JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2014; **00**:1–24
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr

Framing Program Comprehension as Fault Localization

Alexandre Perez[12*], Rui Abreu[12*]

[1]*Department of Informatics Engineering, Faculty of Engineering of University of Porto, Portugal*
[2]*Palo Alto Research Center, Palo Alto, CA, USA*

SUMMARY

Program comprehension is a time-consuming task performed during the process of reusing, reengineering, and enhancing existing systems. Currently, there are tools to assist in program comprehension by means of dynamic analysis. However, most cannot identify the topology and the interactions of a certain functionality in need of change, especially when used in large, real-world software applications. We propose an approach, coined Spectrum-based Feature Comprehension (SFC), that borrows techniques used for automatic software-fault-localization, which were proven to be effective even when debugging large applications in resource-constrained environments. SFC analyses the program by exploiting run-time information from test case executions to identify the components that are important for a given feature (and whether a component is used to implement just one feature or more), helping software engineers to understand how a program is structured and each the functionality's dependencies are. We present a toolset, coined PANGOLIN, that implements SFC and displays its report to the user using an intuitive visualization. A user study with the open-source application Rhino is presented, demonstrating the efficiency of PANGOLIN in locating the components that should be inspected when changing a certain functionality. Participants using SFC spent a median of 50 minutes locating the feature with greater accuracy, whereas participants using coverage tools took 60 minutes. Finally, we also detail the Participatory Feature Detection (PFD) approach, an extension of SFC, where user interactions with the system are captured, removing the hinderance of requiring pre-existing automated tests.

Received ...

KEY WORDS: Software Evolution and Maintenance, Fault Diagnosis, Program Comprehension

1. INTRODUCTION

Software maintenance is a crucial part of software engineering. The need to add or change features in existing software applications is becoming more and more prevalent. Furthermore, the ever increasing complexity of software systems and applications renders software maintenance even more challenging.

One of the most daunting tasks of software maintenance is to understand the application at hand [1]. In fact, recent studies point out that developers spend 60% to 80% of their time in comprehension tasks [2]. During this program understanding task, software engineers try to find a way to make both the source-code and the overall program functionality more intelligible. One of these ways is to create a "mental map" of the system structure, its functionality, and the relationships and dependencies between software components [3, 4].

*Correspondence to: `alexandre.perez@fe.up.pt, rui@computer.org`

Contract/grant sponsor: FCT – Portuguese Foundation for Science and Technology; contract/grant number: FCOMP-01-0124-FEDER-020484

To fully understand how a software application behaves, software engineers need to thoroughly study the source-code, its documentation and any other available artifacts. Only then the engineer gains sufficient understanding of the application, enabling him/her to seek, gather, and make use of available information to efficiently conduct maintenance or evolution tasks. This *program comprehension* (also known as *program understanding/software comprehension*) phase is thus resource and time consuming. In fact, studies show that up to 50% of the time needed to complete maintenance tasks is spent on understanding the software application and gaining sufficient knowledge to change the desired functionality [1]. Currently, there are several approaches that focus on dynamic analysis to provide visualizations of the software system, identifying their components and their relationships, *e.g.*, [5, 6, 7]. However, these approaches may not clearly show what code regions the developer needs to inspect in order to change a certain functionality. Another problem regarding dynamic analysis is the fact that program traces of sizable programs encompass large amounts of data [8]. This leads to two kinds of issues, namely (1) scalability issues when gathering, storing and analyzing traces; and (2) challenges concerning the visualization of such traces and their analysis results to the user.

To address some of the issues of past approaches, we propose an approach, coined Spectrum-based Feature Comprehension (SFC), that exploits techniques used in the software fault-localization domain. Fault-localization techniques exploit coverage information of test cases to calculate the likelihood of each component being faulty, and were shown to be efficient, even for large, resource-constrained environments [9]. Our approach leverages these concepts to provide an efficient dependency analysis and visualization for software comprehension that does not have the same scalability hindrances as other related work.

To support the effectiveness of our approach, we apply information-foraging-based theory. Information-foraging is a theory to explain and predict how people use environmental information to achieve their goals [10]. It builds its hypothesis upon optimal foraging theory, drawing from noticed similarities between users' information searching patterns and animal food-foraging strategies. Information-foraging theory assumes that human beings have the capability to efficiently filter irrelevant information out, so that they achieve their goal at minimum cost (e.g., time it takes to change a feature).

To assess the effectiveness of our solution, we have conducted a user study with the open-source project Rhino. It demonstrates the accuracy and effectiveness of the SFC approach and its visualizations in aiding users to pinpoint the components that need to be inspected when evolving/changing a certain feature in the application. It also shows that users spend less time locating features with SFC when compared to using standard coverage tools. In the case of the user study, participants using SFC spent a median of 50 minutes locating the feature, whereas participants using coverage tools took 60 minutes.

We also address a potential drawback of the SFC approach by extending it to allow Participatory Feature Detection (PFD). To pinpoint a feature with PFD, programmers are asked to interact with the application and to record whether the feature was involved or not in each execution. This way, SFC becomes applicable to any interactive application, without requiring a test suite to obtain the execution traces.

The paper makes the following contributions:

- We describe Spectrum-based Feature Comprehension (SFC), an approach that, similar to fault-localization techniques, exploits run-time information from system executions to identify dependencies between components, helping software engineers in understanding how a program is structured.
- We detail an information-foraging theory to support the effectiveness of our approach.
- We provide a toolset, PANGOLIN, providing a visualization of associated and dissociated components of an application functionality.
- A user study with a large, real-world, software project, demonstrating the effectiveness of our approach in locating the components that should be inspected when evolving/changing a certain feature.

- We extend the approach to allow participatory feature detection: users can manually collect and label executions on any interactive project, removing the hinderance of requiring pre-existing automated tests.
- We propose an accuracy metric to evaluate the effectiveness of the SFC approach.

This paper is an extension of previous work [11], where SFC was first introduced. This work builds on top of it by introducing the user participation concept to gather data and feed the diagnostic algorithm, and by proposing an accuracy metric to evaluate the approach.

The remainder of this paper is organized as follows: In Section 2 we introduce the concepts relevant to this paper, namely program spectra and fault-localization. Section 3 will present our spectrum-based feature comprehension approach. Section 4 introduces the PANGOLIN toolset that implements our approach as an Eclipse plugin. In Section 5 we describe the user study setup and present its results. In Section 6 we extend the approach to enable the participation of users in the collection of runtime traces. We provide an overview of the related work and how it compares to PANGOLIN in Section 7. Finally, in Section 8, we conclude and discuss future work.

2. PRELIMINARIES

In this section, spectrum-based fault-localization is detailed. After that, an approach to visualize diagnostic reports is presented.

2.1. Spectrum-based Fault-Localization

Spectrum-based Fault-Localization (SFL) is a debugging technique that calculates the likelihood of a software component being faulty [12]. It exploits information from passed and failed system transaction. A passed transaction is a program execution that is completed correctly (*i.e.*, the program behaves as expected), and a failed transaction is an execution where an error was detected [9]. The criteria for determining if a transaction has passed or failed can be from a variety of different sources, such as test-case results and program assertions, among others. The execution information gathered for each transaction is their program spectra.

A program spectrum is a characterization of a program's execution on an input collection [13]. This collection of data consists of counters of flags for each software component, and is gathered at runtime. Software components can be of several detail granularities, such as classes, methods, or statements. A program spectrum provides a view on the dynamic behavior of the system under test [14]. Many types of program spectra exist. This paper focuses on registering if a component is involved a certain execution, so binary flags can be used for each component, yielding a small memory footprint and minimal runtime overhead. This particular form of program spectra is also called hit spectra [14].

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

With this input, the next step consists in identifying what columns of the matrix A (*i.e.*, the hit spectrum for each component) resemble the error vector the most. This is done by quantifying the resemblance between these two vectors by means of *similarity coefficients* [15].

The similarity coefficient used throughout this work is the Ochiai coefficient [16]. This coefficient was initially used in the molecular biology domain [17], and is defined as follows:

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

where $n_{pq}(j)$ is the number of transactions in which the component j has been touched during execution ($p = 1$) or not touched during execution ($p = 0$), and where the transactions failed ($q = 1$)

	Runs						n_{pq}				
`largest() {`	1	2	3	4	5	6	n_{00}	n_{10}	n_{01}	n_{11}	s_O
`1: int a,b,c,large;`	●	●	●	●	●	●	0	5	0	1	0.41
`2: print("Enter 3 numbers");`	●	●	●	●	●	●	0	5	0	1	0.41
`3: read(a, b, c);`	●	●	●	●	●	●	0	5	0	1	0.41
`4: if (a>b) {`	●	●	●	●	●	●	0	5	0	1	0.41
`5: if (a>c)`	●			●	●	●	2	3	0	1	0.50
`6: large = a;`				●		●	3	2	1	0	0.0
`7: else large = a;} //BUG`	●				●		4	1	0	1	0.71
`8: else if (b>c)`		●	●				3	2	1	0	0.0
`9: large = b ;`		●					4	1	1	0	0.0
`10: else large = c;`			●				4	1	1	0	0.0
`11: print("Largest: ",large);`	●	●	●	●	●	●	0	5	0	1	0.41
`}` Error vector:	✓	✓	✓	✓	✗	✓					

(a) Test coverage information.

Ranking	s_O	Statement
1	0.71	`7: else large = a;}`
2	0.50	`5: if (a>c)`
4	0.41	`1: int a,b,c,large;`
5	0.41	`2: print("Enter 3 numbers");`
6	0.41	`3: read(a, b, c);`
7	0.41	`4: if (a>b) {`
8	0.41	`11: print("Largest: ",large);`

(b) Faulty statement ranking.

Figure 1. Example of SFL technique with Ochiai coefficient.

or passed ($q = 0$). For instance, $n_{10}(j)$ counts the number of times component j has been involved in passing executions, whereas $n_{01}(j)$ counts the number of failing executions that do not exercise component j. Formally, $n_{pq}(j)$ is defined as:

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

Several other similarity coefficients do exist [18, 19], but Ochiai was shown to be amongst the best in practice for fault localization purposes [20].

The calculated similarity coefficients rank the software components according to their likelihood of containing the fault. This is done under the assumption that a component with a high similarity to the error vector has a higher probability of being the cause of the observed failure. A list of the software components, sorted by their similarity coefficient, is then presented to the developer. This list is also called *diagnostic report*, and helps developers prioritize their inspection of software components to pinpoint the root cause of the observed failure.

Figure 1 is an example of the SFL technique. Under test is a function named `largest()` that reads three integer numbers and prints the largest value. This program contains a fault on line 7 – it should read `large = c;`. Figure 1a shows the function code, as well as the coverage trace and outcome of six test cases. With this information, similarity coefficients can be calculated for each line using the Ochiai coefficient (Equation (1)). The higher the coefficient, the more likely it is that a line contains a fault.

Tools that use SFL for their diagnosis, such as Zoltar [21], rank these similarity coefficients to form an ordered list of the probable faulty statements (also referred to as diagnostic report), and present it to the user. Users can then start inspecting the statements located in the higher positions of the diagnostic report, until they reach the faulty statement. Figure 1b depicts the diagnostic report generated by these tools for the `largest()` example.

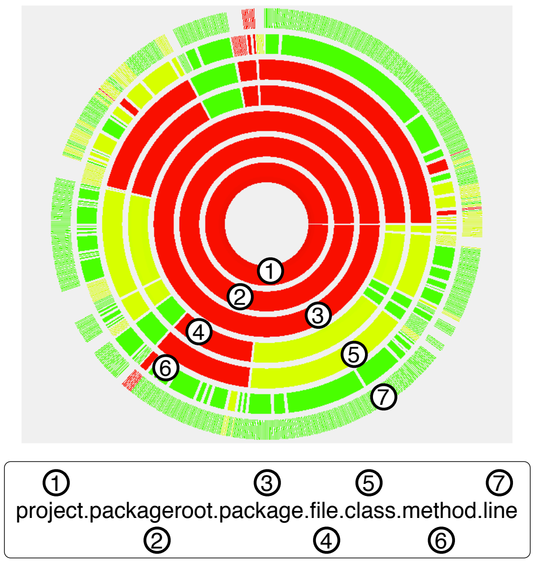


Figure 2. Sunburst Visualization Example.

2.2. *Diagnostic Report Visualization*

To improve the intuitiveness of the diagnostic report generated by SFL, interactive visualization techniques have been proposed and are available within the GZoltar toolset [22, 23]. GZoltar is a fault-localization plugin for the Eclipse integrated development environment. Besides fault-localization, GZoltar also provides mechanisms for test-suite minimization and prioritization [24], but those are beyond the scope of this paper.

GZoltar provides developers with several different visualizations, namely sunburst, vertical partition, and bubble hierarchy. According to user feedback, the sunburst visualization was deemed the most intuitive [23]. In this visualization, each ring denotes a hierarchical level of the source-code organization, as depicted in Figure 2. From the inner to the outer circle, this visualization presents projects, packages, files, classes, methods, and lines of code. Navigation in this visualization is done by clicking on a component, which will display all the inner components of the selected one. As an example, if a user clicks in a class, all of the class methods will be displayed. Users may also zoom in/out and pan to analyze in detail a specific part of the system. Any inner component can also be set as the new root of the visualization, and only that component's sub-tree is displayed. This operation is called a *root change*. The color of each component in the visualization represents its likelihood of being faulty. Ranging from bright green if the similarity is close to zero; to yellow if the similarity is close to 0.5 and finally to red if the component's similarity to the error vector is close to 1.

The effectiveness of these fault-localization techniques and the hierarchical visualization was also demonstrated in a user study [23], where 40 participants, without any knowledge of the system under test, were asked to locate a fault in under 30 minutes. Everyone using the hierarchical visualization with diagnostic information was able to find the bug, whereas only 35% of the participants of the control group succeeded.

3. SPECTRUM-BASED FEATURE COMPREHENSION

This section details our approach coined Spectrum-based Feature Comprehension (SFC) that uses the techniques and code visualizations mentioned in the previous section in the context of program comprehension.

3.1. Concepts & Definitions

To leverage these fault-localization techniques, its concepts and definitions need to be mapped into the program understanding domain. The first one is the notion of a feature.

Definition 1. A **feature** is the source-code portion that implements a certain functionality. It can encompass one or more components.

This definition is closely related to the concept of feature in the domain of software product-line research: *"a feature is a unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option"* [25].

When trying to evolve/modify a certain feature f, we are interested in the relationships and interactions between f and other components in the source-code. As such, one should not use the error vector e to compute the similarity coefficient (cf. Section 2.1). Instead, an *evolution vector* ev_f should be considered.

Definition 2. The **evolution vector** ev_f is an N-length binary vector. In this vector, a given position i is true (*i.e.*, set as 1), if the i^{th} transaction executes feature f.

While the error vector captures whether transactions have passed or failed, the evolution vector ev_f captures whether feature f was involved in each transaction. The resemblance between the evolution vector and system components is captured by the association measure:

Definition 3. **Association measure** of a component j indicates the degree of similarity between j and the evolution vector.

In this paper we consider Ochiai as the similarity coefficient used to calculate the association measure.

When evolving a feature f, it is important to inspect its associated components because they may either call f or be called by f, and thus may need to be modified in accordance with the changes made to f.

Definition 4. A component j is **associated** with f if its association measure is close to 1. This means that when f is executed, j is likely to involved.

In contrast to associated components, if a component is dissociated to f, it does not need to be inspected when f is modified.

Definition 5. A component j is **dissociated** to f if its association measure is close to 0. This means that when f is executed, j is not likely to involved.

Components with association measures of neither 0 nor 1 should be inspected, but only modified with great care. This is because these components are shared amongst features.

3.2. Approach

The SFC approach is depicted in Algorithm 1. The inputs for the algorithm are:

- $\mathcal{P}$ – the program under evaluation.
- $\mathcal{U}$ – set of system runs.
- $\mathcal{U}_f$ – set of runs exercising feature f.

The output is the report $\mathcal{R}$, which is a list of components, each containing its *association measure* to the feature under consideration.

First, all system runs $\mathcal{U}$ of the program $\mathcal{P}$ are executed (Line 3), yielding the program spectra matrix A. This matrix contains the execution traces for every system run in $\mathcal{U}$. Next, the evolution vector is created, by constructing a binary vector of dimension $|\mathcal{U}|$ and setting every element according to its membership in $|\mathcal{U}_f|$.

Algorithm 1 Spectrum-based Feature Comprehension.

```
Input:
    Program $\mathcal{P}$
    Set of runs $\mathcal{U}$
    Set of runs that exercise the feature $\mathcal{U}_f$
Output:
    Report $\mathcal{R}$

1: $\mathcal{R} \leftarrow \varnothing$
2: $A \leftarrow \varnothing$
3: $\forall_{i \in \{1 \ldots |\mathcal{U}|\}} : A \leftarrow A \cup \textsc{GatherCoverage}(P, \mathcal{U}_i)$
4: $ev \leftarrow \textsc{EvolutionVector}(\mathcal{U}, \mathcal{U}_f)$
5: $\forall_{j \in \{1 \ldots |\mathcal{P}|\}, i \in \{1 \ldots |\mathcal{U}|\}} : n_{01}(j) \leftarrow |\{i \mid A_{ij} = 0 \wedge ev_i = 1\}|$
6: $\forall_{j \in \{1 \ldots |\mathcal{P}|\}, i \in \{1 \ldots |\mathcal{U}|\}} : n_{10}(j) \leftarrow |\{i \mid A_{ij} = 1 \wedge ev_i = 0\}|$
7: $\forall_{j \in \{1 \ldots |\mathcal{P}|\}, i \in \{1 \ldots |\mathcal{U}|\}} : n_{11}(j) \leftarrow |\{i \mid A_{ij} = 1 \wedge ev_i = 1\}|$
8: $\forall_{j \in \{1 \ldots M\}} : \mathcal{R}[j] \leftarrow s_O(n_{01}(j), n_{10}(j), n_{11}(j))$
9: return $\mathcal{R}$
```

Following, in Lines 5 to 7 the occurrence variable n_{pq} (as described in Section 2.1) is calculated for each component of program $\mathcal{P}$. Note that n_{00} is not calculated because Ochiai (see Equation (1)) does not use that term. If other similarity coefficient is used instead of Ochiai and if it requires n_{00} to perform the computation, then the variable would need to be gathered.

Finally, for each component, the association measure is calculated with using the Ochiai coefficient, and stored in the report $\mathcal{R}$. Unlike what happens in SFL, the report $\mathcal{R}$ does not have to be sorted. This is because the report will not be inspected as a ranking by the user.

3.3. Complexity Analysis

As for the space complexity, the generated program spectra matrix A has a complexity of $O(|\mathcal{P}| \cdot |\mathcal{U}|)$. The evolution vector and the report $\mathcal{R}$ complexities are $O(|\mathcal{U}|)$ and $O(|\mathcal{P}|)$, respectively. The n_{01}, n_{10} and n_{11} counters each have a complexity of $O(|\mathcal{P}|)$. Therefore, the worst case space complexity is $O(|\mathcal{P}| \cdot |\mathcal{U}| + 4 \cdot |\mathcal{P}| + |\mathcal{U}|) = O(|\mathcal{P}| \cdot |\mathcal{U}|)$. Note that a simple optimization to make is not to store the A matrix but rather update the n counters after each test is executed. This would reduce space complexity to $O(4 \cdot |\mathcal{P}| + |\mathcal{U}|) = O(|\mathcal{P}| + |\mathcal{U}|)$.

The time complexity is as follows. Assuming that all system runs in set $\mathcal{U}$ are executed and take the same amount of time to execute, the complexity of this test execution step is $O(|\mathcal{U}|)$. As for the evolution vector computation, its worst case time complexity is $O(|\mathcal{P}| \cdot |\mathcal{U}|)$. The computation of the n_{pq} occurrence function also has a complexity of $O(|\mathcal{P}| \cdot |\mathcal{U}|)$.

Finally, the Ochiai coefficient calculation to populate the report $\mathcal{R}$ is $O(|\mathcal{P}|)$. The worst case time complexity is $O(|\mathcal{P}| \cdot |\mathcal{U}| + |\mathcal{P}| + |\mathcal{U}|) = O(|\mathcal{P}| \cdot |\mathcal{U}|)$.

3.4. Report Visualization

To visualize the report produced by our approach, we use the Sunburst visualization, just like the GZOLTAR framework. We apply information-foraging theory to explain the usefulness of the sunburst visualization for improving source-code understanding. Information-foraging theory aims to both *"explain and predict how people will best shape themselves for their information environments and how information environments can be shaped for people"*, as defined by Peter Pirolli [10].

This theory is itself based on optimal foraging theory, which tries to explain the behavior of *predators* and *preys*. *Predators* try to find *preys* by following their *scent*, and *preys* are more likely to be in places (or *patches*) where the *scent* is more intense. In the information-foraging context, the *predators* are the people in need of information and the *preys* are the information itself. The *scent* is the interpretation of the environment by the *predators*. Sjoberg *et al.* [26] suggests that a theory is best used to explain (at least one) of the following questions: what is, why, forecast future events, and guiding how to do something. Information-foraging theory can be used to answer all these questions.

In order to apply this information-foraging theory in the context of automatically generated diagnostic reports, a mapping between the theory constructs and this context must be established. Closely following the theory proposed by Lawrence *el al.* [27] for the context of debugging, in this paper we map the information-foraging theory constructs as follows:

- *Predator* is the person performing the maintenance task;
- *Prey* is what the *predator* seeks to know to pinpoint the code regions that need to be changed;
- *Information patches* are localities in the source-code that may contain the *prey*;
- *Proximal cues* are the runtime behaviors that suggest scent related to the *prey*;
- *Information scent* is the *predator* interpretation of the report;
- *Topology* is the collection of paths through the source-code and report through which the programmer can navigate. In essence, it includes IDE features that help navigating the code.

The *topology* is a graph representing elements of the source-code (e.g., classes, methods) and the diagnostic report with navigable links between elements. The navigable links between the elements allow the programmer to traverse the connection at the cost of just one click. Information-foraging theory assumes that the developer's choices are an attempt to maximize the information gain per navigation interaction's cost. As in [28], this can be characterized as

$$choice = max\left(\frac{G}{C}\right)$$

where G is the information gain and C is the cost of the interaction (including both the visualization and the IDE features). Since the G and C values are not known to the developer *a priori*, his decisions will be based on the expected gain and cost.

When looking for the code regions that need to be changed, the developer relies on the cues to decide which *place* to inspect next. Those cues are used to estimate the trade-off between the navigation cost and the value to be gained. In an attempt to take the best decision, the developer will favor links whose cues will lead him to the location of code regions in need of change. Better cues are therefore more likely to lead to better information scent, hence reducing the cost incurred while maximizing the value gained.

By analyzing the Sunburst visualization described in Section 2.2 in regard to information-foraging, we may argue that its visualization of the system's topology and its interaction features can indeed reduce the cost of navigating through the various system components (be it packages, classes, methods, even statements) and thus the cost of navigation C is reduced. By color coding of each component, which is obtained from the fault-localization ranking, we are providing proximal cues, guiding the developer towards likely associated regions of the source-code. At the same time, we are notifying the developer about regions that should not be explored (where, e.g., relevant executions were not active). Hence, a better information scent is conveyed to the developer, increasing the information gain.

4. PANGOLIN TOOLSET

In this Section, we introduce the PANGOLIN toolset[†], an Eclipse plugin that implements the SFC approach and displays its results with the aid of a sunburst visualization. It extends the GZoltar toolset for testing and debugging, also providing all of its features.

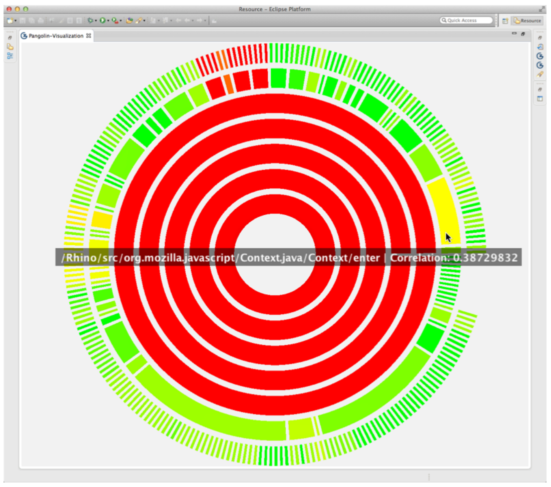

Resource – Eclipse Platform
Pangolin-Visualization
/Rhino/src/org.mozilla.javascript/Context.java/Context/enter | Correlation: 0.38729832

Figure 3. PANGOLIN's sunburst visualization.

The PANGOLIN plugin performs a very lightweight dynamic analysis by instrumenting the project, so that the activity matrix is gathered during runtime. The plugin uses the project's JUnit test cases as the set of system runs. In order to perform the analysis, users must also identify in a specific view which system runs exercise the feature under consideration. After that, PANGOLIN plugin computes a feature-association-measure for every component in the project, and displays that information in a sunburst visualization, as shown in Figure 3. This sunburst visualization depicts the current project's topology in a hierarchic fashion, starting from the root component representing the whole project in the inner circle, up to individual lines of code in the outer circle. Each component is color coded with the corresponding association measure, ranging from bright green if the association measure is close to zero; to yellow if the association measure is close to 0.5 and to red if it is close to 1. When a user hovers the mouse on a component, a label identifying that component and its association measure is shown, as depicted in Figure 3. If he/she clicks that component, Eclipse's code editor will open and the cursor is positioned on the start of the chosen component.

We also enhance the sunburst visualization to show a summary of what each code class is responsible for. We rank every term used in each class file and apply term frequency-inverse document frequency (*tf-idf*) weighing, commonly used in the Information-Retrieval (IR) domain [29]. The *tf-idf* value increases proportionally to the number of times a term appears in the document (in this case, a class file). However, it is also offset by the frequency of that term in the overall collection of documents, so that common terms have less weight. Top terms in the ranking are shown when hovering a class component in the visualization, with the intent of providing more cues to the developer, improving his understanding of the selected component.

5. USER STUDY

In this section, we evaluate the effectiveness of SFC when applied to a real-world application – the Rhino project. This user study aims to answer the following research questions:

[†]PANGOLIN is available online at `www.gzoltar.com/pangolin`.

RQ1: *Can programmers using Pangolin pinpoint a feature more accurately than by inspecting standard test coverage traces?*

RQ2: *Is the time taken to pinpoint a feature with Pangolin at least comparable to inspecting standard test coverage traces?*

In **RQ1**, we are concerned about assessing the effectiveness of the technique and our tool. **RQ2** tries to ensure that our approach will not negatively impact existing program comprehension processes.

First, we describe the subject of our evaluation and the setup for our user study. Afterward, we present the results of the user study and potential threats to validity.

5.1. The Rhino Project

The software application under consideration for this case study is the open-source project Rhino[‡]. Rhino is a Javascript engine written entirely in Java and is managed by the Mozilla Foundation. It is typically embedded into Java applications to provide scripting to end users and also allows JavaScript programs to leverage Java platform APIs. Rhino automatically handles the conversion of JavaScript primitives to Java primitives, and vice versa (*i.e.*: JavaScript scripts can set and query Java properties and invoke Java methods). Rhino is comprised by 28 packages, 433 classes and 75170 source lines of code. Furthermore, this project contains 441 unit tests, written for the JUnit framework, which cover 56% of the project's statements and 45% of branches (9,323 out of a total 20,802 branches).

5.2. User Study Setup

The user study was performed by 108 students enrolled in the Software Engineering course of the Master in Informatics and Computing Engineering program from the Faculty of Engineering of University of Porto.

The experiment was performed in the context of a lab session for the Software Engineering course. A pre-requisite for enrollment in that course is that students must have completed the following courses: 'Programming Fundamentals', 'Algorithms and Data Structures' and 'Object-Oriented Programming Lab', which means that all participants had at least three years of experience with the Java programming language and were familiar with both the Eclipse IDE and the JUnit testing framework. None had, however, used Rhino before. Participants were grouped into pairs to perform the requested task, which was also a course requirement. Each pair had access to one computer to perform the task. They were not compensated for performing the experiment. However, with the experiment being performed in a lab session, its attendance was mandatory.

The requested task was the following. Participants were requested to identify source-code regions that exclusively implement a certain feature (labeled as task (T1)), and also regions where that feature is being used (*i.e.*, code regions shared among different features, labeled as task (T2)). It is important to distinguish between these two kinds of regions when changing a feature. While developers can change regions labeled as (T1) without many concerns, regions labeled as (T2) require a more detailed inspection before changing the code, as the changes can break other functionalities. The feature under consideration for this user study was Rhino's continuation context creation. This feature is responsible for creating a snapshot of a context, which contains the execution information needed to run Javascript code. One example of the information stored in a context is the call stack representation. These snapshots, which Rhino calls continuation objects, allow for users to pause execution and/or to return to a previous state in the execution.[§] A tutorial explaining the feature in detail was given to all participants,[¶] which were given 20 minutes before the

[‡]Available at `https://developer.mozilla.org/en-US/docs/Rhino`

[§]More information about Rhino's continuation objects is available at `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino/New_in_Rhino_1.7R2`

[¶]All tutorials produced for this user study are available online at `http://gzoltar.com/pangolin/replication-package/`

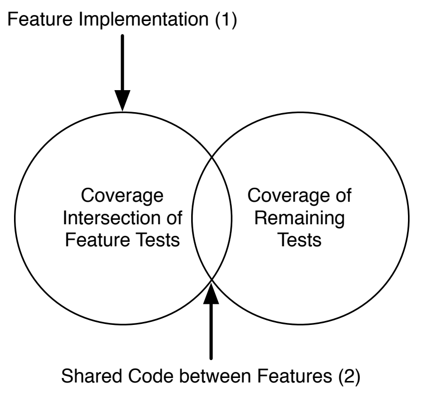

Feature Implementation (1)
Coverage Intersection of Feature Tests
Coverage of Remaining Tests
Shared Code between Features (2)

Figure 4. Feature analysis with a coverage tool.

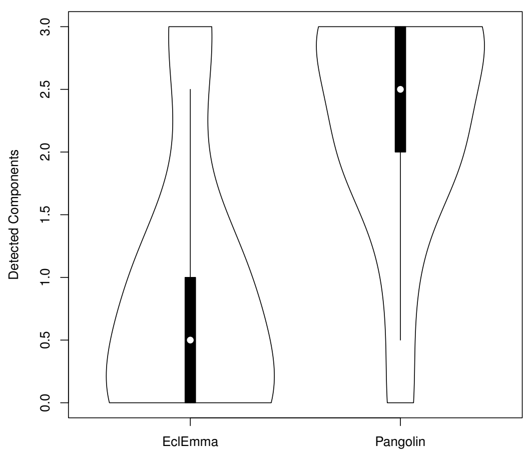
start of the experiment to study all materials provided. A time limit of 100 minutes was established to complete the task.

The set of tests relevant to the creation of continuation contexts was gathered by the authors by manually inspecting every test before when setting up the experiment, and was given to all participants. The methodology for choosing the tests was as follows. First, all tests that did not execute Javascript scripts in interpreted mode were discarded. Rhino's continuations API is only supported in interpreted mode, which is selected by invoking `setOptimizationLevel(-1)` in the `Context` instance. Second, a further restriction taken into consideration is that continuation objects only capture contexts when scripts are called via the methods `executeScriptWithContinuations` and `callFunctionWithContinuations` from the `Context` instance. The search yielded two test classes, `ContinuationsApiTest` and `Bug482203Test`. The former exercises Rhino's continuations API, by executing scripts, pausing them, and capturing/restoring state; the latter was created to expose a bug in a previous version of Rhino that threw `NullPointerException` exceptions when capturing a continuation state. Manual inspection revealed that both test classes exercised the feature under consideration (namely, by covering the `captureContinuation` method in the `Context` class).

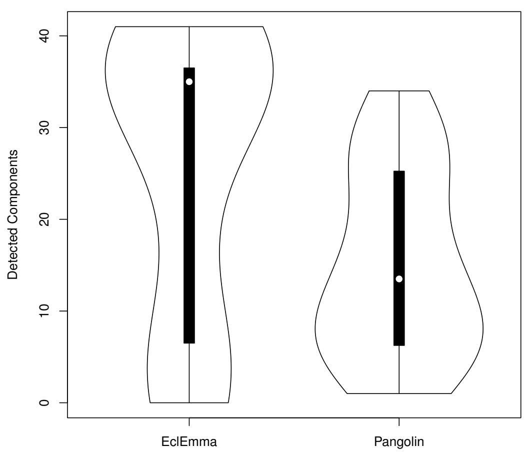
Participants were divided into two groups. One group comprising 26 pairs of participants was asked to use the PANGOLIN plugin to complete the task. As all participants were unfamiliar with PANGOLIN, a short tutorial explaining how to work with the tool (and how to interpret the results) was shown. For this group of participants to successfully complete the task, they need to use the tool to indicate the set of tests exercising the feature and run PANGOLIN's analysis. After the analysis is complete, the sunburst visualization appears in the corresponding Eclipse view. To identify the code regions that implement the feature (T1), participants should look for components whose association measure is 1 (color coded as red). Code regions shared among several features (T2) are components whose association measure is above 0 and below 1, and therefore color coded as different shades of yellow.

The other group of participants comprised by 28 pairs was the control group. Participants were asked to use the features from a standard version of the Eclipse IDE and its code-coverage plugin EclEmma[‖], that shows, for a set of tests, what statements were executed. A short tutorial on how to work with EclEmma plugin was given beforehand. For the task to be successfully completed, participants need to gather the code-coverage information of all tests that exercise the feature, and compute their intersection. The intersection between these tests denotes the code regions that were executed on every test. A set difference between this intersection and the coverage of remaining

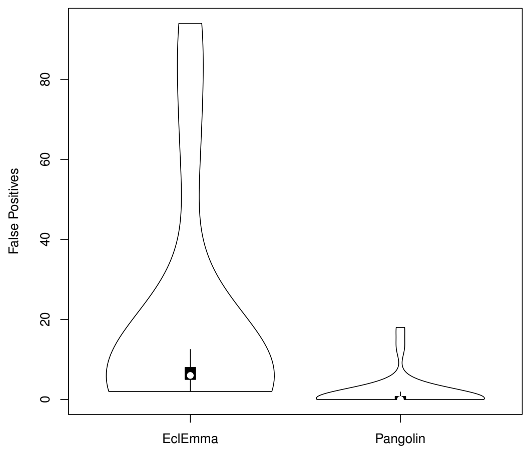
[‖]Available at `http://www.eclemma.org/`



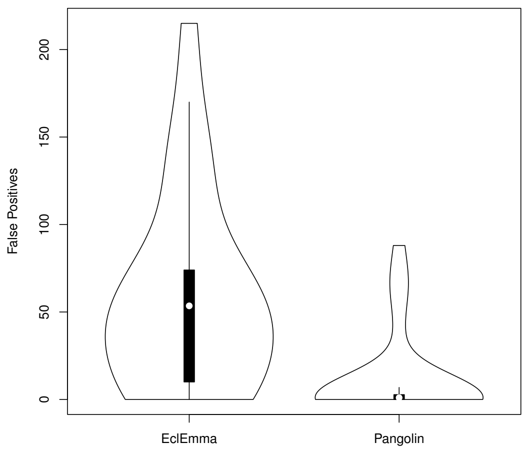
(a) Detected implementation components (T1).



(b) Detected shared components (T2).



(c) False positives labeled as implementation components (T1).



(d) False positives labeled as shared components (T2).

Figure 5. Violing plots depicting both groups' accuracy when labeling components.

tests in the test suite allows us to identify the regions that (T1) exclusively implement the feature and that (T2) are shared among many features, as is depicted in Figure 4.

For the particular feature considered in this user study, Rhino's continuation context creation (described in the previous subsection), users have to identify code regions in 3 classes that exclusively implement the feature (T1), and another 41 classes where that feature is used among many others (T2).

5.3. *Results & Discussion*

From the group that used the PANGOLIN plugin, participants were able to correctly identify a median of 2.5 classes from category (T1) and 13.5 from category (T2). In the group that used the code-coverage plugin EclEmma, participants identified a median of 0.5 classes from category (T1) and 35 from category (T2). Figures 5a and 5b show violin plots[**] depicting the amount of correct components detected by participants. We can see that, for identifying components in category (T1), participants working with PANGOLIN were able to achieve better results. In fact, over two thirds of the pairs of participants working with that plugin were able to find at least two correct components

[**]A violin plot is the combination of a box plot and a kernel density plot.

Table I. Mann-Whitney U tests.

Null Hypothesis	Figure	U	p-value	EclEmma Mean	PANGOLIN Mean	Effect size (d)
Detected components (T1) for both tools come from the same population	5a	127.5	1.12×10^{-5}	0.82	2.27	Large (1.46)
Detected components (T2) for both tools come from the same population	5b	223.5	7.60×10^{-3}	25.07	15.03	Medium (0.75)
False positives (T1) for both tools come from the same population	5c	48.0	1.03×10^{-8}	19.14	1.65	Large (0.83)
False positives (T2) for both tools come from the same population	5d	130.0	2.23×10^{-5}	57.93	10.88	Large (1.07)
Elapsed time for both tools come from the same population	6	234.0	1.20×10^{-2}	64.10	51.23	Medium (0.66)

(out of three in total), as opposed to participants using EclEmma, where only 5 pairs identified at least two correct components.

As for the identification of components from category (T2), participants using EclEmma showed an increased overall accuracy when compared to PANGOLIN. However, and along with the correct code regions, there were several false positives identified by participants. The group using EclEmma registered a median of 6 while identifying regions that exclusively implement the feature, while the group using PANGOLIN registered a median of 0 false positives. The second category, code regions with shared features, yielded a median of 53.5 false positives when using EclEmma versus only 1 false positive when PANGOLIN is used. The amount of false positives each pair of participants identified can be seen in the violin plots from Figures 5c and 5d. Figure 5c concerns the false positives while identifying code regions for category (T1), whereas Figure 5d shows the false positives while identifying category (T2). In both categories, we see a substantial increase in the amount of false positives when the code-coverage EclEmma plugin is used to perform the requested task. This happens because the majority of participants using EclEmma, after gathering code-coverages for the indicted test cases, did not perform an intersection of the traces, as depicted in Figure 4. As a result, a considerable number of components were labeled incorrectly.

We also performed statistical tests to assess whether the gathered metrics yielded statistically significant results. The statistical test used is the Mann-Whitney U test. The reason we use Mann-Whitney instead of, *e.g.*, Student's t-test is because it does not assume that the data is normally distributed. In fact, a Shapiro-Wilk test on the data confirms that the distributions are not normal.

The results shown on Table I include the description of the null hypothesis, the task category from which the data originated, the test's U statistic and p-value. The p-values for all tests performed suggest that, according to the data, the null hypothesis must be rejected. The first four tests indicate that the two groups of participants can be considered statistically significant with 99% confidence, the last one has a 95% significance. Also shown are the means gathered for both groups (using EclEmma and using Pangolin) for each test. In the first two tests, which investigate the detected components distribution, the bigger the values, the more accurate is the tool. When evaluating false positives and elapsed time, smaller values are better. Lastly, the Cohen's difference between two means (d) for measuring the effect size is shown, as well as Cohen's qualitative effect size description.

Revisiting the first research question:

RQ1: *Can programmers using* PANGOLIN *pinpoint a feature more accurately than by inspecting standard test coverage traces?*

Results show that the information about the program provided by the SFC analysis and the sunburst visualization is more accurate than requiring users to inspect and compare several traces with a code-coverage tool. Although in category (T2), users working with EclEmma were able to detect more components, this approach yielded a large number of false positives, which will most likely increase the comprehension effort, as users will need to inspect those components and deem then dissociated from the feature. From an information-foraging standpoint, we argue that due to the intuitiveness of the visualization and accuracy of SFC, PANGOLIN provides better cues than the EclEmma and, ultimately, increases the perceived information gained about the system being inspected.

The last metric gathered was the time each pair of participants took to complete the task. Although a time limit of 100 minutes was established, only two pairs required that amount to submit their results. Figure 6 depicts the elapsed time for each pair of participants. Overall, the group using PANGOLIN completed the task in less time compared to the group using EclEmma. Participants using PANGOLIN took a median of 50 minutes to complete, whereas participants working with the EclEmma plugin took 60 minutes. The main reason for participants using EclEmma taking longer to complete the task is the fact that, after gathering the coverage information, they needed to perform the coverage analysis as shown in Figure 4. Participants using PANGOLIN only needed to gather the information shown to them via the sunburst visualization. No extra analysis was required.

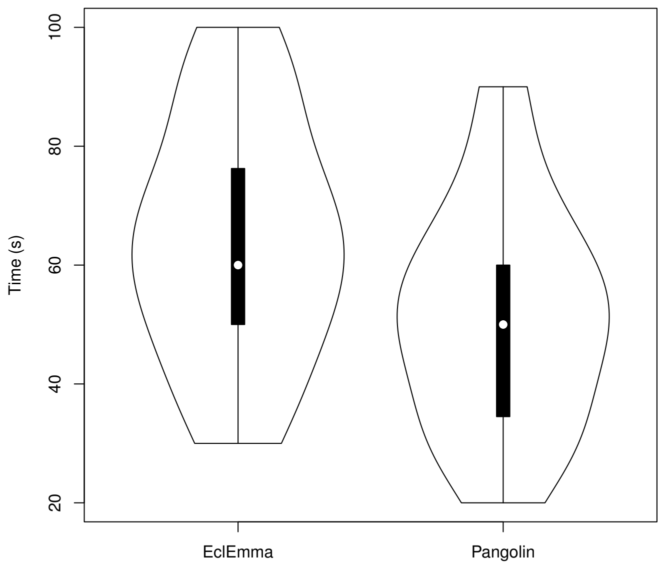


Figure 6. Time required by each pair of participants to complete the task, sorted by ascending order.

Revisiting the second research question:

RQ2: *Is the time taken to pinpoint a feature with* PANGOLIN *at least comparable to inspecting standard test coverage traces?*

We conclude that using PANGOLIN does not negatively impact the time needed for comprehension processes, with the added advantage of being more accurate (as seen by answering **RQ1**).

5.4. Threats to Validity

Construct Validity Regarding construct validity, a threat is the fact that we measure components at a class level. Although the majority of the PANGOLIN groups have provided methods and line numbers in their reports, EclEmma groups had difficulty reporting components of finer granularity. Faced with this fact, we had to settle with class granularity so that both approaches could be compared.

Internal Validity A potential threat to the internal validity of this study is related to the selection of Mozilla Rhino as the application under analysis. When choosing the application for our study, our aim was to opt for application that (1) is indicative of a general large-sized application being worked on by several people, and that (2) also provides significant understanding challenge to participants. To reduce selection bias, we decided to choose an application used in the evaluation of related work [30, 31, 32]. Other threat is the fact that the participants worked in pairs and chose who they wanted to pair up with within each group, which can introduce a potential bias to the experiment. One final threat to internal validity is related to some potential fault in the PANGOLIN tool, or any underlying implementation, such as the instrumentation for gathering program spectra. To minimize this risk, thorough testing and individual result checking were performed before the experimental phase.

External Validity The main threat to the external validity of these results is the fact that participants were given the set of tests that exercise the feature under consideration. Information about what features each test is exercising may not be available or difficult to obtain. In fact, software projects may not even have tests for every feature. Another threat to external validity is the fact that only one feature of one open-source application was used in the study. It is plausible to assume that a different set of subjects, having inherently different characteristics, may yield different results. Also, all participants in the user study were software engineering students, and the study was performed in an academic setting, so it may not correctly reproduce the problems that the industry deals with. However, we argue that our setting closely resembles an important challenge faced in the software industry regarding program comprehension: introducing junior programmers into well established projects.

6. PARTICIPATORY FEATURE DETECTION

Feedback from the user study participants was that PANGOLIN had helped them locating the fault. However, at the end of the experiment, when participants were running the analysis against other projects, they were rarely able to successfully perform the feature localization. This happens because the majority of the projects they experimented with did not have any unit tests, so the hit spectra matrix, required for the SFC analysis, was empty. In fact, as was pointed out in Section 5.4, our approach provides an automated way of locating features in the code, given that (1) there is a test suite and (2) the mapping between features and tests is known. In real software development scenarios, although it is good practice to test the system and to maintain a test-feature mapping, these are rarely available, which limits the applicability of the approach. To address this concern, we introduce and evaluate the concept of Participatory Feature Detection (PFD).

6.1. The PFD Concept

The PANGOLIN tool has limited applicability when there is no pre-existing set of transactions (*i.e.*, no test suite available) or when there is no information on which runs have exercised the feature we are looking for. To address these issues, we extended both our SFC approach and our tool to account for Participatory Feature Detection (PFD). PFD allows users to capture manual interactions with the system, enabling them to label each interaction as associated or dissociated with the feature.

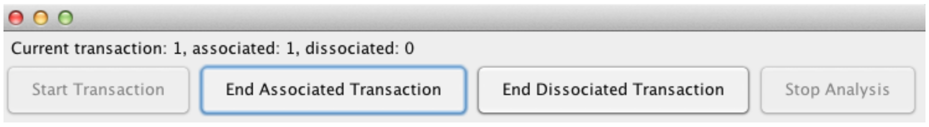

Current transaction: 1, associated: 1, dissociated: 0

Figure 7. PANGOLIN's PFD window.

As SFC only requires to gather an abstraction of the trace during execution, the injected code for instrumentation causes minimal impact on performance. This means that PANGOLIN can be enhanced to feature PFD in an online fashion, where users are part of the analysis loop, and receive immediate feedback through the sunburst visualization after labeling each interaction. We have extended our tool to display an extra window during runtime, as shown in Figure 7, and the typical workflow of feature localization with PFD is as follows:

1. The analysis begins by running PANGOLIN. The subject application starts running and the PFD window appears.

2. To begin a transaction, we click the 'Start Transaction' button.

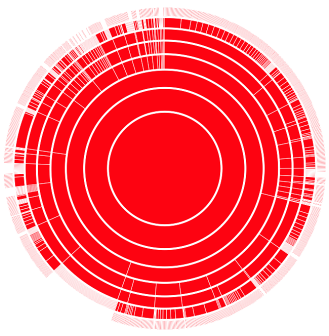
3. PANGOLIN starts recording the trace of the application. We can now interact with the application and execute the feature we are trying to locate.

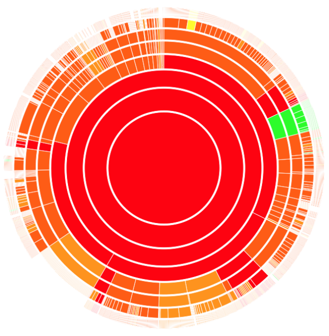
4. By pressing the ‘End Associated Transaction’ button, we are ending the current transaction, and labeling our interactions with the system as associated.

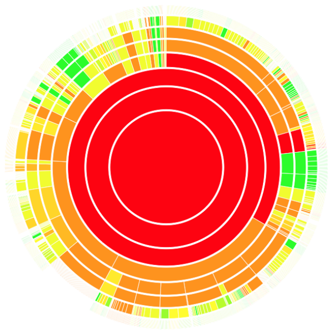
5. PANGOLIN runs the SFC analysis and displays the current results in its sunburst view. After one associated transaction, the visualization will look like Figure 8a, where every component in the trace has an association measure of 1.0.

6. After registering associated transactions, we need to capture dissociated interactions. To do so, we press the button ‘Start Transaction’, interact with the system without exercising the feature we are looking for, and then finish the transaction by pressing ‘End Dissociated Transaction’.

7. When a new transaction is recorded, PANGOLIN will automatically update the SFC analysis and the sunburst visualization. Step 6 can be regarded as a way to minimize the slice of code that needs inspection, and can be repeated by registering different interactions with the system.


(a) 1 Associated Transaction, 0 Dissociated Transactions.


(b) 1 Associated Transaction, 1 Dissociated Transaction.


(c) 1 Associated Transaction, 2 Dissociated Transactions.

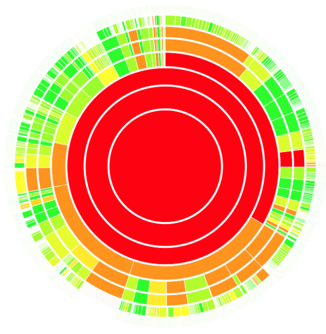

(d) 1 Associated Transaction, 3 Dissociated Transactions.

Figure 8. Sunburst report updated after each transaction.

Figures 8a to 8d show the impact of adding new dissociated transactions to the analysis. By showing the updated report after every transaction, this enables the user to determine when to stop interacting with the system. It is worth noting that adding multiple associated transactions can also be beneficial for minimizing the slice of code to be inspected, since SFC will rank the intersected code regions (labeled as (T1) in Section 5.2) as more likely to contain the implementation of the feature.

The PFD approach, since it requires user participation, is suited for analyzing interactive applications, in particular Graphical User Interface (GUI) applications, as most of their features can be triggered by interacting with the interface in some fashion.

6.2. *Case Study with JHotDraw*

In order to evaluate the PFD approach, we performed a case study with the JHotDraw project. In this case study, we analyze what is the gain in information when adding associated and dissociated transactions to find a pre-selected feature.

JHotDraw[††] is a customizable Java framework for editing drawings. It can be used for sketches, diagrams, and artistic drawings. For the case study, we used the latest version available at the time of writing (JHotDraw 7, revision 789). It is comprised by 688 classes and 65 packages, with over 82,000 lines of code. As opposed to the Rhino project, used in our user study detailed in Section 5, version 7 of JHotDraw does not currently contain any unit tests. Therefore, one could not use the non-PFD variant of PANGOLIN to locate its features.

The feature under consideration for this case study was the creation of triangle shapes in the drawing canvas. The aim of this exercise is to identify where that feature is implemented by just interacting with the application, and without having to inspect many spurious code locations. To do so, we recorded several associated transactions and dissociated transactions. Examples of associated transactions include creating a triangle shape of default size by single-clicking in the drawing area, creating a triangle with a custom size by click-dragging in the drawing area, and copy-pasting a triangle. Examples of dissociated transactions include changing an existing shape's color, resizing it or saving a drawing to a file.

With this case study, we aim to answer the following research questions:

RQ3: *Are manual user interactions an accurate input to the SFC analysis when test cases are unavailable?*

RQ4: *What is the impact of incorrect interaction classifications on the accuracy of the PFD report?*

In **RQ3** we want to assess if PFD can be considered as a fallback alternative in instances where there is no test suite available. **RQ4** is concerned with the viability of placing a human in the analysis loop, which consequently may lead to errors classifying system interactions.

6.3. *Evaluation*

We further detail the experiments performed to assess the accuracy of the participation-based extension of the SFC method. To evaluate our approach, we propose the metric $\mathcal{A}$ that quantifies the accuracy of the report generated by SFC. The metric $\mathcal{A}$ is given by

$$\mathcal{A} = \frac{\sum_i^{R^+} \varepsilon(i) \cdot s(i)}{|R^+|} \tag{3}$$

where R^+ is the list of components that were scored with non-zero association measure. $s(i)$ is the association measure for a given component i, and $\varepsilon(i)$ is a membership function that states whether component i is in fact part of the feature implementation. $\varepsilon(i)$ is given by

$$\varepsilon(i) = \begin{cases} 1 & \text{if component } i \text{ is part of the feature implementation} \\ -1 & \text{otherwise} \end{cases} \tag{4}$$

The value of $\mathcal{A}$ can range between -1 and 1. In the best case scenario, where every reported component has a score of 1.0 and is a member of the feature implementation, the value of $\mathcal{A}$ would be 1. In the worst case scenario, the report would be solely comprised of dissociated components, and its accuracy $\mathcal{A}$ would be -1. In this case study, we use the statement-level as the component granularity.

The first experiment was set up in order to emulate the conventional use of PFD, where users would start by exercising the feature in the first transaction, and after that they would record

[††]Available at `http://sourceforge.net/projects/jhotdraw/`.

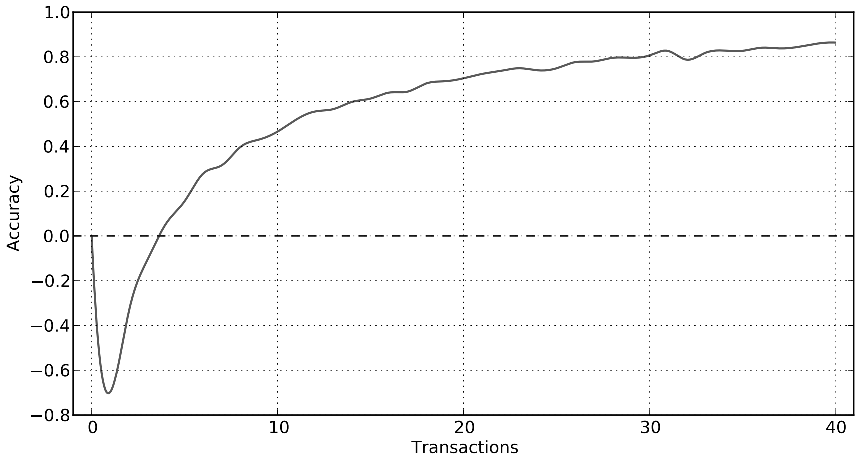
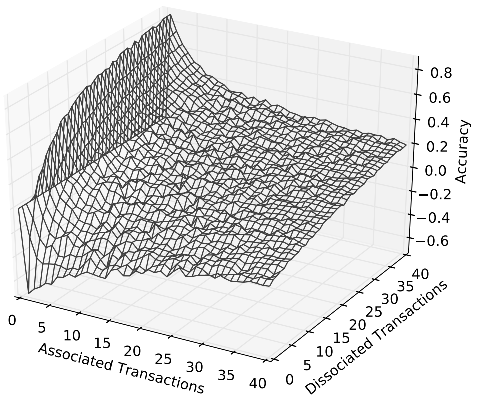


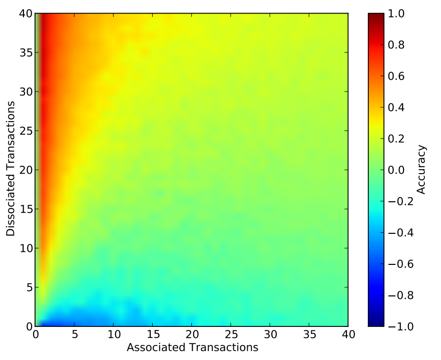
Figure 9. Accuracy metric for each recorded transaction.

several dissociated interactions with the aim of reducing their code inspection efforts. Therefore, in this experiment we analyze the impact of adding new dissociated transactions, given that the first recorded transaction is an associated one. The results are shown in Figure 9. As to be expected, the first transaction – the associated transaction where the feature is exercised – causes a steep dip in the accuracy graph. Since at this time there is only one recorded execution and since there are typically many more dissociated components that are active in a single execution trace, this means that the accuracy $\mathcal{A}$ will be negative. In the case of the triangle shape creation feature for JHotDraw, the accuracy $\mathcal{A}$ drops to a value close to -0.7.

When dissociated transactions are added, we see a steady increase in the accuracy of the SFC analysis with PFD. This happens because, as different transaction traces are added to the similarity-based analysis, we are reducing the cardinality of the set of components that are active exclusively in the associated transaction. This means that the association measure for dissociated components will decrease and, conversely, associated components will remain with a high association measure. We see that, by collecting a low amount of transactions (about 20 in the case of the triangle shape creation), we are able to achieve a high accuracy in the feature localization.



(a) 3D plot.



(b) Heat map.

Figure 10. Accuracy metric when changing the cardinality of associated and dissociated transactions.

In the experiment above, we have only considered one transaction as associated, and analyzed the impact on accuracy of adding dissociated transactions. We have also performed another experiment where we vary the number of associated and dissociated transactions in the feature localization. Results are presented in Figure 10, under the format of a 3D plot (Figure 10a), and as an heat map (Figure 10b). We conclude that having more dissociated transactions yields a better accuracy than having more associated ones. This happens because, generally, two associated traces are much more similar than an associated trace and a dissociated trace. Therefore, a dissociated trace is better at exonerating components and reducing their score. This means that there is more information gained by adding a dissociated transaction than by adding associated transactions to the analysis.

Revisiting the third research question:

RQ3: *Are manual user interactions an accurate input to the SFC analysis when test cases are unavailable?*

Allowing users to manually execute interactive applications and label their interactions as associated or dissociated can be an accurate alternative to the automated test suite. We found that a small number of transactions is enough to achieve considerable accuracy and that the approach is resilient to classification mistakes. We also found that the information gained by adding one dissociated transaction is higher than adding one associated transaction. Therefore, the best scenario to achieve accurate results with the participatory approach is to collect just a few associated transactions, and to collect as many dissociated transactions as possible.

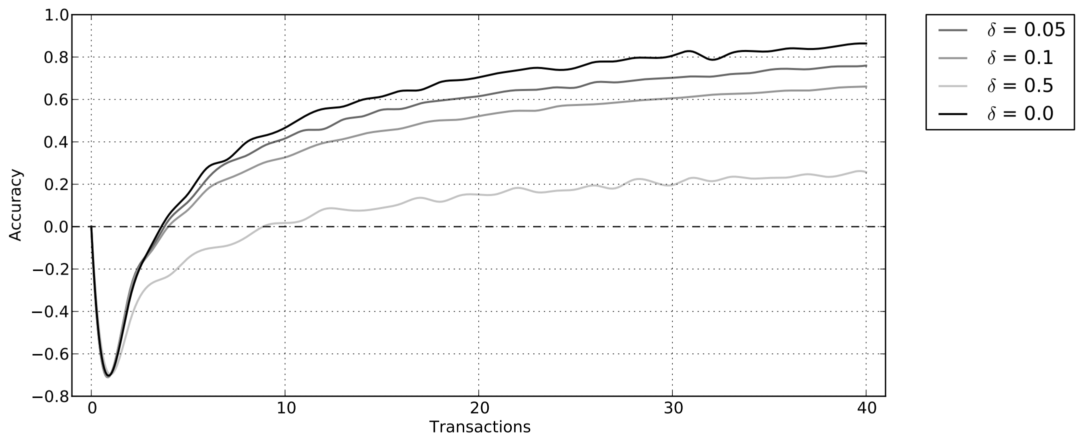


Figure 11. Impact of misclassification (δ) in PFD's accuracy.

Since this participation-based approach to SFC involves users in the analysis loop and requires them to interact with the system, PFD has to be resilient to eventual mistakes when labeling executions. To assess how our approach behaves when there are erroneous transactions present, we consider the concept of misclassification (δ). δ is the probability that a transaction labeled as dissociated by the user is, in fact, associated with the feature. For that, we have repeated the first experiment with δ equal to 0.05, 0.10 and 0.50. The results are shown in Figure 11.

Revisiting the fourth research question:

RQ4: *What is the impact of incorrect interaction classifications on the accuracy of the PFD report?*

Both $\delta = 0.05$ and $\delta = 0.10$ show some decrease in accuracy, but they remain, however, comparable to having $\delta = 0.0$. In fact, we are able to achieve similar accuracies if we increase the number of dissociated transactions in both misclassification scenarios. This means that if users are unsure if some of the transactions they are collecting are in fact dissociated, then they should add more transactions to mitigate this effect. At the extreme case of having $\delta = 0.50$, the results yielded by SFC are much more inaccurate. In such case, the score for the associated components decreases, because their activity pattern starts to differ from the (manually labeled) evolution vector.

6.4. Threats to validity

Construct Validity The case study was conducted by the authors. To ensure no bias in the execution of the experiment, one of the authors selected the feature. The other, which had no prior knowledge of JHotDraw's code, was responsible for running the PANGOLIN analysis. Misclassifications were labelled by checking each transaction's coverage and intersecting it with the feature implementation region.

Internal Validity Mozilla Rhino, the subject application for the experiment in the previous section, could not be used as the baseline for this study. A pre-requisite to use PFD is that the target applications need to have some kind of interactive interface so that the user can label different transactions (i.e., interactions). Rhino, being a javascript interpreter library for java, cannot be used in PFD mode, as it is not an interactive application. Instead, JHotDraw is the subject application selected for this case study. This application has also been used in previous related work [33]. One advantage of using JHotDraw is to show that a project with no test cases can be analyzed using PFD. In fact, this participatory approach is an attempt to increase the applicability domain of SFC by removing the requirement of having existing test cases. Instead, users are asked to interact with the application to generate coverage information that SFC can act upon.

External Validity The results presented may not generalize for all kinds of interactive applications. Due to the different ways applications can handle interaction events, classifying an interaction as associated/dissociated may not be trivial. Also, although it is shown that the approach is resilient with regards to varying levels of interaction misclassifications, there is, to our knowledge, no study on the actual value for the average misclassification rate for interactive applications.

7. RELATED WORK

Various techniques and tools were developed as a result of several years of research into trace visualization and feature localization [34, 35]. This section provides an overview – not meant to be exhaustive – of the related work in this area.

7.1. Trace Visualization

De Pauw *et al.* [36, 5] have developed a tool - termed Jinsight - for visually exploring a program's runtime behavior. Although this tool was shown to be useful for program comprehension, scalability concerns render the tool impractical for use in large applications. Reiss [6] states that execution traces are typically too large to be visualized and understood by the user. As such, Reiss proposed a way to select and compact trace data to improve the visualization's intelligibility. Live run-time visualizations have also been proposed as a way to reduce overheads [37], but make it harder to visualize entire executions.

Ducasse *et al.* [38] propose a way of representing condensed runtime metrics (such as attribute-usage frequencies, object allocation frequencies, object lifetime, among others) with the use of polymetric views. Greevy *et al.* [7] proposes a 3D visualization of the run-time traces of a software system. Greevy displays the amount of information about a component as a tower whose height is influenced by the amount of instances created. The main objective of this technique is to determine which system regions are involved in the execution of a certain feature, but the visualization may not be trivial to grasp.

Cornelissen *et al.* [39, 33] developed a tool - Extravis - that visualizes execution traces by employing two synchronized views: a circular bundle view for structural elements and an interactive overview via a sequence view. Its effectiveness was also demonstrated for three reverse engineering contexts: exploratory program comprehension, feature detection and feature comprehension. Pinzger *et al.* [40] proposes DA4Java, a tool that represents the source-code as a nested graph. Vertices in the graph represent code components, such as packages, classes and methods, and edges represent dependencies (*e.g.*, inheritance or method calls). Graph representations are also used in other works. Such is that of Yazdanshenas *et al.* [41], which like PANGOLIN, was able to visualize information flow at various abstraction levels. Ishio *et al.* [42] also used graphs to generate interprocedural data-flow paths. However, this analysis can also generate infeasible paths.

Trümper *et al.* [43] implemented the TraceDiff tool, to ease the comparison of large-scale system traces. The tool provides visualizations featuring a modified hierarchical edge-bundling layout and icicle plot-node aggregation, so that the scalability of large traces is addressed. Maletic *et al.* [44] proposes the MosaiCode tool, that uses a 2D metaphor to support the visualization and understanding of various aspects of large scale software systems. It supports multiple coordinated views of these systems and leverages a mosaic visualization to map their characteristics so that it is easy to understand by programmers, managers, and architects. Color and pixel maps are used to represent these characteristics such as lines of code, functions, files, and subsystems. MosaiCode is available as a stand-alone tool and is not integrated with a development environment.

Stengel *et al.* [45] developed the View Infinity tool. It provides a zoomable interface of *software product lines* (SPL). In SPL, software is implemented in terms of reusable user-visible characteristics, and is difficult to understand due to its variability. Like PANGOLIN, this tool offered a customizable granularity visualization, as well as a navigable interface.

The SFC approach proposed in this paper differs from the related work because of the low overhead necessary to compute the *association measures* for all the application's components. Another advantage is that it can not only pinpoint what components should be inspected and what components can be completely disregarded when evolving a feature, but also can warn about the existence of functionality that is not properly modularized.

7.2. Feature Localization and Information-Foraging

Work related to locating features in code includes the software-reconnaissance approach proposed by Wilde *et al.* [46, 47]. This approach tries to answer the question *"In which parts of this program is functionality X implemented?"* using only dynamic information, namely execution traces. Similarly to SFC, the Software Reconnaissance approach distinguishes two sets of test cases (or scenarios): scenarios that activate the feature, and scenarios that do not activate the feature. The former are used to locate the portions of code that implement the feature and the latter are used to reduce the size of those code portions. The SFC approach differs from this approach as it uses similarity coefficients, such as Ochiai, to assert the association of each line of code to the feature. Another approach to feature localization, coined SNIAFL, is proposed in the work of Zhao *et al.* [48]. This approach uses static techniques to locate features in the source code. The main idea behind the approach is to use IR to reveal the basic connection between features and computational units in source code, and is based on the premise that programmers use meaningful names as classifiers.

Information-foraging-based theories to explain information-seeking strategies have been used in the context of program comprehension and software engineering before. Relevant works include those of Ko *et al.* [49] in the context of software maintenance; Romero *et al.* [50], Lawrance *et al.* [27], Flemming *et al.* [28] and Piorkowski *et al.* [51] in software debugging; Chi *et al.* [52] and Spool *et al.* [53] for website design and evaluation.

8. CONCLUSIONS

This work tackles the challenge of understanding and locating code features pre-existing source code by leveraging concepts and algorithms from the field of software debugging research. One of the most often-used approaches for fault localization is Spectrum-based Fault Localization (SFL), an approach that exploits run-time information of system executions (such as test-cases) to estimate the likelihood of each component being at fault. In this paper, we provide a mapping between the problem of fault localization and the problem of locating features in the code, and present an approach, coined Spectrum-based Feature Comprehension (SFC), inspired by SFL. The main difference between the two approaches is that, instead of calculating the similarity to failure patterns, SFC calculates the similarity to feature usage patterns. SFC can thus rank components by their likelihood of being part of a feature's implementation and reducing the effort of code inspection.

The paper also presents PANGOLIN, an Eclipse plugin that implements the SFC analysis for Java projects. This tool, besides running test cases and performing SFC, also presents the analysis by means of a tree-based code visualization called Sunburst, where each component is color-coded according to their likelihood of being associated with the feature being analyzed.

To assess the effectiveness of SFC and PANGOLIN, a user study was carried out, where participants where asked to pinpoint components that need to be inspected when evolving/changing a certain feature in the Rhino open source project. Results show that participants using PANGOLIN were able to more accurately pinpoint code that implemented the feature and code that used it when compared to participants only using test coverage reports.

We also extended SFC to enable user participation in the analysis process. With the Participatory Feature Detection (PFD) approach users can, instead of using test cases, capture manual executions of the application and label them as associated or dissociated with the feature. This allows the use of feature localization for interactive applications, even if there are no automated tests available. Our experimental results lead us to conclude that users should strive to collect very few associated executions and many dissociated executions to achieve considerable accuracy, and that the approach is resilient to misclassifications by the user.

Regarding future work, we plan to extend the SFC report visualization to also provide other clues regarding component information, so that the perceived information gain of exploring a given component is augmented. One way to do this is by enhancing highlighted components with summaries of what they are responsible for. We plan to use code summarization-techniques that are based on stereotypes [54, 55].

ACKNOWLEDGEMENTS

We would like to thank the students from the Faculty of Engineering of University of Porto for participating in our user study.

REFERENCES

1. Corbi TA. Program understanding: Challenge for the 1990's. *IBM Systems Journal* 1989; **28**(2):294–306.
2. Tiarks R. What programmers really do - An observational study. *Softwaretechnik-Trends* 2011; **31**(2).
3. Lange D, Nakamura Y. Object-oriented program tracing and visualization. *Computer* 1997; **30**(5):63–70.
4. Renieris M, Reiss SP. Almost: Exploring program traces. *Proceedings of Workshop on New Paradigms in Information Visualization and Manipulation (NPIVM'99)*, 1999; 70–77.
5. De Pauw W, Lorenz D, Vlissides J, Wegman M. Execution patterns in object-oriented visualization. *Proceedings of Conference on Object-Oriented Technologies and Systems (COOTS'98)*, 1998; 219–234.
6. Reiss SP, Renieris M. Encoding program executions. *Proceedings of International Conference on Software Engineering (ICSE'01)*, 2001; 221–230.
7. Greevy O, Lanza M, Wysseier C. Visualizing live software systems in 3D. *Proceedings of ACM Symposium on Software Visualization (SoftVis '06)*, 2006; 47–56.
8. Zaidman A. Scalability solutions for program comprehension through dynamic analysis. *Proceedings of European Conference on Software Maintenance and Reengineering (CSMR'06)*, 2006; 327–330.
9. Abreu R, Zoeteweij P, Golsteijn R, van Gemund A. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 2009; **82**(11):1780–1792.
10. Pirolli P. *Information Foraging Theory: Adaptive Interaction with Information*. 1 edn., Oxford University Press, 2007.
11. Perez A, Abreu R. A diagnosis-based approach to software comprehension. *Proceedings of International Conference on Program Comprehension, ICPC'14*, 2014; 37–47.
12. Abreu R, Zoeteweij P, van Gemund A. On the accuracy of spectrum-based fault localization. *Proceedings of the Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC PART'07)*, 2007; 89–98.
13. Reps T, Ball T, Das M, Larus J. The use of program profiling for software maintenance with applications to the year 2000 problem. *Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC'97/FSE'5)*, 1997; 432–449.
14. Harrold M, Rothermel G, Sayre K, Wu R, Yi L. An empirical investigation of the relationship between fault-revealing test behavior and differences in program spectra. *Journal of Software Testing, Verification, and Reliability* 2000; (3):171–194.
15. Jain A, Dubes R. *Algorithms for clustering data*. Prentice-Hall, 1988.

16. Abreu R, Zoeteweij P, van Gemund A. An evaluation of similarity coefficients for software fault localization. *Proceedings of Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, 2006; 39–46.
17. da Silva Meyer A, Garcia A, de Souza A, de Souza C. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (zea mays l.). *Genetics and Molecular Biology* 2004; **27**:83–91.
18. Naish L, Lee H, Ramamohanarao K. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology* 2011; **20**(3):11.
19. Lucia, Lo D, Jiang L, Thung F, Budi A. Extended comprehensive study of association measures for fault localization. *Journal of Software: Evolution and Process* 2014; **26**(2):172–219.
20. Le TB, Lo D. Will fault localization work for these failures? an automated approach to predict effectiveness of fault localization tools. *2013 IEEE International Conference on Software Maintenance*, 2013; 310–319.
21. Janssen T, Abreu R, van Gemund A. Zoltar: A Toolset for Automatic Fault Localization. *Proceedings of International Conference on Automated Software Engineering (ASE'09)*, 2009; 662–664.
22. Campos J, Riboira A, Perez A, Abreu R. GZoltar: An eclipse plug-in for testing and debugging. *Proceedings of International Conference on Automated Software Engineering (ASE'12)*, 2012; 378–381.
23. Gouveia C, Campos J, Abreu R. Using HTML5 Visualizations in Software Fault Localization. *Proceedings of IEEE Working Conference on Software Visualization (VISSOFT'13)*, 2013; 1–10.
24. Campos J, Abreu R. Leveraging a Constraint Solver for Minimizing Test Suites. *Proceedings of International Conference on Quality Software (QSIC '13)*, 2013; 253–259.
25. Apel S, Batory D, Kästner C, Saake G. *Feature-Oriented Software Product Lines*. Springer, 2013.
26. Sjøberg D, Dybå T, Anda B, Hannay J. Building theories in software engineering. *Guide to Advanced Empirical Software Engineering*. Springer London, 2008; 312–336.
27. Lawrance J, Bogart C, Burnett M, Bellamy R, Rector K, Fleming S. How programmers debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering* 2013; **39**(2):197–215.
28. Fleming S, Scaffidi C, Piorkowski D, Burnett M, Bellamy R, Lawrance J, Kwan I. An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. *ACM Transactions on Software Engineering and Methodology* 2013; **22**(2):14:1–14:41.
29. Manning C, Raghavan P, Schütze H. *Introduction to information retrieval*. Cambridge University Press, 2008.
30. Eaddy M, Aho AV, Antoniol G, Guéhéneuc YG. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, ICPC '08, 2008; 53–62.
31. Eaddy M, Zimmermann T, Sherwood K, Garg V, Murphy G, Nagappan N, Aho A. Do crosscutting concerns cause defects? *Software Engineering, IEEE Transactions on* July 2008; **34**(4):497–515.
32. Zhang Y, Rilling J, Haarslev V. An ontology-based approach to software comprehension - reasoning about security concerns. *Computer Software and Applications Conference, 2006. COMPSAC '06. 30th Annual International*, vol. 1, 2006; 333–342.
33. Cornelissen B, Zaidman A, Holten D, Moonen L, van Deursen A, van Wijk JJ. Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software* 2008; **81**(12):2252–2268.
34. Cornelissen B, Zaidman A, van Deursen A, Moonen L, Koschke R. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering* 2009; **35**(5):684–702.
35. Dit B, Revelle M, Gethers M, Poshyvanyk D. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process* 2013; **25**(1):53–95.
36. De Pauw W, Helm R, Kimelman D, Vlissides J. Visualizing the behavior of object-oriented systems. *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, 1993; 326–337.
37. Reiss SP. Visualizing java in action. *Proceedings of ACM Symposium on Software Visualization (SoftVis'03)*, 2003; 57–65.
38. Ducasse S, Lanza M, Bertuli R. High-level polymetric views of condensed run-time information. *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR'04)*, 2004; 309–318.
39. Cornelissen B, Holten D, Zaidman A, Moonen L, van Wijk JJ, van Deursen A. Understanding execution traces using massive sequence and circular bundle views. *Proceedings of International Conference on Program Comprehension (ICPC'07)*, 2007; 49–58.
40. Pinzger M, Grafenhain K, Knab P, Gall H. A tool for visual understanding of source code dependencies. *Proceedings of International Conference on Program Comprehension (ICPC'08)*, 2008; 254–259.
41. Yazdanshenas A, Moonen L. Tracking and visualizing information flow in component-based systems. *Proceedings of International Conference on Program Comprehension (ICPC'12)*, 2012; 143–152.
42. Ishio T, Etsuda S, Inoue K. A lightweight visualization of interprocedural data-flow paths for source code reading. *Proceedings of International Conference on Program Comprehension (ICPC'12)*, 2012; 37–46.
43. Trümper J, Döllner J, Telea A. Multiscale visual comparison of execution traces. *Proceedings of International Conference on Program Comprehension (ICPC'13)*, 2013; 53–62.
44. Maletic J, Mosora D, Newman C, Collard M, ASutton, Robinson B. MosaiCode: Visualizing large scale software: A tool demonstration. *Proceedings of International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'11)*, 2011; 1–4.
45. Stengel M, Frisch M, Apel S, Feigenspan J, Kästner C, Dachselt R. View infinity: A zoomable interface for feature-oriented software development. *Proceedings of International Conference on Software Engineering (ICSE'11)*, 2011; 1031–1033.
46. Wilde N, Gomez J, Gust T, Strasburg D. Locating user functionality in old code. *Proceedings of Conference on Software Maintenance*, 1992; 200–205.
47. Wilde N, Scully M. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice* 1995; **7**(1):49–62.
48. Zhao W, Zhang L, Liu Y, Sun J, Yang F. SNIAFL: Towards a static noninteractive approach to feature location. *ACM Transactions on Software Engineering and Methodology* 2006; **15**(2):195–226.

49. Ko A, Myers B, Coblenz M, Aung H. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering* 2006; **32**(12):971–987.
50. Romero P, du Boulay B, Cox R, Lutz R, Bryant S. Debugging strategies and tactics in a multi-representation software environment. *International Journal of Man-Machine Studies* 2007; **65**(12):992–1009.
51. Piorkowski D, Fleming S, Scaffidi C, Bogart C, Burnett M, John B, Bellamy R, Swart C. Reactive information foraging: an empirical investigation of theory-based recommender systems for programmers. *Proceedings of Conference on Human Factors in Computing Systems (CHI'12)*, 2012; 1471–1480.
52. Chi E, Pirolli P, Chen K, Pitkow J. Using information scent to model user information needs and actions and the web. *Proceedings of Conference on Human Factors in Computing Systems (CHI'01)*, 2001; 490–497.
53. Spool J, Perfetti C, Brittan D. *Designing for the scent of information*. User Interface Engineering, 2004.
54. Moreno L, Aponte J, Sridhara G, Marcus A, Pollock L, Vijay-Shanker K. Automatic generation of natural language summaries for java classes. *Proceedings of International Conference on Program Comprehension (ICPC'13)*, 2013; 23–32.
55. Alhindawi N, Dragan N, Collard M, Maletic J. Improving feature location by enhancing source code with stereotypes. *Proceedings of International Conference on Software Maintenance (ICSM'13)*, 2013; 300–309.