation where a method is called directly or indirectly from a loop, it will be counted potentially many times. If this call belongs to a failing test case, then it will unnecessarily raise the suspiciousness score of the affected non-faulty methods, which could cause that the actually faulty elements (that are executed less times) remain hidden.

In Figure 9.2 we can see the *dynamic call-tree* for the example in Figure 9.1. A node is a caller when it is a parent node and it is a callee otherwise. For example, a is a callee because t1 and b call it (lines 4-5 in Figure 9.1b and line 21 in Figure 9.1a), but a is also caller for the reason that it "uses" methods g in the else branch (line 13 in Figure 9.1a) and f in the iteration (line 11 in Figure 9.1a). The issue mentioned above can be observed in Figures 9.1 and 9.2: method f is called ten times by t1 (failed test) in a for loop, as opposed to g (faulty method), which is called six times in total by the failed tests (t1 and t2).

The higher execution count of non-faulty methods, such as f and a, compared to the actual faulty method g would result in failing to locate g successfully. Repeated execution in the loop results in high call-value: a(−1), caller method, executes f ten times (Figure 9.1a lines 9-11) and the parameter of b can also result in cyclic repetition. In this example,

method `a` is called eight times by `b` because the parameter (−8) affects the stop condition of the loop (Figure 9.1a lines 19-21), so only counting the execution of a method results in missing the faulty method. The two cases described above illustrate well one of the most important features of the naïve technique (which is also one of the biggest drawbacks): the high degree of sensitivity to repeating code elements.

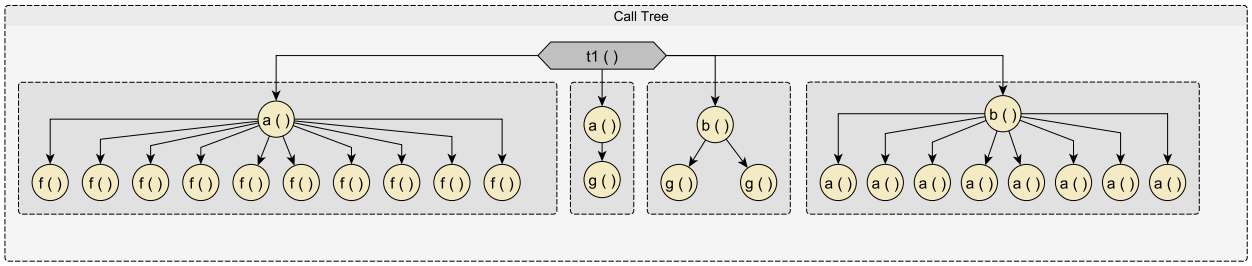


Figure 9.2: Dynamic call-tree collected during the execution of the `t1` test from the running example

The naïve coverage matrix for the example is shown in Table 9.3 (middle matrix). We will refer to this spectrum as the naïve count-based spectrum and denote the matrix as Cov^N (the results vector remains R with the same properties). The naïve matrix is similar to the corresponding binary coverage matrix: where there is a 0 in the hit-matrix, there will be a 0 in the naive-matrix as well, but Cov^N can contain not only 1s but other positive integer values as well, *i.e.*, the number of times a particular test executes an actual code element (a method or function in our case).

		Hit spectrum (Cov^H)				Naïve spectrum (Cov^N)				Unique spectrum (Cov^U)			
		t1	t2	t3	t4	t1	t2	t3	t4	t1	t2	t3	t4
Methods	a	1	1	1	1	10	1	1	2	3	1	1	2
	b	1	1	1	1	2	1	1	1	2	1	1	1
	f	1	0	0	1	10	0	0	10	1	0	0	1
	g	1	1	1	1	3	3	1	1	2	2	1	1
Results (R)		1	1	0	0	1	1	0	0	1	1	0	0

Table 9.3: Naïve (Cov^N) and unique (Cov^U) count-based spectra for the running example (hit-based spectrum is shown for reference)

9.2.3 Unique count-based spectra

Additional included information over hit-based and naïve count-based spectra have been explored previously, *e.g.*, investigating the relationship of the code and the tests then giving weights to them [97, 113], using static or dynamic call graphs [78, 143, 146], and slice-based information [101, 133, 134, 145]. These methods resemble the approach we propose in that they do not generate a new formula but instead "redefine" existing ones by incorporating newly added information and adjusting spectrum metrics.

Our idea to add contextual information is to incorporate how often a specific method has been called (directly or indirectly) and in which context from the test cases. We use the frequency of the investigated method occurring in call stack instances and the number of invocations during the course of executing the test cases. Motivated by research presented in the graph- and slice-based papers (*e.g.* [78, 101, 143, 145]), our basic concept is that when a method is invoked across various contexts during a failing test case, it is more likely to

be responsible for the fault than other methods. The contextual information we use are the *function call chains*, which are introduced in Chapter 8.

We extract test results and stack-trace information from projects at a per-test level with a specially developed tool [119] that includes an online bytecode instrumentation feature. We gather data structures called Unique Deepest Call Stacks (UDCSs). They represent specific instances of call stack snapshots, extending until further methods are transitively called, and halting upon method return. With UDCSs, we utilize coverage data not just in terms of hit information but we can compute how frequently methods appear in these call stacks. For a method under investigation, we aggregate the frequencies of its occurrences across various UDCSs generated by relevant test cases, and this will constitute the *unique count-based spectrum*.

Figure 9.3 illustrates UDCSs for our running example, in which test case `t1` generates UDCSs where four method calls originate directly from the test case. Method `a` is invoked three times, and `b` is invoked twice. These frequencies in the resulting unique deepest call stacks form the foundation of our approach.

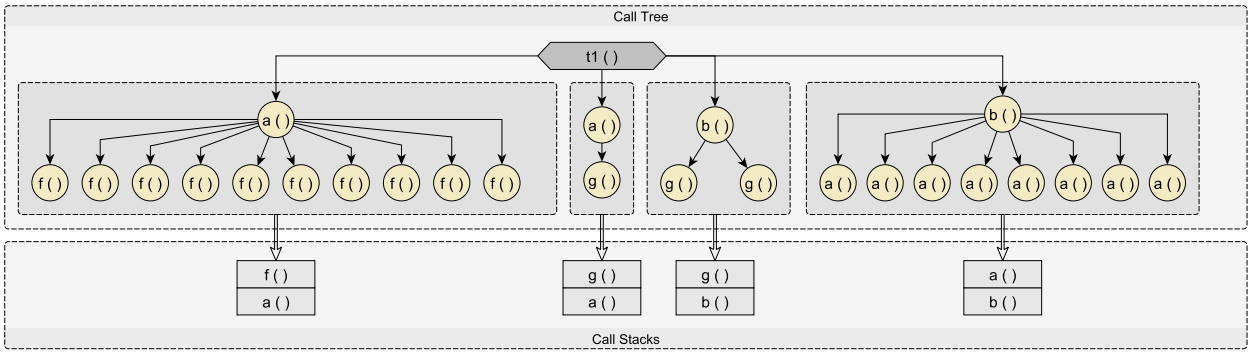


Figure 9.3: Dynamic call-tree and the corresponding unique deepest call stacks (UDCS) collected during the execution of the `t1` test from the running example in Figure 9.1.

The rightmost matrix in Table 9.3, which we will denote by Cov^U, presents a summary of call frequencies within the UDCSs for each method illustrated in the example (the results vector remains R with the same properties as earlier). Similarly to Cov^N, if there is a 0 in a hit-matrix position, there will be a 0 in Cov^U as well, but it can contain not only 1s but other positive integer values as well.

9.2.4 Count-based risk formulae

Adapting the SBFL method to handle frequency-based spectra requires the redefinition of the four spectrum metrics and the risk formulae as well. The following are applicable to both naïve and unique count-based spectra, as the method of calculation is independent from the type of the matrix.

Adapting the spectrum metrics

We introduce the four spectrum metrics for the non-binary matrix as follows. Calculating the two values associated with the tests that executed the code element ($|ef(m)|$ and $|ep(m)|$) is simple: we summarize the (matrix) elements belonging to method m for which the test was failed or passed. We will use the notations $C(ef(m))$ and $C(ep(m))$ for these quantities,

respectively, and define them as follows:

$$C(ef(m)) = \sum_{t \in ef(m)} c_{t,m} \quad \text{and} \quad C(ep(m)) = \sum_{t \in ep(m)} c_{t,m}$$

where $c_{t,m}$ is an element of the Cov^N or Cov^U matrix.

We aimed to incorporate frequency information into the spectrum metrics. In other words, our approach "rewards" methods that repeatedly appear in the UDCS during the execution of both successful and unsuccessful tests. This emphasizes the significance of code elements throughout the execution phase. In the remaining two cases for $|nf(m)|$ and $|np(m)|$, adjusting the metrics poses a greater challenge because not executing an element cannot be simply associated with someting like "how many times not executed". Our approach involves computing the average coverage of tests that remain uncovered by the alternative methods. More precisely,

$$C(nf(m)) = \sum_{t \in nf(m)} \frac{\sum_{m' \in M'} c_{t,m'}}{|M| - 1} \quad \text{and} \quad C(np(m)) = \sum_{t \in np(m)} \frac{\sum_{m' \in M'} c_{t,m'}}{|M| - 1}$$

where M is the set of methods, $M' = M \setminus m$ and $c_{t,m}$ is an element of the Cov^N or Cov^U matrix. These two values indicate the average amount of coverage a method "loses" for passed and failed tests.

Let us consider the naïve count-based spectrum (Table 9.3) and method `f` of our running example to illustrate the adapted spectrum metrics:

- $ef(\texttt{f}) = \{\texttt{t1}\}$ and $C(ef(\texttt{f})) = c_{\texttt{t1,f}} = 10$: `t1` test calls `f` ten times
- $ep(\texttt{f}) = \{\texttt{t4}\}$ and $C(ep(\texttt{f})) = c_{\texttt{t4,f}} = 10$: `f` is used ten times by `t4`
- $nf(\texttt{f}) = \{\texttt{t2}\}$ and $C(nf(\texttt{f})) = \frac{c_{\texttt{t2,a}} + c_{\texttt{t2,b}} + c_{\texttt{t2,g}}}{|\{\texttt{a,b,f,g}\}| - 1} = \frac{1+1+3}{3} = 1.67$: "average coverage" of the failed `t2` not covering `f`
- $np(\texttt{f}) = \{\texttt{t3}\}$ and $C(np(\texttt{f})) = \frac{c_{\texttt{t3,a}} + c_{\texttt{t3,b}} + c_{\texttt{t3,g}}}{|\{\texttt{a,b,f,g}\}| - 1} = \frac{1+1+1}{3} = 1$: "average coverage" of the passed `t3` not covering `f`

Table 9.4 shows the four spectrum metrics for the binary (hit-based) and the two non-binary (naïve and unique count-based) spectra side by side to enable their comparison.

		Hit-based				Naïve count-based				Unique count-based			
		$\|ef(m)\|$	$\|ep(m)\|$	$\|nf(m)\|$	$\|np(m)\|$	$C(ef(m))^N$	$C(ep(m))^N$	$C(nf(m))^N$	$C(np(m))^N$	$C(ef(m))^U$	$C(ep(m))^U$	$C(nf(m))^U$	$C(np(m))^U$
Methods	a	2	2	0	0	11	3	0	0	4	3	0	0
	b	2	2	0	0	3	2	0	0	3	2	0	0
	f	1	1	1	1	10	10	1.67	1	1	1	1.33	1
	g	2	2	0	0	6	2	0	0	4	2	0	0

Table 9.4: Naïve and unique count-based spectrum metrics for the running example (hit-based metrics are shown for reference)

Adapting the risk formulae

The adapted spectrum metrics could be used in new versions of the formulae in different ways, and since it is difficult to predict which strategy would improve fault localization effectiveness, we implemented and empirically evaluated various options. We define the following strategies:

beszedes_242_24

$\Delta^*_{ef^{num}}$ We replace only the $|ef(m)|$ in the numerator provided that it contains only $|ef(m)|$ (this approach cannot be interpreted for the *GP13*, *Naish2* and *Tarantula* formulae).

Δ^*_{ef} Each occurrence of $|ef(m)|$ is overwritten with the new ($C(ef(m))$) value.

Δ^*_{e} The values of all occurrences of $|ef(m)|$ and $|ep(m)|$ are changed in the formula with the corresponding adapted values.

Δ^*_{all} The count-based matrix is used for the calculation of all four metric values in all their occurrences.

These notations are templates to be instantiated with different parameters: Δ symbolizes the formula (B: *Barinel*, D: *DStar*, G: *GP13*, J: *Jaccard*, N: *Naish2*, O: *Ochiai*, R: *Russell-Rao*, S: *Sørensen-Dice* or T: *Tarantula*), $*$ shows what kind of spectrum is used (N: naïve or U: unique), and the replacement strategy is indicated in subscript. Table 9.5 shows example instantiations for *Russell-Rao* using the naïve count-based approach.

$$R^N_{ef^{num}}\colon \frac{C(ef(m))^N}{|ef(m)| + |nf(m)| + |ep(m)| + |np(m)|} \qquad R^N_{ef}\colon \frac{C(ef(m))^N}{C(ef(m))^N + |nf(m)| + |ep(m)| + |np(m)|}$$

$$R^N_{e}\colon \frac{C(ef(m))^N}{C(ef(m))^N + |nf(m)| + C(ep(m))^N + |np(m)|} \qquad R^N_{all}\colon \frac{C(ef(m))^N}{C(ef(m))^N + C(nf(m))^N + C(ep(m))^N + C(np(m))^N}$$

Table 9.5: Adapted *Russell-Rao* formulae using the naïve count-based spectrum

Table 9.6 shows the score values obtained by *Russell-Rao* with the techniques described above for naïve and unique count-based spectra. One of the differences to the values in the hit-based approach is that suspiciousness score values are typically higher. The highest value according to hit-based approach was 0.5 (see Table 9.2, column *Russell-Rao*), while for the naïve and unique count-based concepts, most of the methods scored 0.6 or higher. In addition, the number of ties is much less than in the case of the traditional algorithms.

		Naïve count-based				Unique count-based			
		$R^N_{ef^{num}}$	R^N_{ef}	R^N_{e}	R^N_{all}	$R^U_{ef^{num}}$	R^U_{ef}	R^U_{e}	R^U_{all}
Methods	a	2.75	0.85	0.79	0.79	1.00	0.67	0.57	0.57
	b	0.75	0.60	0.60	0.60	0.75	0.60	0.60	0.60
	f	2.50	0.77	0.45	0.44	0.25	0.25	0.25	0.23
	g	1.50	0.75	0.75	0.75	1.00	0.67	0.67	0.67

Table 9.6: Suspiciousness scores of the methods for the running example calculated using the adapted *Russell-Rao* formulae

As can be seen, all *naïve count-based* versions of the *Russell-Rao* formula associate the highest suspiciousness scores to method `a`, which can be attributed to the corresponding $C(ef(m))$ value being high while $C(ep(m))$ and other values being relatively low. For a similar reason, method `f` and the actually buggy method `g` have the second and third highest suspiciousness values in the case of $R^N_{ef^{num}}$ and R^N_{ef}, where only the $|ef(m)|$ values are replaced. The emphasis that the covering failed tests put on these methods is suppressed by R^N_e and R^N_{all}, where other metric values are also replaced. Interestingly, in these cases method `f` is the least suspicious, hence `g` inherits the second position in the ranked lists.

beszedes_242_24

However, in the case of *unique count-based Russell-Rao* formulae the buggy method is successfully located (except that using $R^U_{ef^{num}}$ and R^U_{ef} method `g` is in tie with method `a`). Having higher suspiciousness score based on the new spectrum metrics and the additional contextual information that the UDCSs provide, `g`, the actual buggy method, can be easily distinguished from the other methods in the case of R^U_e and R^U_{all}.

9.3 Empirical evaluation

The main goal of empirically evaluating the proposed approach was twofold: first, to contrast the fault localization effectiveness of the new algorithms with that of traditional hit-based methods. Second, we aimed to determine which of the traditional SBFL formulae could be most effectively enhanced with call frequency data. In this section, we outline the key parameters of our experiments.

We implemented our approach for analyzing Java programs, and for the evaluation, we selected Defects4J (v2.0.0) [35]. This version of the benchmark contains 17 open source Java projects with manually validated, non-trivial real bugs. The initial dataset contained 835 bugs, but certain cases had to be omitted from the analysis due to instrumentation errors or unreliable test outcomes. 786 defects were deemed suitable for inclusion in the final dataset. Table 9.7 presents the key characteristics of each project.

Subject	**Number of bugs**	**Size (KLOC)**	**Number of tests**	**Number of methods**	**Number of UDCS-s**	**Avg. length of UDCS-s**
Chart	25	96	2.2k	5.2k	122k	8.3
Cli	39	4	0.1k	0.3k	91k	3.7
Closure	168	91	7.9k	8.4k	889k	26.0
Codec	16	10	0.4k	0.5k	6k	9.6
Collections	1	46	15.3k	4.3k	387k	4.2
Compress	47	11	0.4k	1.5k	28k	5.0
Csv	16	1	0.2	0.1k	4k	3.8
Gson	15	12	0.9k	1.0k	126k	1043.1
JacksonCore	25	31	0.4k	1.8k	27k	4.5
JacksonDatabind	101	4	1.6k	6.9k	3467k	17.8
JacksonXml	5	6	0.1k	0.5k	7k	3.8
Jsoup	89	14	0.5k	1.4k	127k	13.5
JxPath	21	21	0.3k	1.7k	215k	57.8
Lang	61	22	2.3k	2.4k	6k	4.4
Math	104	84	4.4k	6.4k	228k	14.8
Mockito	27	11	1.3k	1.4k	11k	7.8
Time	26	28	4.0k	3.6k	150k	10.1

Table 9.7: Properties of subject programs

For evaluating the fault localization effectiveness of the new SBFL algorithms, we relied on the *Expense* and the *Enabling improvement* measures defined in Chapter 8. In addition, we also looked at the *Accuracy* measurement which counts the bugs that have been found at the top positions in the rank list. Several studies indicate that developers typically examine only the initial 5 or 10 elements in the recommendation list generated by the SBFL algorithms [90, 135]. In particular, we consider the bugs that have been localized within the *Top-5* positions. The family of similar metrics is commonly referred to as *Top-N* or *acc@N* [108]. Higher values are better for this metric.

beszedes_242_24

9.4 Results

Note, that certain data in the following evaluation are missing from the tables due to the impossibility to interpret specific variants of formulae, as discussed in Section 9.2.4.

9.4.1 Results for hit-based SBFL

Table 9.8 shows the baseline absolute average rank values for each traditional SBFL formula and subject program. In this setting, the *Ochiai* and *DStar* formulae produce the best results (33.98-33.99), but these are closely followed by *Barinel*, *Jaccard*, *Sørensen-Dice* and *Tarantula* (with average ranks around 36). Next are *GP13* and *Naish2* (43.3 in both cases) with a more significant gap to the other formulae. The worst performing formula by far is *Russell-Rao* (135.96).

Subject	*B*	*D*	*G*	*J*	*N*	*O*	*R*	*S*	*T*
Chart	15.94	9.18	34.34	9.46	34.34	8.82	50.36	9.46	15.94
Cli	16.68	15.29	13.94	16.58	13.94	15.40	23.10	16.58	16.68
Closure	79.44	71.63	95.49	81.10	95.49	71.64	346.62	81.10	79.43
Codec	6.78	6.50	5.25	6.53	5.25	6.53	7.22	6.53	6.78
Collections	1.00	1.00	1.00	1.00	1.00	1.00	8.00	1.00	1.00
Compress	17.49	15.94	15.36	17.14	15.36	16.02	22.74	17.14	17.49
Csv	6.50	6.50	6.50	6.50	6.50	6.50	14.81	6.50	6.50
Gson	19.27	19.23	19.50	19.17	19.50	19.23	26.53	19.17	19.27
JacksonCore	6.84	6.64	9.44	6.36	9.44	6.92	28.18	6.36	6.78
JacksonDatabind	59.53	59.11	64.45	59.57	64.45	59.12	251.63	59.57	59.53
JacksonXml	18.60	18.60	17.80	18.60	17.80	18.60	29.90	18.60	18.60
Jsoup	31.25	30.48	33.31	31.19	33.31	30.44	84.30	31.19	31.25
JxPath	44.24	54.36	116.86	45.00	116.86	54.07	221.95	45.00	44.24
Lang	5.18	4.46	4.39	4.55	4.39	4.46	5.46	4.55	5.18
Math	10.20	10.25	10.83	10.08	10.83	10.32	21.45	10.08	10.20
Mockito	26.11	25.93	42.44	26.07	42.44	25.89	81.81	26.07	26.11
Time	19.79	18.65	22.81	19.67	22.81	18.38	55.50	19.67	19.79
All	36.01	33.99	43.30	36.06	43.30	33.98	135.96	36.06	36.01

Table 9.8: Absolute Expense measure for hit-based formulae. Row "All" represents the mean calculated on all bugs of the dataset. (Notations in the header: *B* - *Barinel*, *D* - *DStar*, *G* - *GP13*, *J* - *Jaccard*, *N* - *Naish2*, *O* - *Ochiai*, *R* - *Russell-Rao*, *S* - *Sørensen-Dice* and *T* - *Tarantula*)

It can be observed that Closure has a much higher average than the rest of the programs. This may be due to the fact that Closure has a different purpose than most of the other subjects, consequently a special program and test suite structure as well. While the rest of the subjects are smaller Java libraries, Closure is a large compiler tool for JavaScript. Therefore, most of its tests are complex ones (system tests as opposed to unit tests), and also these test cases have to go through a common starting phase (the initialization of the compiler, preprocessing, parsing, etc.) before they reach the sophisticated parts of the compiler itself, which usually contain the root cause of bugs. Hence, they generate very large UDCSs (see Table 9.7) resulting in high Expense values.

beszedes_242_24

9.4.2 Results for naïve count-based SBFL

Table 9.9 shows the average Expense values for each naïve count-based formula variant. For reference, we included the results of the hit-based SBFL formulae (row "hit") and highlighted the best results in bold for each column.

Var.	*B*	*D*	*G*	*J*	*N*	*O*	*R*	*S*	*T*
hit	**36.01**	**33.99**	**43.30**	**36.06**	**43.30**	**33.98**	135.96	**36.06**	**36.01**
$\Delta^N_{ef^{num}}$	63.46	97.94		63.55		94.76	167.78	64.11	
Δ^N_{ef}	62.60	97.94	153.43	62.89	153.43	62.59	167.41	62.89	37.09
Δ^N_{e}	42.73	66.14	153.98	43.03	153.60	41.85	**73.15**	43.03	44.34
Δ^N_{all}	42.73	66.38	153.98	43.10	153.45	41.74	126.12	43.10	44.80

Table 9.9: Average Expense of naïve count-based formulae. Row "hit" represents the corresponding values from Table 9.8.

The data clearly shows that the naïve count-based approach could not outperform the traditional hit-based approach in terms of average Expense, except for one case with the *Russell-Rao* formula. In fact, the values are much worse. However, looking at only the average values can be misleading. First, outliers could distort the overall information on the performance of our formula. Second, this data provides nothing about the distribution of the different rank values. We believe that not all (absolute) rank positions are equally important, as we discussed in Section 9.3. Hence, in the next set of experiments we will concentrate on the Top-5 (Accuracy) findings.

In Table 9.10, we can see the number of successfully localized bugs within the Top-5 category (with the respective percentages), accumulated for the whole benchmark, for each naïve and, for reference, the hit-based formula. The best values for each category (in columns) are highlighted in bold.

	hit		$\Delta^N_{ef^{num}}$				Δ^N_{ef}				Δ^N_{e}				Δ^N_{all}			
	#	%	#	%	Det.	E. Im.	#	%	Det.	E. Im.	#	%	Det.	E. Im.	#	%	Det.	E. Im.
B	357	(45.4%)	**288**	**(36.6%)**	101	32	297	(37.8%)	86	26	320	(40.7%)	65	28	320	(40.7%)	65	28
D	366	(46.6%)	232	(29.5%)	164	30	232	(29.5%)	164	30	276	(35.1%)	119	29	279	(35.5%)	118	31
G	361	(45.9%)					166	(21.1%)	214	19	163	(20.7%)	219	21	163	(20.7%)	219	21
J	358	(45.5%)	286	(36.4%)	105	33	297	(37.8%)	91	30	319	(40.6%)	69	30	**321**	**(40.8%)**	68	31
N	361	(45.9%)					166	(21.1%)	214	19	163	(20.7%)	219	21	163	(20.7%)	219	21
O	**367**	**(46.7%)**	227	(28.9%)	170	30	295	(37.5%)	101	29	**322**	**(41.0%)**	74	29	**321**	**(40.8%)**	74	28
R	111	(14.1%)	142	(18.1%)	**10**	**41**	143	(18.2%)	9	**41**	220	(28.0%)	**5**	**114**	172	(21.9%)	**9**	**70**
S	358	(45.5%)	279	(35.5%)	113	34	297	(37.8%)	91	30	319	(40.6%)	69	30	**321**	**(40.8%)**	68	31
T	357	(45.4%)					**350**	**(44.5%)**	**8**	1	320	(40.7%)	56	19	311	(39.6%)	65	19

Table 9.10: Accuracy (number of bugs in the Top-5 category) and enabling improvements for hit-based and naïve count-based formulae (highlighted are the best values in each column)

We can instantly observe that the hit-based formulae outperformed all the rest in each of the categories by a large margin. The only exceptions are *Tarantula* whose T^N_{ef} variant performed only a bit worse than its hit-based counterpart, and *Russell-Rao* whose naïve variants were able to improve the overall worst result among the hit-based formulae. Similarly to earlier findings, the least bad performance is achieved by the variants of *Ochiai*, *Jaccard*, *Barinel*, *Sørensen-Dice* and *Tarantula*. Considering the overall performance, the best results were achieved by the Δ^N_e and Δ^N_{all} variants.

Table 9.10 also summarizes the enabling improvements in the columns noted with "E. Im. " for each formula. In addition, the columns noted with "Det. " show the number of bugs whose position was deteriorated, *i.e.*, the rank calculated by the hit-based formula was $<= 5$ while the naïve formula produced a rank that was > 5. Although each naïve formula

achieves enabling improvements, the number of these cases is low, except for *Russell-Rao* (which is aligned to what we have observed earlier).

9.4.3 Results for unique count-based SBFL

In this section, we present the results of our experiments on the unique count-based formulae in a similar fashion to the previous one.

Table 9.11 shows the average Expense values for each unique count-based formula variant. Best results are highlighted in bold, and the hit-based results are included in row "hit".

Var.	*B*	*D*	*G*	*J*	*N*	*O*	*R*	*S*	*T*
hit	36.01	33.99	**43.30**	36.06	**43.30**	33.98	135.96	36.06	**36.01**
$\Delta^U_{ef^{num}}$	24.68	35.60		24.63		36.05	70.80	24.89	
Δ^U_{ef}	**24.55**	35.60	66.73	**24.40**	66.73	**24.30**	70.60	**24.40**	36.87
Δ^U_{e}	38.63	**29.08**	67.19	38.64	67.04	36.44	**35.12**	38.64	85.35
Δ^U_{all}	38.63	29.13	67.19	38.34	67.00	36.99	54.95	38.34	74.56

Table 9.11: Average Expense of unique count-based formulae. Row "hit" represents the corresponding values from Table 9.8.

In this set of data we can recognize similar patterns to what we observed for the naïve count-based approach, but this time there are notable improvements compared to the hit-based method: for 6 formulae out of 9, the hit-based method was improved significantly. The unique count variants of *Russell-Rao* achieve large improvements: 65.2-100.8 ranks on average on the whole dataset, but the performance of the hit-based *Russell-Rao* formula was poor to start with.

Considering *Tarantula*, it could not improve the hit-based results, the least bad variant is T^U_{ef} which is behind the original *Tarantula* by 0.9. *GP13* and *Naish2* show similar performance. The unique count variants of *Barinel*, *Jaccard* and *Sørensen-Dice* achieve about 11.5-11.7 improvement with the Δ^U_{ef} configuration. Different variants of *DStar* and *Ochiai* have 4.9-9.7 advantage over the hit-based versions. Overall, the best result are 24.30 - $Ochiai^U_{ef}$, 24.40 - $Jaccard^U_{ef}$ and $Sørensen\text{-}Dice^U_{ef}$, 24.55 - $Barinel^U_{ef}$, and 29.08 - $DStar^U_{e}$.

In Table 9.12, we can see the number of bugs belonging to the Top-5 category (with the respective percentages), accumulated for the whole benchmark, for each unique count-based and the hit-based formula. It also presents the enabling improvements and the number of bugs which were moved in the opposite direction, which were deteriorated.

	hit		$\Delta^U_{ef^{num}}$				Δ^U_{ef}				Δ^U_{e}				Δ^U_{all}			
	#	%	#	%	Det.	E. Im.	#	%	Det.	E. Im.	#	%	Det.	E. Im.	#	%	Det.	E. Im.
B	357	(45.4%)	**373**	**(47.5%)**	78	94	376	(47.8%)	65	84	354	(45.0%)	45	42	354	(45.0%)	45	42
D	366	(46.6%)	327	(41.6%)	128	89	327	(41.6%)	128	89	**391**	**(49.7%)**	86	111	**388**	**(49.4%)**	88	110
G	361	(45.9%)					266	(33.8%)	172	77	269	(34.2%)	170	78	269	(34.2%)	170	78
J	358	(45.5%)	372	(47.3%)	81	95	**380**	**(48.3%)**	69	91	357	(45.4%)	47	46	356	(45.3%)	48	46
N	361	(45.9%)					266	(33.8%)	172	77	269	(34.2%)	170	78	269	(34.2%)	170	78
O	**367**	**(46.7%)**	323	(41.1%)	135	91	**380**	**(48.3%)**	74	87	363	(46.2%)	51	47	361	(45.9%)	54	48
R	111	(14.1%)	249	(31.7%)	**12**	**150**	248	(31.6%)	**12**	**149**	380	(48.3%)	**6**	**275**	356	(45.3%)	**10**	**255**
S	358	(45.5%)	366	(46.6%)	87	95	**380**	**(48.3%)**	69	91	357	(45.4%)	47	46	356	(45.3%)	48	46
T	357	(45.4%)					351	(44.7%)	**12**	6	258	(32.8%)	111	12	254	(32.3%)	117	14

Table 9.12: Accuracy (number of bugs in the Top-5 category) and enabling improvements for hit-based and unique count-based formulae (highlighted are the best values in each column)

Every unique count-based formula achieves enabling improvements, but there are about the same number of deteriorations as well. On average, the number of enabling improvements is around 70-110, but *Russell-Rao* and *DStar* perform exceptionally well.

There are notable improvements regarding the Top-5 accuracy as well. Traditional formulae rank at most 367 (46.7%) bugs into the Top-5 thanks to *Ochiai*, while the best unique count-based formulae have 380-391 (48.3-49.7%) bugs in the Top-5 which is about 4-7% improvement. The best accuracy values can be accounted to the Δ_e^U variant of *DStar* (391) and $DStar_{all}^U$ (388), closely followed by $Jaccard_{ef}^U$, $Ochiai_{ef}^U$ and $Sørensen\text{-}Dice_{ef}^U$ (380) and $Barinel_{ef^{num}}^U$ (373). The improved *Russell-Rao* again outperforms the baseline hit-based formulae by a huge margin.

Summary. In 6 out of 9 cases, the unique count-based approaches can improve the effectiveness of their hit-based counterparts by 5-101 positions on average. At the same time, the naïve count-based approach is usually worse than the traditional method. Also, all new formulae are able to achieve enabling improvements, and the accuracy regarding the number of bugs in the Top-5 category is increased by about 4-7% as well. Considering the magnitude of differences and the consistency of the improvements, Δ_{ef}^U offers the best performance in the unique count-based setting, but the other variants are only marginally behind as well. This strategy emphasizes the importance of the relationship between the failing tests and exercised code elements.

9.5 Conclusions

This research confirms earlier results that the naïve approach of extending hit-based spectra using execution counts does not lead to improvements. Our concept of unique counts relies on the notion of Unique Deepest Call Stacks, data structures that capture call stack state information occurring on test case execution, and count the number of method occurrences within these structures, and this way we can eliminate the problem of large number of code repetitions due to loops. Empirical measurements confirm that this approach can improve the hit-based method to a large degree.

Our method of adapting the spectrum metrics and the risk formulae is novel in the field of Spectrum-Based Fault Localization, and opens up possibilities for further research. For example, it is an open question how the technique could be adapted to other granularities, primarily statements, and what other kinds of contextual information could be used besides call frequencies.

Contribution

This chapter is based on the publication:

[32] Béla Vancsics, Ferenc Horváth, Attila Szatmári, and **Árpád Beszédes**. *Fault Localization Using Function Call Frequencies.* Journal of Systems and Software, Volume: 193, 2022, publisher: Elsevier. (conference version: [31])

These papers received 6 independent citations so far.

The definition of the two procedure call frequency concepts and the extension of the naïve counts using call stacks are mostly my contribution, while the use of call frequency based on call chains to improve SBFL algorithms, and the adaptation of SBFL spectra, spectrum metrics, and risk formulae with the associated empirical study are joint work.

10 Use of Dynamic Slicing in Spectrum-Based Fault Localization

10.1 Introduction

Coverage-based (hit-based) SBFL is straightforward to implement using existing profiling tools, and the corresponding algorithms to calculate faultiness are simple, hence this provided a fruitful ground for a large set of SBFL techniques proposed in the literature (see Chapter 6). An important insight about coverage-based spectra is that it is based on the assumption that a code element covered by failing tests should be treated as suspicious. However, this is an over-approximation of the original intent to look for the code *responsible for the fault*. The faulty code element must be executed in a failing run, but it also needs to cause a failure-inducing chain toward the output [141]. If a statement is executed but it is not participating in the computation responsible for the failure, it causes noise in the SBFL process.

In other words, instead of the simple coverage information, only its subset should be used, which takes part in the computation. This is, precisely, the concept of the *program slice* [129]. In particular, we are interested in the *backward dynamic program slice* [92] computed from the output statement as the criterion, and use this information in the program spectra. If computed properly, a dynamic program slice will be a subset of the coverage information and in the spectrum, it will provide exactly the information which is needed by SBFL formulae. (For an overview of dynamic slicing concepts, refer to Chapter 2.)

The question then is: how big is the influence of the over-approximation caused by the coverage-based spectrum compared to the slice-based? This depends on the relative size of the slices with respect to the coverage information, and the way superfluous elements affect the SBFL algorithm. In other words, what is the effect of the executed code elements that are not in the slice on the final ranking lists?

The idea of combining program slices and SBFL is not new, and there are various approaches to do the same. Surprisingly, relatively few studies among these utilize backward dynamic slices in place of the coverage in the program spectrum [47, 101, 112]. Also, these studies do not elaborate on the relationship of the coverage-based and slice-based spectra, typically only high-level measurement results are provided. The main reason for this modest visibility could be very pragmatic: computing precise slices requires difficult algorithms, and the computation costs can be very high compared to simply using the coverage.

We aim at filling this gap and providing more insight into "how bad is the coverage spectrum?" when compared to the slice-based spectrum, and what are the typical situations where the deficiencies manifest. Given the fact that dynamic slices can be quite small (about 33% [144] to 50% [57] of the executed instructions on average), we expect a large impact on the overall algorithm effectiveness.

It is not the aim of our experiments to discuss concrete slicing techniques and their effect on SBFL, just the conceptual relationship backed up by an empirical case study to illustrate the differences. We provide a theoretical analysis of why coverage-based spectra necessarily produce suboptimal results compared to dynamic slice-based spectra. We also implemented a dynamic slice-based SBFL method using a precise yet feasible approach, and performed a case study using a well-known subject program and real faults. We thoroughly analyzed every fault in the case study to understand the most typical causes of suboptimal performance of the coverage-based approach.

10.2 Coverage vs. slice-based spectra

In fault localization literature, several works explore the possibilities of combining the spectrum-based approach with program slicing [47, 101, 112]. In this work, we concentrate on the class of methods in which the traditional SBFL method based on code coverage is modified by replacing the spectrum matrix with slice information. Several variations are possible at this point, but we define the *slice-based spectrum* as the spectrum matrix whose rows include the backward dynamic slices computed from the corresponding test cases instead of their coverages (the results vector remains the same).

In this chapter, we will rely on the notations from Chapter 6 with the following extensions. The traditional coverage-based (hit-based) spectrum matrix will be denoted by M, while we will use M' for its slice-based counterpart. $C(t) \subseteq E$ will denote the set corresponding to the t-th row in M, *i.e.*, the set of covered elements by test case t in the coverage-based matrix. To evaluate the fault localization effectiveness, information about the known faults will be used from benchmark programs. It will be represented by the *faults vector* F of size $|E|$ in which $f_j = 1$ if the j-th code element contains a fault. For simplicity, we will also use the same notations M, R, and F to represent not only the matrix/vectors but the corresponding sets and functions as well, depending on the context.

This research deals with backward dynamic program slicing (see Chapter 2) with the slicing criterion being the "output statement" of the test case. In practice, the output statement may correspond to an assertion point in the test case with the asserted variable. In the following, we will use $DS(t) \subseteq E$ to denote the backward dynamic program slice corresponding to test case t. Hence, M' is constructed by replacing each $C(t)$ in matrix M by $DS(t)$.

The benefit of a slice-based SBFL over a coverage-based one can be easily illustrated in a simple example. Listing 10.1 includes a Java snippet of a circle implementation with methods for calculating the area and perimeter, along with three associated unit tests in Listing 10.2. The execution of the tests results in `t2()` failing due to the bug in line 6 for calculating the perimeter, and the other two passing. The coverage-based spectrum, along with the spectrum metrics, and the suspiciousness scores computed by the Barinel formula in the bottom row (see Chapter 6 for definition) are shown in Table 10.1 (left hand side).

The SBFL formula cannot distinguish between code lines 5 and 6 as the simple coverage is over-approximating the actual calculation chains: both constructor lines are included in all passing and failing tests. The slice-based spectrum differs from the coverage one exactly at these two critical lines. The backward dynamic slices computed from the test cases correctly include only the appropriate lines setting the `area` or `perimeter` fields, respectively. This

beszedes_242_24

```
public class Circle {
   private double area;
   private double perimeter;
   public Circle(double radius) {
      area = radius * radius * Math.PI;
      perimeter = radius * Math.PI; // faulty statement
   }
   double getArea() {
      return area;
   }
   double getPerimeter() {
      return perimeter;
   }
}
```

Listing 10.1: Faulty code example

```
public class CircleTest extends TestCase {
  static Circle circle = new Circle(0);
   public void t1() {
      assertEquals(Math.PI, new Circle(1).getArea(), 1e-10);
   }
   public void t2() {
      assertEquals(2.0*Math.PI, new Circle(1).getPerimeter(), 1e-10);
   }
   public void t3() {
      assertEquals(0, circle.getPerimeter(), 1e-10);
   }
}
```

Listing 10.2: Tests for the faulty code

can be seen in the right hand side of Table 10.1, where the differences to the coverage-based spectrum are underlined. The result is that the faulty line 6 is now correctly localized at the first ranking position with the score 1 ($ef = 1$ and $ep = 0$), while the non-faulty line 5 gets a score 0 with $ef = 0$ and $ep = 1$).

As mentioned, there are only a handful of researchers who utilized this concept to develop a combined SBFL and slicing approach. Mao *et al.* [101] presented an approach that used slices to construct more precise program spectra, and they experimented with various slicing algorithms including Approximate Dynamic Backward Slicing and Relevant Slicing. The algorithm called *Tandem-FL* was proposed by Reis *et al.* [112], which is able to locate and reduce the suspiciousness of those components that are mostly involved in failing tests but seldom covered by passing ones. Their idea uses SBFL to calculate the suspiciousness of

	4	5	6	8	9	11	12	R
$C(t1)$	1	1	**1**	1	1	0	0	0
$C(t2)$	1	**1**	1	0	0	1	1	1
$C(t3)$	0	0	0	0	0	1	1	0
ef	1	**1**	1	0	0	1	1	
ep	1	1	**1**	1	1	1	1	
nf	0	**0**	0	1	1	0	0	
np	1	1	**1**	1	1	1	1	
Barinel	0.5	**0.5**	**0.5**	0.0	0.0	0.5	0.5	

	4	5	6	8	9	11	12	R
$DS(t1)$	1	1	**0**	1	1	0	0	0
$DS(t2)$	1	**0**	1	0	0	1	1	1
$DS(t3)$	0	0	0	0	0	1	1	0
ef	1	**0**	1	0	0	1	1	
ep	1	1	**0**	1	1	1	1	
nf	0	**1**	0	1	1	0	0	
np	1	1	**2**	1	1	1	1	
Barinel	0.5	**0.0**	**1.0**	0.0	0.0	0.5	0.5	

Table 10.1: Coverage (left) and slice-based (right) spectra and fault localization results for the example

each element, then select the top k from the list. Alves *et al.* [47] use the basic slice-based approach but introduce variations to accommodate for change-based analysis. None of these reports deal with analyzing the differences between the coverage and the slice information nor seek to understand how much the former one is over-approximated. Furthermore, all the other related research we are aware of combines the two techniques differently.

10.3 Theoretical analysis of coverage and slice

We will use notations M', ef', nf', ep', and np' to denote the slice-based spectrum matrix, and the associated spectrum metrics. For this theoretical analysis, we assume the following:

1. Program P includes exactly one fault which can be identified at a single code location, *i.e.*, $|F| = 1$. The faulty code element will be denoted by f in the following (n will be used for all other elements).

2. All coverages include at least one code element, *i.e.*, $\forall t \in T \ : \ |C(t)| > 0$

3. The faulty code element is executed by all failing test cases, *i.e.*, $\forall t \in T \ : \ R(t) = 1 \Rightarrow M(t, f) = 1$

4. Each test case t can be associated with exactly one slicing criterion, which means that rows of matrices M and M' are compatible.

5. The backward dynamic slice is computed correctly, meaning $\forall t \in T \ : \ DS(t) \subseteq C(t) \subseteq E$, furthermore

6. f contributes to the slicing criterion in all failing test cases, implying that

7. f is included in all slices for failing test cases, *i.e.*, $\forall t \in T \ : \ R(t) = 1 \Rightarrow M'(t, f) = 1$.

8. All slices include at least one code element, *i.e.*, $\forall t \in T \ : \ |DS(t)| > 0$.

Following the property of the dynamic slice being the subset of the code coverage, we can look at how much more precise it is. We can express this in terms of the slice size with respect to the coverage size for each test case t in a program P. The average slice size will be used as a proxy to the probability $p \in (0, 1]$ that a covered code element e will be also in the slice:

$$p = \frac{\sum_{t \in T} \frac{|DS(t)|}{|C(t)|}}{|T|}$$

Based on the assumptions above, we can make the following observations. For the faulty element f, $ef'(f) = ef(f)$ and $nf'(f) = nf(f)$ because each failing test's slice must include f and both M and M' have the same T set of tests. Regarding the passing tests, there are no such requirements, so $ep'(f) \subseteq ep(f)$ and $np'(f) \supseteq np(f)$. In the case of any other non-faulty element n, the subset relationship will be the same for all four sets. We can calculate the expected values of the four spectrum metrics for the slice-based spectrum as:

For f:	For any n:
$ef' = ef$	$ef' = p \cdot ef$
$nf' = nf$	$nf' = (1 - p) \cdot ef + nf$
$ep' = p \cdot ep$	$ep' = p \cdot ep$
$np' = (1 - p) \cdot ep + np$	$np' = (1 - p) \cdot ep + np$

beszedes_242_24

We can now investigate how values of the suspiciousness formulae relate to each other for the same program P but computed on M and M'. For the faulty element, the value of ef does not change while the others get smaller, and since many of the formulae include ef in the numerator and the others in some form in the denominator, the score value will usually be bigger in the slice-based spectrum. For the non-faulty element, the score will typically either be the same or smaller.[1]

Table 10.2 shows the formulae we work with in this research in more detail (the original definitions are given in Chapter 2, here we use abbreviated names). Here, the slice-based spectrum metrics are simply substituted to form the slice-based formulae.

$Bar'(f) = \frac{ef}{ef+p\cdot ep} \geq Bar(f)$	$Bar'(n) = \frac{p\cdot ef}{p\cdot ef+p\cdot ep} = Bar(n)$
$Tar'(f) = \frac{\frac{ef}{ef+nf}}{\frac{ef}{ef+nf}+\frac{p\cdot ep}{ep+np}} \geq Tar(f)$	$Tar'(n) = \frac{\frac{p\cdot ef}{ef+nf}}{\frac{p\cdot ef}{ef+nf}+\frac{p\cdot ep}{ep+np}} = Tar(n)$
$Och'(f) = \frac{ef}{\sqrt{(ef+nf)\cdot(ef+p\cdot ep)}} \geq Och(f)$	$Och'(n) = \frac{p\cdot ef}{\sqrt{(ef+nf)\cdot(p\cdot ef+p\cdot ep)}} = \sqrt{p}\cdot Och(n) \leq Och(n)$
$Jac'(f) = \frac{ef}{ef+nf+p\cdot ep} \geq Jac(f)$	$Jac'(n) = \frac{p\cdot ef}{p\cdot ef+(1-p)\cdot ef+nf+p\cdot ep} = \frac{ef}{\frac{ef}{p}+\frac{nf}{p}+ep} \leq Jac(n)$
$Sor'(f) = \frac{2\cdot ef}{2\cdot ef+nf+p\cdot ep} \geq Sor(f)$	$Sor'(n) = \frac{2p\cdot ef}{2p\cdot ef+(1-p)\cdot ef+nf+p\cdot ep} = \frac{2\cdot ef}{\frac{p+1}{p}ef+\frac{nf}{p}+ep} \leq Sor(n)$
$Dst'(f) = \frac{ef^2}{p\cdot ep+nf} \geq Dst(f)$	$Dst'(n) = \frac{p^2\cdot ef^2}{p\cdot ep+(1-p)\cdot ef+nf} = \frac{ef^2}{\frac{ep}{p}+\frac{nf}{p^2}+\frac{1-p}{p^2}ef} \leq Dst(n)$

Table 10.2: Coverage and slice-based formula relationships

We could show that, in all investigated cases, the suspiciousness score of the faulty element is the same or bigger in the slice-based spectrum than for the coverage-based one, while all non-faulty elements' scores are either smaller or the same for the slice-based spectrum. We checked several other published formulae if they exhibit this property and found that they do, but we cannot rule out the possibility that there are some counter-examples. However, we expect that any formula that has a meaningful combination of the spectrum metrics will behave similarly.

We can also observe from the above that the smaller p is the bigger the difference will be between the two methods. The effect is that, with these assumptions, the coverage-based SBFL necessarily produces a worse ranking than the slice-based SBFL, due to the over-approximation of the real dynamic dependences using the coverage, and the bigger this over-approximation the worse the result will be.

The assumptions from above will not necessarily hold for realistic situations, but we believe that they are good approximations of reality. Furthermore, the benchmarks typically used in related research often aim at achieving these ideal situations. Slicing tools are not perfect either, and they may violate one or more of the assumptions. Nevertheless, we think that further research is necessary to understand what the performance of realistic implementations and real programs and faults is. Also, the imprecision of coverage-based SBFL with respect to slice-based SBFL should be measured in practice by looking at the slice sizes (p) since this turns out to be an essential parameter.

[1]It is worth noting that this shows the average case and the expected values based on the average slice size, but in reality, the final scores for the individual elements can change in either direction because the individual n elements can have various slice ratios for passing and failing cases.

10.4 Case Study

The goal of our case study was to verify the presented concepts in practice, which may serve as a motivation for further research. We verify the relationship between the coverage-based and slice-based spectra in depth, but instead of performing an extensive empirical evaluation involving multiple programs with bugs and reporting overall high-level results, we selected one subject program from a benchmark suite and evaluated each fault separately in detail.

Our goal with the case study was to implement the basic method outlined as closely as possible, *i.e.*, we did not want to use approximate slicing algorithms or other optimizations on the matrix. There were several difficulties, however, including the imperfection of the slicer tool we selected, and the way test cases, slicing criteria, and code elements could be matched, as discussed in the later section.

10.4.1 Study settings

Creating Slice-Based Spectra

Coverage-based program spectra might include statements that do not affect the tested value. These additional but irrelevant statements can lower the effectiveness of the method. The solution already proposed by previous works is to compute the spectrum from slices [101]. We use this kind of "test-slice" spectrum in our evaluation.

In an ideal case, a test should check only one value, and there are test environments where this is ensured. For example, sometimes the output of the program under test is written to the standard output or a file (using a single statement) for later comparison with a reference output. However, in practice, especially in unit test frameworks, a single test usually checks multiple values. In unit tests, this is implemented as multiple assertions in a single test case, and from the execution logs of a test case, it can be determined which assertion has failed.

Decomposing the tests and creating the spectrum for assertions instead of tests might produce more detailed information on the position of the fault. In other words, we create one row to the slice-based spectrum matrix *for each assert* rather than for each test case. To be able to compare the slice-based results to the traditional coverage-based ones, we then merge assert-slices for each test case by calculating the union of assertion slices per test case.

Slicing Tool

During the preparation of our experiments, we tested several tools that are capable of creating dynamic backward slices, Slicer4J [36, 45], JavaSlicer [74], and Java SDG Slicer [67] amongst others. However, most of the publicly available tools are not well-maintained, and they have deprecated or unavailable dependences. Finally, we decided to use the open-source dynamic slicing tool Slicer4J to collect the slices. It uses low-overhead instrumentation to collect a runtime execution trace; it then constructs a thread-aware, inter-procedural dynamic control-flow graph, and a set of pre-constructed data-flow summaries to compute the slice.

Determining the Slicing Criteria

Determining the slicing criterion is relatively straightforward in the case of unit tests. We have asserts in the test cases that check some actual computed values against some expected values. We should simply slice for the actual values used in the assert statements. As the slicer we used is able to slice for a source code line (i.e., it is enough to give a line number as a slicing criterion and it will compute slices for all appropriate variables of that line), we first simply selected those lines of the test cases that contained asserts and passed

these lines to the slicer. At the same time, we had to employ some workarounds to specific exception-handling constructs found in the subject program.

Spectrum matrices and fault localization

Since we used the traditional coverage-based SBFL approaches as the baseline of our evaluation, we had to calculate the corresponding results as well. To extract the coverage-based program spectra and calculate the results based on them, we used the approach published by Pearson *et al.* [109].

The slicer and coverage tools identified the tests and instructions in a mostly similar but slightly different way. The basis of the instruction identifier we used is the fully qualified name of the Java class and the line number. For the test cases, we used the fully qualified Java method name (without return value and parameter specification). The asserts were also identified by test case name and an additional absolute line number of the assert in the file. While the two dimensions of our proposed spectra are asserts and instructions, in the case study, we aggregated our slices by test cases. Thus, we computed the slices of all asserts of a given test case and assigned their union to the test case as its slice set. We used our own scripts to calculate the slice-based spectrum matrix from the raw data produced by the slicer, the spectrum metrics, and the ranks.

Subject program

We focus on the qualitative evaluation of the differences among traditional coverage-based and slice-based program spectra, hence we used a subset of the bugs of the program Time from the Defects4J (v2.0.0) [35] benchmark. We chose Time as our subject because its domain is fairly easy to understand and this gives us the opportunity to demonstrate the effects of the different approaches more easily. In addition, the complexity of the program, the tests, and the faults is medium and could be regarded as typical for this benchmark. There are 26 bugs (program versions) in this benchmark item with 12.9k-14.1k executable statements and 3.7k-4.0k tests depending on the version. There are 1-8 faulty statements in each version.

10.4.2 Results and quantitative evaluation

Data preparation

The subject program has 27 buggy versions, but bug 21 is marked as deprecated, resulting in 26 bugs we could work with. Table 10.3 shows whether the bug was included or excluded in our examination (column "Inc."), as well as, the reason behind the exclusion (column "Reason"), which is described in detail below.

Inc.	Reason	Bugs
✗	*Bad Slice*	{1, 2, 7, 8, 11, 13, 19, 20}
	Omission	{3, 6, 14, 15, 18, 24, 25, 27}
	Exception	{5}
✔	-	{4, 9, 10, 12, 16, 17, 22, 23, 26}

Table 10.3: Properties of the investigated bugs

We excluded 8 bugs because their fix contains only added statements (*Omission*). These statements are missing from the buggy versions, *i.e.*, neither the coverage-based nor the

slice-based spectra can point to them. Another reason was that the slice did not contain the faulty element, but the test cases covered it (*Bad Slice*). In one case, the test failed before the assertions due to an exception (*Exception*), which made the dynamic slice computation impossible. In 7 cases, we could not find out why the slice did not contain the faulty (and covered) statements. In two cases the faulty statements were spread across multiple lines, and the reported location of the fault (determined from change sets) did not match the location reported by the slicer (the first line of the multiline statements). We corrected these two computations by hand. As a result, we had 9 bugs for which we could compute meaningful slices.

Comparing spectra

As discussed above, in theory, the slice spectra should be a subset of coverage spectra. Unfortunately, with our toolset, this turned out to not hold in about 60% of the slices. We checked the reasons why slices can be inaccurate in this sense, and we found two main reasons. One is that the slicer does not slice into Java library methods, and while losing dependences through them it also follows some false dependences. The other cause is that the slicer and the coverage tool can report different source code lines for multiline expressions.

To approximate the theoretical parameter p from Section 10.3 on our subject program, we had to deal with this inaccuracy of the slicing tool. We simply ignored statements that are not covered but are part of the slice, hence we computed the size of the intersection of the coverage and slice sets and divided it with the coverage size. We then averaged this value for every test case. Table 10.4 shows the resulting values along with some other statistics.

Bug	Avg.	Med.	Min.	Max.	Std.dev.
4	0.4259	0.4167	0.0	1.0	0.2834
9	0.4321	0.4286	0.0	1.0	0.2847
10	0.4321	0.4286	0.0	1.0	0.2849
12	0.4335	0.4344	0.0	1.0	0.2836
16	0.4369	0.4378	0.0	1.0	0.2828
17	0.4345	0.4357	0.0	1.0	0.2811
22	0.4444	0.4463	0.0	1.0	0.2796
23	0.4442	0.4463	0.0	1.0	0.2798
26	0.4634	0.4706	0.0	1.0	0.2750

Table 10.4: Measured average slice sizes with respect to the coverage on the subject programs

The average slice size varies between 42.5% and 46.4% of the coverage. However, we expect that the correct values would be even lower because the slicer seems to be over-approximating as was the case with the not covered elements. Looking at the relationship between the expected fault localization scores for the two types of matrices in Section 10.3, we see this value to be a significant factor responsible for the differences in the final rankings.

Comparison of SBFL effectiveness

We use the *Expense* metric to assess the effectiveness of fault localization, which is essentially the absolute average rank of faulty statements (see Chapter 8). Table 10.5 shows these results for different formulae on the included bugs for both kinds of spectra. We can see that in most cases the slice-based results are better, sometimes notably. There were a couple of cases where the results were the same, and in two cases (bugs 9 and 16) the coverage-based

beszedes_242_24

method performed better. For bugs 4 and 17, the faulty element was placed on the highest rank position by the slice-based approach but since it was tied with several other elements, the value shown is not 1.

Bug	*B*		*D*		*J*		*O*		*S*		*T*	
	cov.	slice	cov.	slice	cov.	slice	cov.	slice	cov.	slice	cov.	slice
4	23.5	1.5	20.5	1.5	23.5	1.5	23.5	1.5	23.5	1.5	23.5	1.5
9	3	11	2	11	3	11	3	11	3	11	3	11
10	21.5	9	11.5	10	19.5	10	15.5	10	19.5	10	21	9
12	2.5	2.5	29.5	2.5	8	2.5	5	2.5	8	2.5	2.5	2.5
16	8	10	8	11	8	11	8	11	8	11	8	10
17	5	5	5	5	5	5	5	5	5	5	5	5
22	23.5	14.5	23.5	13.5	23.5	14.5	23.5	13.5	23.5	14.5	23.5	14.5
23	25.5	20.5	25.5	20.5	25.5	20.5	25.5	20.5	25.5	20.5	25.5	20.5
26	270.5	20.5	60.5	10.5	132.5	19.5	71.5	15.5	132.5	19.5	156.5	20.5
avg	**42.6**	**10.5**	**20.7**	**9.5**	**27.6**	**10.6**	**20.1**	**10.1**	**27.6**	**10.6**	**29.8**	**10.5**

Table 10.5: Fault localization effectiveness (Expense values). *B: Barinel, D: DStar, J: Jaccard, O: Ochiai, S: Sørensen-Dice, T: Tarantula*

The last row represents the overall average ranks, from which we can infer the degree of improvement in general. The difference is notable in all cases (between 10 and 32), *i.e.*, the slice-based method ranks the faulty statement higher in the suspiciousness list by 10-32 positions. Overall, *DStar* performed best (9.5), followed by *Ochiai* (10.1) but the other formulae produced similar results.

Although we cannot draw definitive conclusions from the results due to the small sample size, we can say that the overall performance of slice-based SBFL compared to coverage-based one is positive: on this subject program it improved the ranking position of the faulty elements notably, in many cases achieving the top positions. Only two bugs showed negative results, and we attribute these to inaccuracies in the slicing tool.

10.4.3 Qualitative evaluation

We examined each investigated bug in detail to find out why coverage-based SBFL produced sub-optimal results compared to slice-based SBFL (in the cases when the result was negative, the reasons for it as well). The focus of the comparison was primarily on the spectrum metrics, rather than the score and rank values. (In this section, the item names are relative to the *org.joda.time* package.)

time-4: Test *TestPartial_Basics.textWith3* fails here because no exception is thrown. The reason for this is that the *Partial.with(DateTimeFieldType, int)* method calls a wrong constructor (in line 464). The slice of the test case contains 4 statements (lines 430, 431, 464, 466), while the coverage has 24 additional ones. In addition, 3 utility statements are covered by only the faulty test case. This resulted in the score of the faulty statement ranking at 23.5 on average, together with 14 other statements. As the mentioned utility and additional statements are omitted by the slicer, their ef' values were reduced, allowing the formulae to rank lines 464 and 466 in the first position with the same score.

time-9: Here *DateTimeZone:264* has a slice-based rank 11 with ef'=1 and ep'=5, and shares these values with 8 statements including *DateTimeZone:604*. However, *DateTime-*

beszedes_242_24

Zone:604 was covered by not 5 but *ep*=11 passed tests. For example, the slice of *TestDateTimeZone.testSerialization2* does not include the above-mentioned statement but covers it.

The computed slice of the test contains only the statements of the test except for the 1011th, 1006th, and 1000th instructions. As `oos` is of a stock Java class type, the slicer does not analyze its method call in line 1004 but seems to treat it as a definition of a `zone` instead of treating it as a use. This can be the reason why line 1000 is (incorrectly) not included in the slice, so the ep' values of the statements accessible through it (e.g. *DateTimeZone:604*) are not increased by the test result of *testSerialization2*, thus, the scores are not decreased, i.e. statements are ranked as more suspicious. Due to cases like this, the results of the (passed) tests will not be counted for certain statements and, therefore, it is possible that the coverage-based result will be better than the slice-based one.

time-10: The second assert fails in both failing test cases. While the bug is covered by both test cases, it is contained only in the slice of the first, not failing assert of test case *TestDays.testFactory_daysBetween_RPartial_MonthDay*. We could not find the reasons for this omission, it is probably due to a slicer issue.

time-12: We found that the slices of the asserts contain only a few statements, so there is a non-negligible difference between the slice-based and coverage-based spectrum metrics. The reason for the difference is that statement *LocalDateTime:612* (in the *isSupported()* method) has *ef*=4, which makes it among the most suspicious elements according to the coverage-based algorithm, while the ef'=0, which puts it in the bottom of the suspicion ranking of the slice-based approach. The *ef*, *ep*, and *nf* values of the faulty statement *LocalDate:211* are the same in the two spectra, so it precedes several statements in the ranking that are more suspicious than it according to the coverage-based algorithm but ranked lower by their slice-based spectra.

time-16: We examined the tests related to the fault at *format.DateTimeFormatter:709* in method *parseInto()*. In the case of the coverage-based measurement, test *TestDateTimeFormatter.testParseInto_monthOnly* covers the buggy statement, however, the slice-set belonging to the assert in line 869 includes only the first line of the *parseInto()* method (line 698). This is interesting because the return value of *f.parseInto(result, "5", 0)* function call (and the other statements affecting it) was omitted due to a probable slicer problem and this misled the slice-based FL algorithms.

time-17: Bug 17 has 9 instructions with the same highest score and average rank of 5. These 9 instructions belong to 3 methods, 2 of which (*DateTime.withEarlierOffsetAtOverlap()* and *DateTime.withLaterOffsetAtOverlap()*) are simple sequential methods, and *DateTimeZone.adjustOffset(long, boolean)* is a bit longer having a decision. All instructions but the alternative return of the last method are both covered and part of the slices. These 9 instructions are always executed together. As a result, their two spectra are identical, resulting in the same scores and ranks.

time-22: 8 passed tests have incorrectly computed slices, *e.g.*, the slice of the assert in line *TestMutablePeriod_Basics:451* contains only 1 statement, and does not include the constructor of class *MutablePeriod* and thus (incorrectly) could not reach the buggy line *base.BasePeriod:222*. Yet, the scores and ranks of the faulty instruction improve, because its $ep' < ep$ due to the bad computations.

time-23: Test *TestDateTimeZone.testForID_String_old* fails when it checks the contents of a map previously filled by the *DateTimeZone.getConvertedID(String)* method. The coverage-based calculations rank *DateTimeZone:314* to first place as it is executed by the failing and a single passing test case. Then a tie with 48 elements follows, including the faulty lines, lines filling the map, and other lines of the *DateTimeZone.getDefault()* method. All of these are covered by the sole failing and multiple passing tests. The slicer is unable to decompose which items in the map are used in the failing test, keeping the whole map

filling section in the slice. However, it is able to omit the lines of the *getConvertedID(String)* method from all test cases except for the passing one (and it does the same for two additional instructions of *getDefault()*), while keeping them in the slice of the failing test case. This reduction of ep' results in a tie of 40 elements, including all faulty lines, in the first place (with an average rank of 20.5).

time-26: There are 8 failing test cases and 8 modified lines for fixing the same bug. The fix replaces the call to *convertLocalToUTC(long, boolean)* with the newly added *convertLocalToUTC(long, boolean, long)*. However, only one buggy line, *chrono.ZonedChronology:467* is exercised by the tests. It is covered by all faulty test cases but contained only in 4 of their slices, while 175 passing tests also cover the line but only 49 slices of passing tests do the same. Thus, while ef is higher than ef', ep is much higher than ep', causing our metrics to give a higher score to the faulty instruction. The slicer also eliminates many instructions from the spectrum of faulty test cases. While coverage shows 1615 instructions with non-zero ef, there are only 872 instructions with non-zero ef'. This also helps improve the rank of the faulty statement.

The cases when slice-based SBFL did not overtake coverage-based SBFL were due to imperfections or defects in the slicer. In one case (Bug 10) the slicer seemed to miscalculate slices for the faulty test, yet the ranks were still able to improve. In all other cases, the slice-based spectrum worked as expected, and either raised the score of the faulty element or lowered the score of non-faulty statements.

10.5 Conclusions

We argue that using code coverage in the SBFL spectrum is such an over-approximation that it could impair the achievable effectiveness of SBFL to a level that makes it not useful in practice. Code coverage is, in essence, a proxy to the dynamic backward program slice which captures the code elements with an actual influence on the defective behavior. In Section 10.3, we showed that, in principle, the coverage-based SBFL will necessarily produce worse code element ranking compared to the slice-based spectrum because a correctly computed backward dynamic slice is a subset of the coverage. Furthermore, it is expected that the rate of imprecision of the coverage will directly and severely influence the performance of the suspiciousness formulae.

The preconditions set in Section 10.3 will not hold in many practical situations, but as our experimental study showed, even under imperfect conditions, the benefits of slicing are clearly visible over coverage. In fact, the case study showed that the slice is less than half of the coverage and that the overall localization effectiveness is typically much better with the slice-based SBFL than with the coverage-based one. The study also highlighted several deficiencies in the slicing tool used, but despite of these, the results were positive. We cannot rule out the possibility that the coverage-based measurement had flaws too, but since we used a well-established technique and mature tools, we attribute these errors mostly to the slicer tool. Finally, through real examples, the qualitative evaluation showed why the coverage-based spectrum was detrimental. So, the question is, why are we still using code coverage as the basis for SBFL?

One part of the answer is that computing coverage is simple using existing tools, and the SBFL implementations are straightforward. But, despite its several decades-long history, program slicing is still a difficult area, and practically usable tools are not easy to develop. Slicers that produce more precise results often require huge computation resources, while sub-optimal and approximate slicing algorithms may be very imprecise. Another challenge with the slicing approach is that a slicing criterion is needed for each test case, which is often not trivial to determine in practice. Our case study dealt with unit tests, and there

the assertions could serve this purpose, but when the tests are higher level or more complex, this can be a challenge.

Furthermore, there are many technical limitations of practical program slicing tools, which increases the risk of their use in fault localization, and this risk is much higher than with the coverage-based tools. If we fail to tell whether an instruction is covered, it will affect only that particular instruction; but when a dependency is missed or falsely added during slicing, it can affect a significant amount of dependent instructions. Difficulties often relate to special features like multi-threading or exception handling, dependences from 3rd party or mixed language components, or unobservable artifacts e.g., files and databases. Many practical tools either miss dependences, leading to false results, or employ a conservative approach and imprecise slices, the very essence of the disadvantage of the coverage-based approach discussed in this research.

On the positive side, it must be noted that program slicing also carries structural information about the program and the computation paths. In a simple SBFL approach, the programmers need to walk down the list of suspicious elements that often do not have any relationship to each other. However, when using program slicing, the dependences can be followed, which reflects the actual computations, and this information may be a big aid during debugging [108]. A hybrid approach, such as the one proposed by Soremekun *et al.* [121], could be considered as well.

In summary, we are convinced that this area needs more research. Slice-based spectra should once more attain much higher focus, such as research on more efficient approximate or hybrid slicing. There are several open questions to be addressed. One of the possible directions is to investigate the actual performance of various slicing algorithms and tools, and how they could be improved to better serve this application. A good alternative would be to experiment with hybrid or approximate slicing algorithms, as some related techniques did [101]. On a more conceptual level, one should consider specific aspects of the SBFL process and slicing, such as the issues of multiple faults, ranking ties, the impact of various slice concepts, as well as the question of the slicing criterion in different types of tests.

Contribution

This chapter is based on the publication:

[26] Péter Attila Soha, Tamás Gergely, Ferenc Horváth, Béla Vancsics, **Árpád Beszédes**. *A Case Against Coverage-Based Program Spectra.* In Proceedings of the 16th IEEE International Conference on Software Testing, Verification and Validation (ICST 2023), pages: 13-24, Dublin, Ireland, April 2023.

Despite its recent publication, this paper already received one independent citation. Our results can serve as evidence for the serious drawback of coverage-based SBFL, and this has not been systematically verified before, hence we believe that the investigation of program slicing for SBFL may receive bigger attention in the future.

The theoretical model that describes the relationship between coverage and slice-based SBFL and the level of difference in their results with respect to the inaccuracy of code coverage are mostly my contribution, while the use of the spectrum matrix with dynamic slices computed from the asserts in test cases, and its quantitative and qualitative empirical comparison to the coverage-based approach are joint work.

11 Conclusion

In this thesis, various topics related to program dependence analysis have been addressed with mention of different applications including debugging, testing, and other areas of software development and maintenance. A large portion of the presented methods are some form of dynamic program analysis, which in general means more precise results compared to static analysis. However, additional challenges arise due to the need for analyzing (potentially very large) execution traces and dynamic relations between program elements.

Our findings related to dynamic program slicing and the dynamic function coupling show that it is possible to design cost-effective dynamic dependence algorithms by either using efficient data structures or performing the analysis on a lower granularity level.

Results from dependence cluster analysis highlight the need to continue research in the direction of more efficient linchpin detection and program refactoring techniques to reduce the burden of large dependence clusters. Static analyses inherently include imprecision, so it is not known how much these imprecisions are the cause for the formation of dependence clusters. Using dynamic dependence analysis as the underlying dependence for cluster computation instead of static analysis could provide interesting insights about this topic.

Results in this thesis related to Spectrum-Based Fault Localization aim at improving the efficiency of fault localization, but there are a lot of possibilities to explore in this area. Using additional context information beyond the traditional coverage-based spectra seems to be inevitable, which is supported by multiple findings in this thesis (using call chains, call frequencies and program slices). Involving the user knowledge into the process in an interactive fault localization process is another promising direction [16]. Apart from that, it is important to continue work on the development of practical fault localization tools in development environments and debuggers [22, 23, 27]. Several challenges have already been identified which make this task difficult [15, 17, 20, 34], and addressing these would bring us closer to the everyday use of this technique.

Acknowledgments

This research was supported by the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory; and by the national Project no. TKP2021-NVA-09 which has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

beszedes_242_24

Bibliography

Referenced publications of the author

[1] Árpád Beszédes. Global dynamic slicing for the C language. *Acta Polytechnica Hungarica*, 12(1):117–136, 2015.

[2] Árpád Beszédes, Csaba Faragó, Zsolt Mihály Szabó, János Csirik, and Tibor Gyimóthy. Union slices for program maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 12–21. IEEE Computer Society, October 2002.

[3] Árpád Beszédes, Tamás Gergely, Szabolcs Faragó, Tibor Gyimóthy, and Ferenc Fischer. The Dynamic Function Coupling metric and its use in software evolution. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 103–112, March 2007.

[4] Árpád Beszédes, Tamás Gergely, and Tibor Gyimóthy. Graph-less dynamic dependence-based dynamic slicing algorithms. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'06)*, pages 21–30, September 2006.

[5] Árpád Beszédes, Tamás Gergely, Judit Jász, Gabriella Tóth, Tibor Gyimóthy, and Václav Rajlich. Computation of Static Execute After relation with applications to software maintenance. In *Proceedings of the 2007 IEEE International Conference on Software Maintenance (ICSM'07)*, pages 295–304, 2007.

[6] Árpád Beszédes, Tamás Gergely, Zsolt Mihály Szabó, János Csirik, and Tibor Gyimóthy. Dynamic slicing method for maintenance of large C programs. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR 2001)*, pages 105–113. IEEE Computer Society, March 2001.

[7] Árpád Beszédes, Tibor Gyimóthy, Gábor Lóki, Gergely Diós, and Ferenc Kovács. Using backward dynamic program slicing to isolate influencing statements in GDB. In *Proceedings of the 2007 GCC Developers' Summit*, pages 21–30, July 2007.

[8] Árpád Beszédes, Ferenc Horváth, Massimiliano Di Penta, and Tibor Gyimóthy. Leveraging contextual information from function call chains to improve fault localization. In *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'20)*, pages 468–479, February 2020.

[9] Árpád Beszédes, Lajos Schrettner, Béla Csaba, Tamás Gergely, Judit Jász, and Tibor Gyimóthy. Empirical investigation of SEA-based dependence cluster properties. In *Proceedings of the 13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'13)*, pages 1–10, September 2013.

[10] Árpád Beszédes, Lajos Schrettner, Béla Csaba, Tamás Gergely, Judit Jász, and Tibor Gyimóthy. Empirical investigation of SEA-based dependence cluster properties. *Science of Computer Programming*, 105(0):3 – 25, 2015. Special Issue on SCAM'13.

[11] David Binkley, Árpád Beszédes, Syed Islam, Judit Jász, and Béla Vancsics. Uncovering dependence clusters and linchpin functions. In *Proceedings of the 31th IEEE International Conference on Software Maintenance and Evolution (ICSME'15)*, pages 141–150, September 2015.

[12] Viktor Csuvik, Roland Aszmann, Árpád Beszédes, Ferenc Horváth, and Tibor Gyimóthy. On the stability and applicability of deep learning in fault localization. In *Proceedings of the 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'24)*, pages 546–555, March 2024.

[13] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus – reverse engineering tool and schema for C++. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 172–181. IEEE Computer Society, October 2002.

[14] Tibor Gyimóthy, Árpád Beszédes, and István Forgács. An efficient relevant slicing method for debugging. *Lecture Notes in Computer Science*, 1687:303–321, 1999.

[15] Ferenc Horváth, Roland Aszmann, Péter Attila Soha, Árpád Beszédes, and Tibor Gyimóthy. Context switch sensitive fault localization. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (EASE'24)*, pages 110–119, June 2024.

[16] Ferenc Horváth, Árpád Beszédes, Béla Vancsics, Gergő Balogh, László Vidács, and Tibor Gyimóthy. Using contextual knowledge in interactive fault localization. *Empirical Software Engineering*, 27(150):69, 2022.

[17] Ferenc Horváth, Tamás Gergely, Árpád Beszédes, Dávid Tengeri, Gergő Balogh, and Tibor Gyimóthy. Code coverage differences of Java bytecode and source code instrumentation tools. *Software Quality Journal*, 27(1):79–123, 2019.

[18] Judit Jász, Árpád Beszédes, Tibor Gyimóthy, and Václav Rajlich. Static Execute After/Before as a replacement of traditional software dependencies. In *Proceedings of the 2008 IEEE International Conference on Software Maintenance (ICSM'08)*, pages 137–146, 2008.

[19] Judit Jász, Lajos Schrettner, Árpád Beszédes, Csaba Osztrogonác, and Tibor Gyimóthy. Impact analysis using Static Execute After in WebKit. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR'12)*, pages 95–104, March 2012.

[20] Qusay Idrees Sarhan and Árpád Beszédes. A survey of challenges in Spectrum Based Software Fault Localization. *IEEE Access*, 10:10618–10639, 2022.

[21] Qusay Idrees Sarhan, Tamás Gergely, and Árpád Beszédes. Systematically generated formulas for spectrum-based fault localization. In *Proceedings of the 2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 344–352, April 2023.

[22] Qusay Idrees Sarhan, Hassan Bapeer Hassan, and Árpád Beszédes. SFLaaS: Software fault localization as a service. In *Proceedings of the 16th IEEE International Conference on Software Testing, Verification and Validation (ICST'23)*, pages 467–469, April 2023.

[23] Qusay Idrees Sarhan, Attila Szatmári, Rajmond Tóth, and Árpád Beszédes. CharmFL: A fault localization tool for Python. In *Proceedings of the 21st IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'21)*, pages 114–119, September 2021.

[24] Lajos Schrettner, Judit Jász, Tamás Gergely, Árpád Beszédes, and Tibor Gyimóthy. Impact analysis in the presence of dependence clusters using Static Execute After in WebKit. In *Proceedings of the 12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'12)*, pages 24–33, September 2012.

[25] Lajos Schrettner, Judit Jász, Tamás Gergely, Árpád Beszédes, and Tibor Gyimóthy. Impact analysis in the presence of dependence clusters using Static Execute After in WebKit. *Journal of Software: Evolution and Process*, 26(6):569–588, June 2014. Special Issue on SCAM'12.

[26] Péter Attila Soha, Tamás Gergely, Ferenc Horváth, Béla Vancsics, and Árpád Beszédes. A case against coverage-based program spectra. In *Proceedings of the 16th IEEE International Conference on Software Testing, Verification and Validation (ICST'23)*, pages 13–24, April 2023.

[27] Attila Szatmári, Qusay Idrees Sarhan, Péter Attila Soha, Gergő Balogh, and Árpád Beszédes. On the integration of spectrum-based fault localization tools into IDEs. In *Proceedings of the 1st ACM/IEEE Workshop on Integrated Development Environments (IDE'24)*, pages 24–29, April 2024.

[28] Attila Szegedi, Tamás Gergely, Árpád Beszédes, Tibor Gyimóthy, and Gabriella Tóth. Verifying the concept of union slices on Java programs. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 233–242, March 2007.

[29] Dávid Tengeri, Árpád Beszédes, Dávid Havas, and Tibor Gyimóthy. Toolset and program repository for code coverage-based test suite analysis and manipulation. In *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'14)*, pages 47–52, September 2014.

[30] Dávid Tengeri, Ferenc Horváth, Árpád Beszédes, Tamás Gergely, and Tibor Gyimóthy. Negative effects of bytecode instrumentation on Java source code coverage. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, pages 225–235, March 2016.

[31] Béla Vancsics, Ferenc Horváth, Attila Szatmári, and Árpád Beszédes. Call frequency-based fault localization. In *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'21)*, pages 365–376, March 2021.

[32] Béla Vancsics, Ferenc Horváth, Attila Szatmári, and Árpád Beszédes. Fault localization using function call frequencies. *The Journal of Systems and Software*, 193:111429, 2022.

beszedes_242_24

[33] László Vidács, Árpád Beszédes, Dávid Tengeri, István Siket, and Tibor Gyimóthy. Test suite reduction for fault detection and localization: A combined approach. In *Proceedings of the CSMR-WCRE 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE'14)*, pages 204–213, February 2014.

[34] Dániel Vince, Attila Szatmári, Ákos Kiss, and Árpád Beszédes. Division by zero: Threats and effects in Spectrum-Based Fault Localization formulas. In *Proceedings of the 22nd IEEE International Conference on Software Quality, Reliability, and Security (QRS'22)*, pages 221–230, December 2022.

beszedes_242_24

Other references

[35] Defects4j's github repository. `https://github.com/rjust/defects4j/releases/tag/v2.0.0`. Accessed: 2022-10-21.

[36] Slicer4j's github repository. `https://github.com/resess/Slicer4J`. Accessed: 2024-10-05.

[37] Rui Abreu, Alberto Gonzalez-Sanchez, and Arjan JC van Gemund. Exploiting count spectra for bayesian fault localization. Proceedings of the 6th International Conference on Predictive Models in Software Engineering, pages 1–10, New York, NY, USA, 2010. ACM, Association for Computing Machinery.

[38] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[39] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 89–98, 2007.

[40] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. Spectrum-based multiple fault localization. Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, pages 88–99. IEEE, IEEE Press, 2009.

[41] Mithun Acharya and Brian Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *Proceedings of the 33rd ACM SIGSOFT International Conference on Software Engineering (ICSE)*, pages 746–765, 2011.

[42] Pragya Agarwal and Arun Agrawal. Fault-localization techniques for software systems. *ACM SIGSOFT Software Engineering Notes*, 39:1–8, 09 2014.

[43] Hiralal Agrawal. *Towards Automatic Debugging of Computer Programs*. PhD thesis, Purdue University, 1992.

[44] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, number 6 in SIGPLAN Notices, pages 246–256, White Plains, New York, June 1990.

[45] Khaled Ahmed, Mieszko Lis, and Julia Rubin. Slicer4j: A dynamic slicer for java. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1570–1574. ACM, 2021.

[46] Khalid Alemerien and Kenneth Magel. Examining the effectiveness of testing coverage tools: An empirical study. *International Journal of Software Engineering and Its Applications*, 8(5):139–162, 2014.

[47] Elton Alves, Milos Gligoric, Vilas Jagannath, and Marcelo d'Amorim. Fault-localization using dynamic slicing and change impact analysis. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 520–523. IEEE, 2011.

[48] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 432–441, May 2005.

[49] Tien-Duy B Le, David Lo, Claire Le Goues, and Lars Grunske. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 177–188. ACM, 2016.

[50] Benoit Baudry, Franck Fleurey, and Yves Le Traon. Improving test suites for efficient fault localization. In *28th international conference on Software engineering*, ICSE '06, pages 82–91. ACM, 2006.

[51] Boris Beizer. *Software Testing Techniques*. John Wiley and Sons Inc., 605 Third Ave. New York, NY, United States, 1990.

[52] David Binkley. Source code analysis: A road map. In *Proceedings of 2007 Future of Software Engineering (FOSE'07)*, pages 104–119. IEEE Computer Society, 2007.

[53] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Zheng Li. Efficient identification of linchpin vertices in dependence clusters. *ACM Trans. Program. Lang. Syst.*, 35(2):7:1–7:35, July 2013.

[54] David Binkley and Mark Harman. Locating dependence clusters and dependence pollution. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM'05)*, pages 177–186. IEEE Computer Society, September 2005.

[55] David Binkley and Mark Harman. Identifying 'linchpin vertices' that cause large dependence clusters. In *Proceedings of the Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'09)*, pages 89–98, 2009.

[56] David Binkley, Mark Harman, Youssef Hassoun, Syed Islam, and Zheng Li. Assessing the impact of global variables on program dependence and dependence clusters. *Journal of Systems and Software*, 83(1):96–107, 2010.

[57] David W Binkley and Mark Harman. A survey of empirical results on program slicing. *Adv. Comput.*, 62(105178):105–178, 2004.

[58] Rex Black, Erik van Veenendaal, and Dorothy Graham. *Foundations of Software Testing: ISTQB Certification*. Cengage Learning, 2012.

[59] Sue Black, Steve Counsell, Tracy Hall, and David Bowes. Fault analysis in OSS based on program slicing metrics. In *Proceedings of the EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 3–10. IEEE Computer Society, 2009.

[60] Shawn A. Bohner and Robert S. Arnold, editors. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[61] Clover homepage. `https://www.atlassian.com/software/clover/`, 2024. Last visited: 2024-10-05.

[62] Cobertura. `http://cobertura.github.io/cobertura/`. Last visited: 2019-10-25.

[63] Higor Amario de Souza, Marcos Lordello Chaim, and Fabio Kon. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *CoRR*, abs/1607.04347, 2016.

[64] W. Eric Wong and Yu Qi. Bp neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering*, 19, 11 2011.

[65] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its uses in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[66] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419. ACM, 2011.

[67] Carlos Galindo, Sergio Pérez, and Josep Silva. Slicing unconditional jumps with unnecessary control dependencies. In *Logic-Based Program Synthesis and Transformation: 30th International Symposium, LOPSTR 2020, Bologna, Italy, September 7–9, 2020, Proceedings*, page 293–308. Springer-Verlag, 2020.

[68] GCC, the GNU Compiler Collection. `http://gcc.gnu.org/`, 2014. Last visited: 2014-06-12.

[69] GitHub homepage. `https://github.com/`, 2024. Last visited: 2024-10-05.

[70] R. Gopal. Dynamic program slicing based on dependence relations. In *Proceedings of the Conference on Software Maintenance*, pages 191–200, Sorrento, Italy, 1991. IEEE Computer Society Press.

[71] Homepage of GrammaTech's CodeSurfer. `http://www.grammatech.com/products/codesurfer`.

[72] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, April 2001.

[73] Brent Hailpern and Peter Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, December 2001.

[74] Clemens Hammacher. Design and implementation of an efficient dynamic slicer for Java. Bachelor's Thesis, nov 2008.

[75] Mark Harman, David Binkley, Keith Gallagher, Nicolas Gold, and Jens Krinke. Dependence clusters in source code. *ACM Transactions on Programming Languages and Systems*, 32(1):1–33, November 2009.

[76] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194, 2000.

[77] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An empirical investigation of program spectra. In *Proc. of the 1998 ACM SIGPLAN-SIGSOFT workshop PASTE '98*, pages 83–90. ACM, 1998.

[78] Hongdou He, Jiadong Ren, Guyu Zhao, and Haitao He. Enhancing spectrum-based fault localization using fault influence propagation. *IEEE Access*, 8:18497–18513, 2020.

[79] Simon Heiden, Lars Grunske, Timo Kehrer, Fabian Keller, Andre Van Hoorn, Antonio Filieri, and David Lo. An evaluation of pure spectrum-based fault localization techniques for large-scale software systems. *Software: Practice and Experience*, 49(8):1197–1224, 2019.

[80] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[81] JaCoCo homepage. `http://eclemma.org/jacoco/`, 2024. Last visited: 2024-10-05.

[82] Javassist. `http://jboss-javassist.github.io/javassist/`. Last visited: 2019-10-25.

[83] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, Sept 2011.

[84] Siyuan Jiang, Collin McMillan, and Raul Santelices. Do programmers do change impact analysis in debugging? *Empirical Software Engineering*, 22, 04 2017.

[85] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proc. of International Conference on Automated Software Engineering*, pages 273–282. ACM, 2005.

[86] JUnit homepage. `http://junit.org/`, 2024. Last visited: 2024-10-05.

[87] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2014.

[88] Mariam Kamkar, Nahid Shahmehri, and Peter Fritzson. Interprocedural dynamic slicing. In *Proceedings of the 4th International Conference on Programming Language Implementation and Logic Programming (PLILP'92)*, volume 631 of *Lecture Notes in Computer Science*, pages 370–384. Springer-Verlag, 1992.

[89] Fabian Keller, Lars Grunske, Simon Heiden, Antonio Filieri, Andre van Hoorn, and David Lo. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), pages 114–125. IEEE, IEEE Press, 2017.

[90] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*, pages 165–176, New York, New York, USA, 2016. ACM Press.

[91] Bogdan Korel. Computation of dynamic program slices for unstructured programs. *IEEE Transactions on Software Engineering*, 23(1):17–34, January 1997.

[92] Bogdan Korel and Janusz W. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[93] Bogdan Korel and Janusz W. Laski. Dynamic slicing in computer programs. *The Journal of Systems and Software*, 13(3):187–195, 1990.

[94] T. B. Le, F. Thung, and D. Lo. Theory and practice, do they match? a case with spectrum-based fault localization. Proceedings of the 2013 IEEE International Conference on Software Maintenance, pages 380–383. IEEE, IEEE Press, 2013.

[95] Nan Li, Xin Meng, J. Offutt, and Lin Deng. Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools (experience report). In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 380–389, Nov 2013.

[96] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 169–180. ACM, ACM, 2019.

[97] Yihan Li and Chao Liu. Effective fault localization using weighted test cases. *J. Softw.*, 9(8):2112–2119, 2014.

[98] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. *SIGPLAN Not.*, 40(6):15–26, June 2005.

[99] Lucia Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi. Extended comprehensive study of association measures for fault localization. *Journal of software: Evolution and Process*, 26(2):172–219, 2014.

[100] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[101] Xiaoguang Mao, Yan Lei, Ziying Dai, Yuhua Qi, and Chengsong Wang. Slice-based statistical fault localization. *Journal of Systems and Software*, 89:51–62, 2014.

[102] Wes Masri, Rawad Abou-Assi, Marwa El-Ghali, and Nour Al-Fatairi. An empirical study of the factors that reduce the effectiveness of coverage-based fault localization. In *Proceedings of the 2nd International Workshop on Defects in Large Software Systems*, DEFECTS '09, pages 1–5, New York, NY, USA, 2009. ACM.

[103] Wes Masri and Rawad Abou Assi. Prevalence of coincidental correctness and mitigation of its impact on fault localization. *ACM Trans. Softw. Eng. Methodol.*, 23(1):8:1–8:28, February 2014.

[104] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):1–32, 2011.

[105] N Neelofar, Lee Naish, and Kotagiri Ramamohanarao. Spectral-based fault localization using hyperbolic function. *Software: Practice and Experience*, 48(3):641–664, 2018.

[106] Thomas Ostrand. White-box testing. *Encyclopedia of Software Engineering*, 2002.

[107] Priya Parmar and Miral Patel. Software fault localization: A survey. *International Journal of Computer Applications*, 154(9):6–13, 2016.

[108] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209, New York, NY, USA, 2011. ACM.

[109] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. page 609–620, 2017.

[110] Václav Rajlich and Prashant Gosavi. Incremental change in object-oriented programming. *IEEE Software*, 21(4):62–69, 2004.

[111] S. Rayadurgam and M.P.E. Heimdahl. Coverage based test-case generation using model checkers. In *Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings. Eighth Annual IEEE International Conference and Workshop on the*, pages 83–91, 2001.

[112] Sofia Reis, Rui Abreu, and Marcelo d'Amorim. Demystifying the combination of dynamic slicing and spectrum-based fault localization. In *IJCAI*, pages 4760–4766, 2019.

[113] Xiaoxia Ren and Barbara G Ryder. Heuristic ranking of java program edits for fault localization. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 239–249, 2007.

[114] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, pages 30–39. IEEE Computer Society, 2003.

[115] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. *ACM SIGSOFT Software Engineering Notes*, 22(6):432–449, November 1997.

[116] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[117] Gregg Rothermel, Mary Jean Harrold, Jeffery von Ronne, and Christie Hong. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12(4):219–249, 2002.

[118] Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. Do stack traces help developers fix bugs? Proceedings of the 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), pages 118–121. IEEE, 05 2010.

[119] SED Java instrumenter. `https://github.com/sed-szeged/java-instrumenter`, 2024. Last visited: 2024-10-05.

[120] SoDA library. `http://soda.sed.hu`, 2014. Last visited: 2014-06-11.

[121] Ezekiel Soremekun, Lukas Kirschner, Marcel Böhme, and Andreas Zeller. Locating faults with program slicing: an empirical analysis. *Empirical Software Engineering*, 26(3):1–45, 2021.

[122] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. Proceedings of the 2013 International Symposium on Software Testing and Analysis, page 314–324, New York, NY, USA, 2013. ACM, Association for Computing Machinery.

[123] Attila Szegedi and Tibor Gyimóthy. Dynamic slicing of Java bytecode programs. In *Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, pages 35–44. IEEE Computer Society, September 2005.

[124] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[125] Macario Polo Usaola and Pedro Reales Mateo. Mutation testing cost reduction techniques: A survey. *IEEE Software*, 27(3):80–86, 2010.

[126] Jeffrey M. Voas. Pie: A dynamic failure-based technique. *IEEE Trans. Softw. Eng.*, 18(8):717–727, August 1992.

[127] The WebKit open source project. http://www.webkit.org/. Last visited: 2013-01-25.

[128] WebKit Bugzilla homepage. https://bugs.webkit.org/. Last visited: 2013-01-25.

[129] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[130] W Eric Wong and Vidroha Debroy. A survey of software fault localization. *Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45*, 9, 2009.

[131] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, 2013.

[132] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[133] W Eric Wong and Yu Qi. Effective program debugging based on execution slices and inter-block data dependency. *Journal of Systems and Software*, 79(7):891–903, 2006.

[134] W Eric Wong, Tatiana Sugeta, Yu Qi, and Jose C Maldonado. Smart debugging software architectural design in sdl. *Journal of Systems and Software*, 76(1):15–28, 2005.

[135] Xin Xia, Lingfeng Bao, David Lo, and Shanping Li. "automated debugging considered harmful" considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. Proceedings of the 32nd International Conference on Software Maintenance and Evolution (ICSME), pages 267–278. IEEE, IEEE Press, 2016.

[136] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Trans. Softw. Eng. Methodol.*, 22(4):31:1–31:40, October 2013.

[137] Xiaofeng Xu, Vidroha Debroy, W. Eric Wong, and Donghui Guo. Ties within fault localization rankings: Exposing and addressing the problem. *International Journal of Software Engineering and Knowledge Engineering*, 21:803–827, 2011.

[138] Qian Yang, J Jenny Li, and David M Weiss. A survey of coverage-based testing tools. *The Computer Journal*, 52(5):589–597, 2009.

[139] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[140] Shin Yoo. Evolving human competitive spectra-based fault localisation techniques. Proceedings of the 2012 International Symposium on Search Based Software Engineering, pages 244–258. Springer, Springer, 2012.

[141] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering 2002, Charleston, South Carolina, USA, November 18-22, 2002*, pages 1–10. ACM, 2002.

[142] Andreas Zeller. *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2009.

[143] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. Boosting spectrum-based fault localization using pagerank. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 261–272, New York, NY, USA, 2017. ACM.

[144] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. A study of effectiveness of dynamic slicing in locating real faults. *Empirical Software Engineering*, 12(2):143–160, apr 2007.

[145] Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. Experimental evaluation of using dynamic slices for fault location. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 33–42. ACM, 2005.

[146] Guyu Zhao, Hongdou He, and Yifang Huang. Fault centrality: boosting spectrum-based fault localization via local influence calculation. *Applied Intelligence*, pages 1–23, 2021.

[147] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael Ernst, and Lu Zhang. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, PP, 03 2018.