Cues for Scent Intensification in Debugging

Alexandre Perez
Department of Informatics Engineering
Faculty of Engineering, University of Porto
Porto, Portugal
alexandre.perez@fe.up.pt

Rui Abreu
Department of Informatics Engineering
Faculty of Engineering, University of Porto
Porto, Portugal
rui@computer.org

***Abstract*—Information foraging is a theory to understand how people search for information. In this theory, information scent is the perceived likelihood by the "predator" that a cue will lead to a "prey". The better the cues, the better the information scent. In automatic debugging, it is the perceived likelihood that the diagnostic report leads to the cause of failures. In this paper, we detail a visualization, offered by the GZOLTAR toolset, that has the potential to provide better cues. With better we mean providing more information that leads to the fault than, e.g., the source code and code coverage information. The toolset provides a graphical display of the diagnostic reports yielded by well-known debugging techniques. From an information foraging point of view, we argue that the visualization is of added value while debugging. Finally, we report a user study to confirm that GZOLTAR's visualization provides better cues for pinpointing faults.**

Keywords*—Fault localization, information foraging, visualizations, user experience.

I. INTRODUCTION

Several coverage-based techniques that aid developers in finding software faults by computing a ranked list of possible candidates (e.g., source code statements) have been proposed in the past [1], [2], [3], [4][1]. Although these techniques have been shown to work well *in practice* (which actually should be read as: in the controlled environments used to assess their *diagnostic accuracy*), successful stories in transferring this technology to industry are yet to come.

One of the main barriers for the lack of world-wide adoption is certainly because automatic debugging researchers have not considered how developers debug *in practice*. Apart from just a few works, such as Whyline [5], Hipikat [6], and Mylyn[2], researchers have (strongly) assumed that (i) developers traverse the ranked list of suspicious statements; and (ii) perfect bug understanding. In fact, a recent empirical study has demonstrated that the assumptions just outlined do not hold in practice [7]. Another study, in an industrial setting, has shown interesting results [1], [8]: a candidate ranking that was considered to be close-to-perfect (given the metrics used to assess efficiency), was actually deemed by developers as not very useful. We believe that this happens because developers have a mental model of the software behavior and structure in their heads, and will most likely come up with an hypothesis that explain the observed failures. Therefore, the automatic technique is often used as an *oracle* to either confirm or refute the hypothesis.

These studies make it clear that state-of-the-art software fault localization techniques do not take into account how developers seek information in the source code. And therefore, despite advancing the field, these techniques have experienced a strong resistance from the industry. In this paper, instead of simply displaying the ranking in plain text, we propose to use a graphical interface integrated in the development environment. Exploiting the fact that the code is inherently hierarchical, we discuss the use of a Sunburst-like visualization that depicts the code structure to developers, previously proposed in [9]. We enhanced the visualization to also display the suspiciousness of a software component of being faulty, according to the output of fault localization techniques. Then, we use an information foraging theory to discuss the added value of such visualization. Information foraging is a theory to explain and predict how people use environmental information to achieve their goals. In this paper, the goal is to find software defects, whereas the environment information is the yielded visual diagnostic report, the features provided by the IDE (such as opening editor in a given line), and the source code of the software under analysis. The main contributions of this paper are

- We discuss an hierarchical graphical representation of the program, displaying not only its structure, but also the suspiciousness of each component being faulty;
- We offer the graphical representation in the GZOLTAR toolset, an Eclipse plugin for automating the testing and debugging phases;
- We propose and discuss an information foraging theory to motivate the choice for our graphical representation of the software;
- We perform an user study to confirm that the visualization brings extra cues while debugging.

Information foraging theories in software engineering have been considered before, including to explain how developers debug software [10], [11], but not the usefulness of visualizations in this domain. Also, while the use of hierarchical visualizations such as Sunburst have been proposed in [9], an explanation and discussion on their added value is still needed. In this paper, we map debugging concepts to an information foraging theory to provide an insight on the benefit of using visualizations to locate faults.

[1]Other techniques exist, but they are not the focus of the paper. However, the contributions of this paper may generalize to other techniques just as well.
[2]http://www.eclipse.org/mylyn/

II. Fault Localization

Spectrum-based Fault Localization (SFL) is amongst the most effective statistical techniques for software fault localization [1]. It exploits information from program executions to compute a list of suspicious software components (statements in the context of this paper), sorted by their suspiciousness of being faulty. In SFL, the following is given:

- A set $\mathcal{C} = \{c_1, c_2, \cdots, c_M\}$ of M components.
- A set $\mathcal{R} = \{r_1, r_2, \cdots, r_N\}$ of N program executions. Execution outcomes are gathered in a N-length error vector e, where $e_i = 1$ if execution r_i fails, and 0 if r_i passes. The criteria for determining if an execution has passed or failed can be from a variety of different sources, namely test case results and program assertions, among others.
- A $N \times M$ coverage matrix A, where $A_{ij} = 1$ if execution r_i involves component c_j, and 0 otherwise. This matrix is also called the *hit-spectra* matrix.

The fault localization consists in identifying what columns of the coverage matrix A resemble the error vector e the most. For that, several different similarity coefficients can be used [12]. One of the most effective is the Ochiai coefficient [13], used in the molecular biology domain:

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

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

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

The computed similarity coefficients are then used to rank system components according to their suspiciousness of being faulty. A list of components, sorted by their similarity coefficient, is presented to the user, helping prioritize the inspection of software components to pinpoint the root cause of failures.

SFL can be used with *hit-spectra* of several different software component granularities. However, it is most commonly used at the statement level and at the basic block level. Using coarser granularities would be difficult for programmers to investigate if a given fault hypothesis generated by SFL was, in fact, faulty. Throughout this work, we use statement level as the component granularity for the fault localization diagnosis report.

III. The Sunburst Visualization

The reason behind offering a visualization for the diagnostic reports is mainly because it is extremely difficult to interpret a ranked list of components and associated suspiciousness in plain text. In fact, just using the plain text, we deliberately discard important information such as the structure of the software system. As software is inherently hierarchical, a visualization leveraging such hierarchical information has been implemented. Furthermore, both to facilitate adoption and also let the developer use IDE features, we offer the visualization within our own GZoltar Eclipse Plug-in [14] for automatic testing and debugging[3].

We have selected the sunburst visualization to depict the program, as it allows the representation of hierarchical dependencies between components of the system. Each ring denotes a hierarchical level (or granularity) of the source code. See Figure 1 for a more intuitive description. The color of each component represents its failure suspiciousness, ranging from green - no fault suspiciousness - to red - high fault suspiciousness. This fault likelihood is computed by the diagnostic algorithm outlined in the previous section.

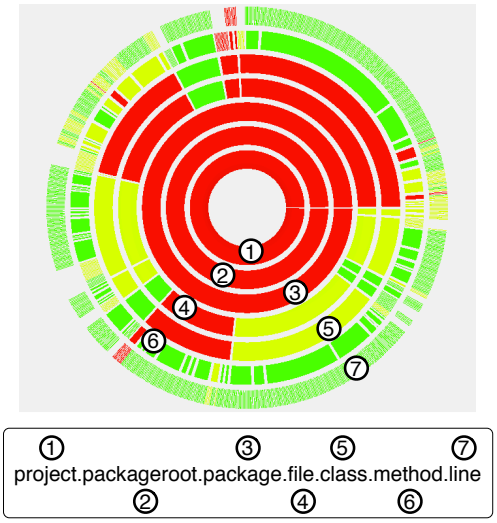

Fig. 1. Sunburst Hierarchical Levels.

Navigating through the visualization is straightforward. Developers can *expand* components of interest by simply `left-clicking` with the mouse in the desired components. Such action triggers an event in the graphical engine so that the *child* components are also shown. As an example, if the developer expands a component representing a class, the visualization will render the methods within that class.

Zooming in/out and panning the visualization to analyze in more detail a specific part or region of the software system is also possible. This feature is particularly interesting for very large projects as it may be difficult to properly visualize the outermost components (i.e., statements).

Another feature to improve understandability is the *root change* functionality. It allows any component to be the root of the visualization. The other sibling components, as well as parent components will be automatically hidden, and thus removed from the visualization. This feature is used, for example, when the developer has already pinpointed the cause of the failure to a given component and wants to analyze that component, and its dependencies, only. To perform this action, the developer simply `right-clicks` with the mouse in the root-to-be component.

If all the tests pass (i.e., there were no observed failures), the underlying fault localization technique yields an empty

[3] GZoltar is available online. To install the GZoltar toolset, users need to request a license at http://www.gzoltar.com/

diagnostic report. As such, our visualization will show all components in the system as green. It is worth noting that this does not mean that the system under test is bug-free, but rather that no failure was observed.

IV. Information Foraging Theory

Peter Pirolli, one of the pioneers of the theory, defines information foraging as a theory to both "explain and predict how people will best shape themselves for their information environments and how information environments can be shaped for people" [15]. Sjoberg *et al.* [16] 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. As a matter of fact, Information Foraging Theory is used to answer all these four questions.

To be able to use information foraging theory in the context of automatically produced diagnostic reports, we first need to map the theory constructs into this context. Following the information foraging theory proposed by Lawrence *et al.* [10], in this paper we map the information foraging theory constructs as follows:

Predator is the person debugging the program;
Prey is what the programmer seeks to know to pinpoint the bug;
Information patches are localities in the source code that may contain the fault;
Proximal cues are the runtime behaviors that suggest scent relative to the prey;
Information scent is the predator interpretation of the diagnostic report;
Topology is the collection of paths through the source code and diagnostic report through which the programmer can navigate. It also 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 the elements. As said before, we use the Sunburst visualization to represent the topology of the system. A Sunburst visualization is a radial space-filling visualization technique for displaying tree like structures. This is adequate to visualize software, as it can be regarded as a tree structure.

The navigable links between the elements allow the programmer to traverse the connection at the cost of just one click. This is important as information foraging draws from the theory that the developer's next move is one that maximizes the information leading to the prey.

Information foraging theory assumes that the developer's choices are an attempt to maximize the information gain per interaction's cost. As in [11], this can be characterized as

$$\text{choice} = \max(G/C)$$

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 root cause, the developer relies on the cues to decide which *place* to inspect next. Meaning that the developer uses those cues to estimate trade-off between the cost incurred and 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 the fault.

In information foraging, information scent is the perceived likelihood by the developer that a cue will potentially lead to the fault. 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 proposed in the previous chapter in regard to information foraging we may conclude that its visualization of the system's topology and its interaction features can indeed reduce the cost of navigation through the various system components (be it packages, classes, methods, even statements) and thus C is reduced. The color coding of each component, which is obtained from the fault localization ranking, can be regarded as a proximal cue, guiding the developer towards likely faulty regions of the source code and at the same time, notifying the developer about regions that should not be explored (where, *e.g.*, faulty executions have not touched). Hence, a better information scent is conveyed to the developer, increasing the information gain.

V. User Study

We carried out a user study to validate the usefulness of the visualization discussed in this paper. While some of the results of this study were previously detailed in [9], in this paper we investigate the use of information foraging to predict how developers navigate through the diagnostic report and source code when looking for a fault. This section details the user study and draws conclusions from the feedback given by the participants regarding the toolset offering the visualization.

A. Participants

We carried out a user study with 40 students of the Master in Informatics and Computing Engineering program at University of Porto. As already said, all participants were experienced developers in Java (more than 5 years; mostly as freelancers) and also used regularly the Eclipse IDE to develop and JUnit as the testing tool.

Participants were asked to locate and fix a fault in a software program. They have been split into two groups, each comprised of 20 subjects:

Control group This group was supposed to find and fix the fault using only the default IDE features provided by Eclipse. Among many other features, breakpoints and JUnit tests could be used.
Experimental group The experimental group had access to the GZoltar toolset, along with the sunburst visualization.

The participants had no previous experience with the GZoltar toolset. Before starting the user study, the main features of the toolset were briefly explained. We decided not to give a very detailed introduction in order to assess how intuitive the visualization is.

B. Subject Program

To evaluate the efficiency of our toolset and its visualization, we used the XStream[4] project as the subject for our user study. XStream is a library that (de)serializes Java objects into XML. None of the users were familiar with the XStream's source code before the user study. XStream version 1.4.4 has 17389 lines of code, 306 classes and 22 packages. The program also provides 1418 JUnit test cases.

We have injected a logic operator fault in the program: a *not equals* operator ("!=") was changed to an *equals* operator ("==") in line 455 of the `AnnotationMapper` class from the `com.thoughtworks.xstream.mapper` package. This fault allows the code to be compiled (a requirement to use GZOLTAR, since SFL performs a dynamic analysis), and leads to unexpected behavior. Participants were provided with all test cases, and a timeout of 30 minutes was set to find and fix the fault.

Developers were observed remotely while debugging to be able to better understand their movements and whether or not the visualization is of added value. We have also logged the participants' events and changes to the source code.

C. Results

We observed that 100% of the participants in the experimental group were able to successfully find and fix the injected fault within the time limit. On average, participants required, on average, $\mu = 7.9$ minutes to fully accomplish the task of finding and fixing the fault (with a standard deviation of $\sigma = 4.9$ minutes and a median time of $\bar{t} = 7.1$ minutes).

Regarding the control group, that performed the task using the more traditional methods, the results were rather different from those participants using the toolset. Only 35% of participants found and fixed the fault. The remaining 65% have not managed to find the fault within the time time limit. They were actually not even close to being able to pinpoint the root cause of observed failures, meaning that the information scent and cues provided by the source code and test cases was not as good as the one of the visualization. Feedback is that they would need more time to even comprehend the source code. For those that did not find the fault, were assigned as taking the maximum time (30 minutes). With this into account, the average time to perform the task was $\mu = 23.4$ minutes ($\sigma = 9.8$ and $\bar{t} = 30$). Therefore, this confirms that the sunburst visualization offered by GZOLTAR is of great value when doing testing and debugging (in particular, if the testing team is different from the development team), speeding up the debugging task.

These results suggest that the cues provided by the visualization increase the information scent of the developers, this way leading to the *prey* (faulty code) quickly. Our findings are in agreement with the ones reported in [10]: participants' pursuits of scent were triggered mostly in the source code. Moreover, our results suggest that the participants also resorted to the visualization to pursuit scent.

[4]XStream homepage http://xstream.codehaus.org/, 2013.

D. Feedback

In general, the GZOLTAR toolset had a good acceptance amongst the participants. The concepts underlying the toolset were well comprehended. The participants' reviews, added to the good results of the study, where the majority was able to reach the main goal, reveal that the GZOLTAR toolset is effective.

Participants were invited to give their opinions and suggestions for further improvements. The feedback obtained was regarded as very positive. Participants mentioned that the toolset was both efficient and effective. They also gave suggestions to improve in future releases, such as to improve it to work with other programming languages. With this experiment we were able to confirm the usefulness of this toolset. The scenario of this experiment was rather demanding, because participant had no previous contact with the toolset and XStream before. Nevertheless, the results were very promising, and participants showed to be pleased with the use of this toolset.

In this experiment, as discussed in [9], other visualizations were also tested. Also implemented in the toolset are the Vertical Partition and the Bubble Hierarchy visualizations. All participants from the experimental group interacted with the three visualizations, but quickly gravitated towards Sunburst to complete the debugging task. After the fact, participants stated that Sunburst was the most intuitive visualization. From an information foraging point of view, we argue that due to this intuitiveness, Sunburst provides better cues than the other visualizations and, ultimately, increases the information gained about the system being diagnosed.

It is worth noting that we have not explicitly asked which features of the toolset they used and/or found useful while searching the faulty statement because we monitored the fault localization process. Since such monitoring allowed us to verify the procedures/steps taken by the participants, we concluded that the tool was heavily relied upon. In fact, participants gave us feedback such as

> *"Without the visualization, it would be practically impossible to locate the fault."*
>
> *"The visualization helped me better understand the components' execution patterns."*
>
> *"The visualization and code editor interaction was fundamental to quickly find the fault."*

Therefore, we conclude that the visualization of the debugging report gives better cues to the developers, thus improving the scent, during the the debugging problem.

E. Threats to Validity

Empirical experiments have threats to the validity of their results. In the following we discuss threats to the validity of the empirical evaluation reported in this paper.

The external validity of the results obtained in the user study can be questioned given the fact that we have used a medium-sized, real program. It may be the case that the software program used has unusual characteristics that would not generalize to other programs. In order to strengthen the validity of our findings, we have applied the toolset to other, real world

experiments (as reported in, e.g., [17]), demonstrating that our toolset can be of great value during testing and debugging.

Our results may not generalize because of the fact that we have injected only one fault. The injected fault may again not generalize to all sorts of problems. We, however, think that it is an interesting fault because it is not easy to pinpoint. Yet another external validity is the fact that participants are all computer science students. Moreover, we have chosen a system written in Java because GZOLTAR only handles Java source code. Therefore, we cannot generalize the results to other programming languages.

The internal validity of the findings from the user study can be questioned given that fact that we have only briefly explained the toolset. Hence, some features provided by the tool may not have been completely understood and therefore misused during the experiment. Furthermore, the time limit is somewhat artificial and may put participants under pressure.

VI. Related Work

Automating the debugging process has been a hot topic in the last couple of years (e.g. [1], [18], [19], [20], [21], [22], [23]). However, despite the large body of work, most Integrated Development Environment (IDE)s still either offer limited or only manual debugging utilities, such as breakpoints. Amongst the most sophisticated IDEs is the JIVE toolset [24], which provides a representation of the execution history. The lack of debugging features in IDEs is particularly noticeable when considering state-of-the-art fault localization techniques.

Currently, one of the most well known automatic debugging toolset is Tarantula [25]. This tool relies on code coverage of multiple test executions - like the underlying technique considered in this paper. Its visualization resembles an overview of the entire code, and the color of each statement represents its failure probability. While the concepts intrinsic to Tarantula may be applied to many languages, this tool only works with C projects. Tarantula does not integrate with (J)Unit tests and is not integrated into an IDE.

Zoltar [26] is another available automatic debugging tool. Like Tarantula, it also relies on code coverage of multiple test executions and is a standalone tool that does not integrate (J)Unit tests. The results processed by this tool may be visualized using the command line interface to obtain an ordered list of statements, where the statements that are most likely to contain a fault are ranked first. It is also possible to use xZoltar [26] to visualize the source code with the most suspicious lines highlighted in red.

Vida [27] is an Eclipse plug-in based in the Tarantula toolset. It suggests places where breakpoints should be placed, considering the fault suspiciousness of each statement. The visualization offered represents the program, and it resembles the Tarantula standalone tool, with no interactive features.

EzUnit4 [28] is also an Eclipse plug-in that bases its execution on JUnit tests, and uses statistical analysis to calculate the failure probability of each tested method. It uses a combination of multiple weighted statistical metrics to create a failure ranking, presented as a view in Eclipse. The background color of each line of that ranking list ranges from green to red, according to its failure probability.

Finally, there are other tools that perform automatic debugging using other methodologies, such as the Delta Debugging [29], predicate-based, aspect-based and model-based debugging tools. Delta debugging integrates with Eclipse as a plug-in, and uses an algorithm that analyses software changes (input and code) to localize the faults. Other techniques for automatic debugging are as follows: as those based on predicates such as Cooperative Bug Isolation (CBI) [30] and Sober [3], based in aspects such as Bugdel [31] and based in models [32]. However, an off-the-shelf toolset offering these techniques is not publicly available. None of these concepts and tools offer an easy to use/understand visualization of the system, with the fault likelihood of each component, and integrated into an IDE.

As for information foraging theory, despite its success in the domain of Web foraging, only a few researchers have applied information foraging theory to software engineering problems. One recent study of how developers navigate source code used information foraging theory to interpret results from an empirically based model of program comprehension [33]. A second formative study mentioned that developers appear to look for documentation in a manner consistent with what information foraging theory advocates but did not mention how any specific information foraging theory constructs, such as scent, matched up with empirical observations [34].

To our knowledge, none of the techniques above considered information foraging in order to build their automatic debugging interface. We therefore kindly argue that our visualization to aid developers is better than the ones in the related work.

VII. Conclusion

In this paper, we discuss the added value of a visualization technique to display the diagnostic reports produced by the automatic coverage-based fault localization techniques, previously proposed in [9]. We argue that such visualization, and its integration in the Eclipse IDE as a plugin, is more intuitive and leads developers quickly to the defects responsible for observed failures than seeking the defects by just using the source code, coverage information, and IDE features (e.g., breakpoints). This visualization is intended to help the developer to interpret the diagnostic report of fault localization techniques. A hierarchical view of the system under test - called Sunburst - is shown to the developer. This view represents the hierarchical dependencies of the components of the system. Furthermore, developers can also explore the Sunburst visualization using many interaction features, such as zooming and panning, and changing the root component, as a way to abstract from the other components.

To explain the added value of the visualization, we use information foraging theory. Information foraging is a theory to explain and predict how people use environmental information to achieve their goals. It builds its hypothesis upon optimal foraging theory, drawing from noticed similarities between developers' information searching patterns and animal food foraging strategies. In this paper, the goal is to find software defects, whereas the environment information is the yielded visual diagnostic report, features provided by the IDE (such as opening editor in a given line), and the source code of the software under analysis.

Future work includes the following. In information foraging, information scent is the perceived likelihood by the

predator that a cue will potentially lead to a prey. That is, information scent is the programmer's perception of the value of information. We plan to enhance the visualization by providing more cues to the developer; hence leading to better information scent. We plan to do this by providing a summary of the class by computing the frequencies of words used in the class, allowing for a description of what each class does. We plan to use the concepts of term frequency (*tf*) and inverse-document frequency (*idf*), used in the field of Information Retrieval [35] as a way of weighing and ranking frequent words used in every class.

ACKNOWLEDGMENT

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

REFERENCES

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

[2] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, 2002, pp. 467–477.

[3] C. Liu, L. Fei, X. Yan, J. Han, and S. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Transactions on Software Engineering (TSE)*, vol. 32, pp. 831–848, 2006.

[4] E. Wong, T. Wei, Y. Qi, and L. Zhao, "A crosstab-based statistical method for effective fault localization," in *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST'08)*, 2008, pp. 42–51.

[5] A. J. Ko and B. A. Myers, "Debugging reinvented: asking and answering why and why not questions about program behavior," in *Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 301–310.

[6] D. Cubranic, G. Murphy, J. Singer, and K. Booth, "Hipikat: a project memory for software development," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 446 – 465, june 2005.

[7] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011, pp. 199–209.

[8] P. Zoeteweij, R. Abreu, R. Golsteijn, and A. J. van Gemund, "Diagnosis of embedded software using program spectra," in *Engineering of Computer-Based Systems, 2007. ECBS '07. 14th Annual IEEE International Conference and Workshops on the*, Mar. 2007, pp. 213–220.

[9] C. Gouveia, J. Campos, and R. Abreu, "Using HTML5 Visualizations in Software Fault Localization," in *Proceedings of IEEE Working Conference on Software Visualization (VISSOFT'13)*, 2013, to appear.

[10] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming, "How programmers debug, revisited: An information foraging theory perspective," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 197–215, 2013.

[11] S. D. Fleming, C. Scaffidi, D. Piorkowski, M. Burnett, R. Bellamy, J. Lawrance, and I. Kwan, "An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 2, 2013.

[12] A. K. Jain and R. C. Dubes, *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

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

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

[15] P. L. T. Pirolli, *Information Foraging Theory: Adaptive Interaction with Information*, 1st ed. Oxford University Press, Inc., 2007.

[16] D. I. Sjøberg, T. Dybå, B. C. Anda, and J. E. Hannay, "Building theories in software engineering," in *Guide to Advanced Empirical Software Engineering*. Springer London, 2008, pp. 312–336.

[17] J. Campos, "Regression Testing with GZoltar: Techniques for Test Suite Minimization, Selection, and Prioritization," MSc Thesis, University of Porto, 2012.

[18] B. Liblit, "Cooperative debugging with five hundred million test cases," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'08)*, 2008, pp. 119–120.

[19] M. Nica and F. Wotawa, "From constraint representations of sequential code and program annotations to their use in debugging," in *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI'08)*, vol. 178, 2008, pp. 797–798.

[20] M. Burger and A. Zeller, "Minimizing reproduction of software failures," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA'11)*, 2011, pp. 221–231.

[21] G. K. Baah, A. Podgurski, and M. J. Harrold, "Causal inference for statistical fault localization," in *Proceedings of the 19th international symposium on Software testing and analysis (ISSTA'10)*, ser. ISSTA '10, 2010, pp. 73–84.

[22] ——, "Mitigating the confounding effects of program dependences for effective fault localization," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC-FSE'11)*, 2011, pp. 146–156.

[23] J. Rößler, G. Fraser, A. Zeller, and A. Orso, "Isolating failure causes through test case generation," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA'12)*, 2012, pp. 309–319.

[24] J. K. Czyz and B. Jayaraman, "Declarative and visual debugging in Eclipse," in *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, 2007, pp. 31–35.

[25] J. A. Jones, M. J. Harrold, and J. T. Stasko, "Visualization for Fault Localization," in *Proceedings of ICSE 2001 Workshop on Software Visualization Toronto Ontario Canada*, 2001, pp. 71–75.

[26] T. Janssen, R. Abreu, and A. Gemund, "Zoltar: A Toolset for Automatic Fault Localization," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, Nov. 2009, pp. 662–664.

[27] D. Hao, L. Zhang, L. Zhang, J. Sun, and H. Mei, "VIDA: Visual interactive debugging," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 583–586.

[28] P. Bouillon, J. Krinke, N. Meyer, and F. Steimann, "EzUnit: A Framework for Associating Failed Unit Tests with Potential Programming Errors," in *Agile Processes in Software Engineering and Extreme Programming*, 2007, vol. 4536, pp. 101–104.

[29] P. Bouillon, M. Burger, and A. Zeller, "Automated debugging in Eclipse: (at the touch of not even a button)," in *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, 2003, pp. 1–5.

[30] B. R. Liblit, "Cooperative Bug Isolation," Ph.D. dissertation, University of California, Berkeley, Dec. 2004.

[31] Y. Usui and S. Chiba, "Bugdel: an aspect-oriented debugging system," in *Software Engineering Conference, 2005. APSEC '05. 12th Asia-Pacific*, 2005.

[32] W. Mayer and M. Stumptner, "Evaluating Models for Model-Based Debugging," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 128–137.

[33] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Trans. Softw. Eng.*, vol. 32, no. 12, pp. 971–987, Dec. 2006.

[34] J. Brandt, P. J. Guo, J. Lewenstein, and S. R. Klemmer, "Opportunistic programming: how rapid ideation and prototyping occur in practice," in *Proceedings of the 4th international workshop on End-user software engineering*, 2008, pp. 1–5.

[35] R. Baeza-Yates and B. Ribeiro-Neto, *Modern information retrieval*. Addison Wesley, 1999.