Pangolin: An SFL-based Toolset for Feature Localization

Bruno Castro
IST, University of Lisbon
Lisbon, Portugal
bruno.castro.machado@tecnico.ulisboa.pt

Alexandre Perez
Palo Alto Research Center
USA
aperez@parc.com

Rui Abreu
IST, University of Lisbon and INESC-ID
Lisbon, Portugal
rui@computer.org

***Abstract*—Pinpointing the location where a given unit of functionality—or feature—was implemented is a demanding and time-consuming task, yet prevalent in most software maintenance or evolution efforts. To that extent, we present PANGOLIN, an Eclipse plugin that helps developers identifying features among the source code. It borrows Spectrum-based Fault Localization techniques from the software diagnosis research field by framing feature localization as a diagnostic problem. PANGOLIN prompts users to label system executions based on feature involvement, and subsequently presents its spectrum-based feature localization analysis to users with the aid of a color-coded, hierarchic, and navigable visualization which was shown to be effective at conveying diagnostic information to users. Our evaluation shows that PANGOLIN accurately pinpoints feature implementations and is resilient to misclassifications by users. The tool can be downloaded at https://tqrg.github.io/pangolin/.**

***Demonstration video:* https://youtu.be/vvTCCrzyRg8**

***Index Terms*—Spectrum-based Fault Localization, Program Understanding, Maintenance and Evolution**

I. INTRODUCTION

Let us consider a software feature to be *"a unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option"* [1]. Typically, in software maintenance or evolution scenarios, developers need to *find* features in the source code to further develop them or possibly revise them. It is not uncommon for developers to spend 60% to 80% of their time in such feature location and comprehension tasks [2]. In fact, 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 do engineers gain sufficient understanding of the application, enabling them to seek, gather, and make use of available information to efficiently conduct maintenance or evolution tasks.

Previous work has shown that features can be located using execution scenarios of the system [3], [4]. In fact, recently, it was argued that the process of finding features is indeed very similar to the process of diagnosing software faults: in both cases, developers need to pinpoint the root cause of a particular software *behavior* [5], [6]. Therefore, it was hypothesized that feature localization tasks can be framed as a software diagnosis effort. Consequently, an approach was proposed, coined Spectrum-based Feature Comprehension (SFC), that leverages concepts and techniques from the fault localization domain [5]. In particular, the approach maps the task feature localization to Spectrum-based Fault Localization (SFL). SFL exploits coverage information of test cases to calculate the likelihood of each software component (*i.e.*, the unit of software whose coverage is being measured) being faulty, following the assumption that components which are frequently involved in failing tests are more likely to contain the fault than ones that are frequently involved in passing tests. In a similar way, SFC measures the correlation between associated transactions (executions that exercise the feature) and component involvement in such transactions.

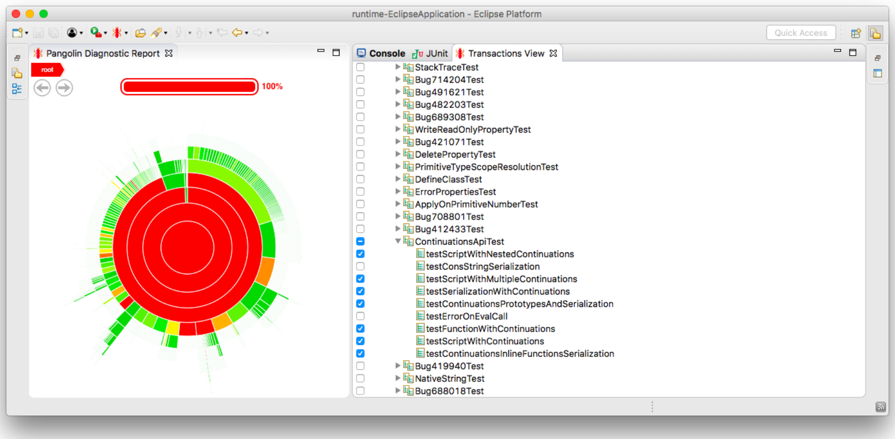
We present PANGOLIN, a toolset that provides an SFC implementation within the Eclipse Integrated Development Environment (IDE). PANGOLIN collects coverage information for every system execution (*e.g.*, every test case), prompts users to label executions regarding feature involvement, and subsequently performs the SFC analysis. The result of this analysis is shown to users via an interactive, hierarchic visualization which was previously shown to be effective at conveying diagnostic cues [7]. Our evaluation of SFC and the PANGOLIN plugin shows that the tool accurately locates features within the code and that it is able to maintain its accuracy when users misclassify feature involvement in system executions.

II. PANGOLIN

We now introduce the PANGOLIN toolset, an Eclipse plugin that implements the SFC approach, measuring the association between system executions (also referred to as *transactions*) that exercise a targeted feature and its involved components, displaying its results with the aid of a hierarchic, interactive visualization. It is based on the GZOLTAR toolset which enables fault localization within the Eclipse IDE [8].

The PANGOLIN plugin performs a lightweight dynamic analysis of Java programs by instrumenting the project's bytecode, so that per-transaction component involvement is recorded, in order to subsequently perform the SFC analysis. The plugin, as depicted in Figure 1, provides two new views

This material is based upon work supported by Fundação para a Ciência e a Tecnologia (FCT), with the reference PTDC/CCI-COM/29300/2017.


runtime-EclipseApplication - Eclipse Platform
Quick Access
Pangolin Diagnostic Report
root
100%
Console
JUnit
Transactions View

```
☐ ▸ StackTraceTest
☐ ▸ Bug714204Test
☐ ▸ Bug491621Test
☐ ▸ Bug482203Test
☐ ▸ Bug689308Test
☐ ▸ WriteReadOnlyPropertyTest
☐ ▸ Bug421071Test
☐ ▸ DeletePropertyTest
☐ ▸ PrimitiveTypeScopeResolutionTest
☐ ▸ DefineClassTest
☐ ▸ ErrorPropertiesTest
☐ ▸ ApplyOnPrimitiveNumberTest
☐ ▸ Bug708801Test
☐ ▸ Bug412433Test
⊟ ▾ ContinuationsApiTest
☑       testScriptWithNestedContinuations
☐       testConsStringSerialization
☑       testScriptWithMultipleContinuations
☑       testSerializationWithContinuations
☑       testContinuationsPrototypesAndSerialization
☐       testErrorOnEvalCall
☑       testFunctionWithContinuations
☑       testScriptWithContinuations
☑       testContinuationsInlineFunctionsSerialization
☐ ▸ Bug419940Test
☐ ▸ NativeStringTest
☐ ▸ Bug688018Test
```

Fig. 1: PANGOLIN plugin for the Eclipse IDE.

in the Eclipse IDE: a diagnostic report view and a transaction view. The former, shown to the left, visually depicts the SFC report in a tree-based, navigable visualization called sunburst. The latter view, shown to the right, lists all system transactions run by PANGOLIN and allows users to select which executions are associated with the desired feature.

The 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. The visualization is HTML5-based—and shown in a browser view within Eclipse—, ensuring that the visualization can be easily extended and can be ported to other IDEs and development environments. Sunburst was shown to be effective at conveying *actionable* diagnostic information (more so than mere list-based SFL reports) [7], [9]. Each element in the visualization is color coded according to the resulting *association measure*, as computed by the SFC approach [5]. The association measure quantifies the degree of correlation between a given component and the feature under consideration. Components are colored ranging from bright green if the association measure is close to zero (*i.e.*, low correlation with the feature); to yellow if the association measure is close to 0.5 and to red if it is close to 1 (indicating high correlation with the feature).

As sunburst is a hierarchical visualization, parent components exhibit the maximum association measure of all their child components. When a user hovers the mouse on a component its association measure—as well as the component's *canonical name*—is shown. Clicking a component prompts a *root change*, where the selected component becomes the root of the visualization, hiding all elements that are not descendants of the selected component. By double clicking a particular component, Eclipse's code editor will open and the cursor is positioned on the start of the chosen component. In order to find the features, users are advised to inspect components that exhibit high association measures.

PANGOLIN provides two modes of execution: test-based feature detection, where the project's test cases function as system transactions; and participatory feature detection, where users manually record their interactions with the system and label them as either associated or dissociated. These modes of execution will be detailed in the following subsections.

A. *Test-based Feature Detection*

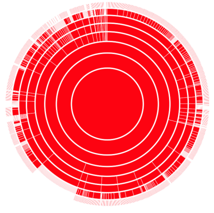
This mode of execution uses the project's JUnit test cases as the set of system transactions. It leverages the JUnit test runner within Eclipse's Java Development Tools to launch and execute test suite. To perform test-based feature detection with the PANGOLIN plugin, users should select the 'Run Pangolin As: JUnit Test' option in the contextual Run menu.

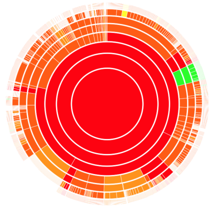
After test cases are run, the PANGOLIN's transactions view—shown to the right in Figure 1—is populated with all executed test cases. The test list is shown hierarchically: every test from the same enclosing test class is grouped together; similarly, each test class from the same package is also grouped. Note that each element in this view has an accompanying checkbox. In order to perform feature localization, users are prompted to identify which system transactions exercise the feature under consideration (*i.e.*, which transactions are *associated*) by clicking the respective checkbox. Every time a checkbox is selected or deselected, PANGOLIN's SFC engine is re-run, and the updated diagnostic report is shown in the sunburst visualization.

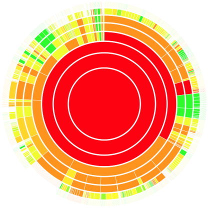
By default, test failures are automatically set as associated transactions. This means that developers can also use PANGOLIN as a fault localization tool, helping to prioritize the inspection of components that are likely to explain the observed failures. Users can override this default behavior by deselecting the associated transactions.

B. *Participatory Feature Detection*

The mode of execution described in the previous section assumes that (1) the project contains a thorough test suite, and that (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 not frequently available, which limits


(a) 1 Associated Transaction, 0 Dissociated Transactions.


(b) 1 Associated Transaction, 1 Dissociated Transaction.


(c) 1 Associated Transaction, 2 Dissociated Transactions.

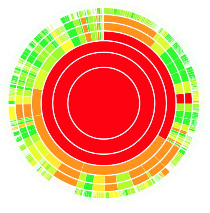

(d) 1 Associated Transaction, 3 Dissociated Transactions.

Fig. 2: Sunburst visualization updated after each transaction.

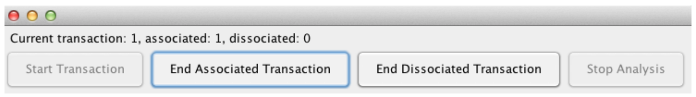

Current transaction: 1, associated: 1, dissociated: 0

Fig. 3: PANGOLIN's PFD control menu.

the applicability of the proposed approach. To address this concern in interactive applications, we have included another mode of execution to PANGOLIN, named Participatory Feature Detection (PFD). To perform PFD with the PANGOLIN plugin, users should select the 'Run Pangolin As: Java Application' option in the contextual Run menu.

The PFD technique, as described in [6], allows users to capture manual interactions with the system, enabling them to label each interaction as associated or dissociated with the feature they are trying to locate. As SFC only requires abstracted execution traces, the instrumentation code that is injected into the application causes minimal impact on performance. This means that PANGOLIN can be enhanced to feature user participation in an online fashion, where users are part of the analysis loop, receiving 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 3, and the typical workflow of feature localization with PFD is as follows:

1) To begin a transaction, users click the 'Start Transaction' button.
2) PANGOLIN starts recording the trace of the application.
3) By pressing the 'End Associated Transaction' button, users end the current transaction, labeling their interactions with the system as associated.
4) PANGOLIN runs the SFC analysis and displays the current results in its sunburst view. After one associated transaction, the visualization will look like Figure 2a, where most components in the trace have high association measures.
5) After registering associated transactions, users need to capture dissociated interactions. To do so, we press the button 'Start Transaction', interact with the system without exercising the feature they want to locate, and then finish the transaction by pressing 'End Dissociated Transaction'.
6) When a new transaction is recorded, PANGOLIN will automatically update the SFC analysis and the sunburst visualization. Step 5 can be regarded as a way to minimize the slice of code that needs inspection.

Figures 2a to 2d 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 as more likely to contain the implementation of the feature.

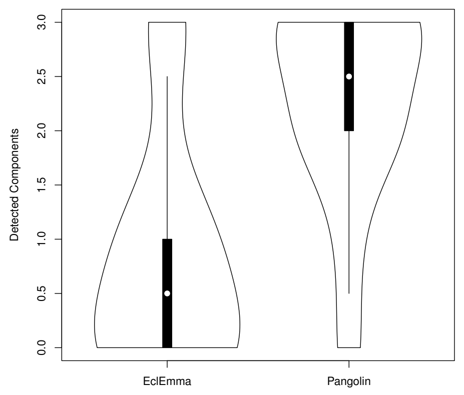
III. EVALUATION

To evaluate the effectiveness of SFC, we have performed a user study where 108 participants attempted to detect a specific feature of the Rhino project[1]. Rhino is a Javascript engine written entirely in Java. Participants were requested to pinpoint the implementation of the *Continuation* feature—responsible for storing and resuming a Javascript runtime. This feature is implemented in three distinct code locations. One group of participants was asked to use the PANGOLIN plugin to complete the feature localization task. The other group of participants was asked to use the features from a standard Eclipse IDE and its EclEmma[2] code-coverage plugin. In order to detect a feature using EclEmma, participants need to gather the code-coverage information for all tests that exercise the feature, and figure out their intersections.

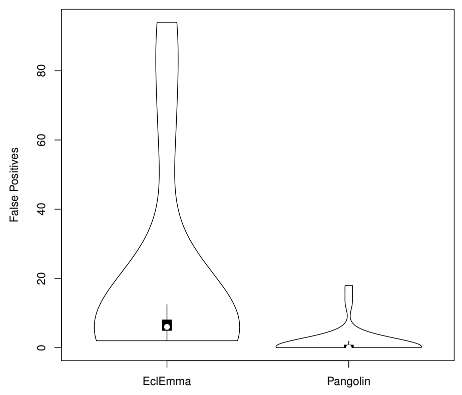
Figures 4a and 4b show violin plots depicting the number of detected feature components reported by the participants and the number of false positives for both groups of participants, respectively. Results show that the PANGOLIN plugin helped participants to more accurately pinpoint the three code locations responsible for the implementation of the feature, while leading to less false positives.

Since we ask users to classify transactions as associated or dissociated with the feature they want to locate, we also evaluate the impact of misclassification on feature localization

[1]Available at https://developer.mozilla.org/en-US/docs/Rhino
[2]Available at http://www.eclemma.org/



(a) Detected Components.



(b) False Positives

Fig. 4: Violin plots depicting feature localization accuracy.

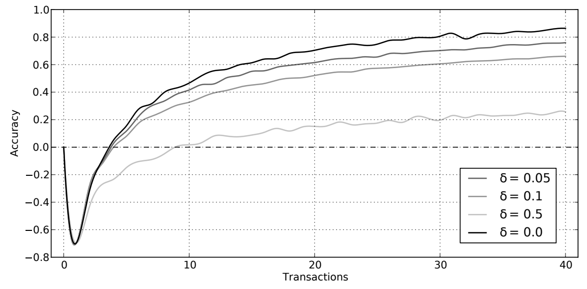


Fig. 5: Impact of misclassification on accuracy.

accuracy. Our measure of accuracy ranges from 1.0—where only components that are part of the implementation have non-zero, maximal association measures— to -1.0—where only dissociated components have maximal, non-zero association measures (please refer to [6] for a formal definition of accuracy).

We studied JHotDraw[3], a customizable Java framework for creating and editing 2D drawings. The feature under consideration for this study was the creation of triangle shapes in the drawing canvas. To do so, we recorded several associated transactions and dissociated transactions. An example associated transaction is to create a triangle shape by clicking the triangle button and click-dragging the mouse in the canvas area. Examples of dissociated transactions include changing an existing shape's color, or creating oval shapes.

We define misclassification (δ) to be the probability that a transaction labeled as dissociated by the user is, in fact, associated with the feature (or vice-versa). The accuracy of several misclassification rates is shown in Figure 5. Both $\delta = 0.05$ and $\delta = 0.10$ show a slight decrease in accuracy, but still comparable to having $\delta = 0.0$. In fact, we are able to achieve similar accuracies if we increase the number of transactions in these misclassification scenarios. This means that if users are unsure if some of the transactions they are collecting are in fact correctly labeled, then they should record more transactions to mitigate this effect.

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

IV. Conclusions

Pangolin is a feature localization plugin for Eclipse that helps developers pinpoint the implementation of a unit of functionality, either by labeling tests as *associated* with a particular feature, or by recording manual interactions with the system. The tool employs a spectrum-based analysis that identifies software components that are likely to be part of a feature implementation based on their involvement on each execution, and displays the result in a sunburst visualization. As future work, we plan to augment the information displayed to users, *e.g.*, by enhancing components with summaries of what they are responsible for, via code summarization techniques based on stereotypes [10], along with searchable, interactive visualization filtering options.

Pangolin can be downloaded at:
https://tqrg.github.io/pangolin/

References

[1] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines*. Springer, 2013.

[2] R. Tiarks, "What programmers really do - An observational study," *Softwaretechnik-Trends*, vol. 31, no. 2, 2011.

[3] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE Transactions on Software Engineering*, vol. 33, no. 6, pp. 420–432, 2007.

[4] B. Dit, M. Revelle, and D. Poshyvanyk, "Integrating information retrieval, execution and link analysis algorithms to improve feature location in software," *Empirical Software Engineering*, vol. 18, no. 2, pp. 277–309, 2013.

[5] A. Perez and R. Abreu, "A diagnosis-based approach to software comprehension," in *In Proc. ICPC'14*, 2014, pp. 37–47.

[6] ——, "Framing program comprehension as fault localization," *Journal of Software: Evolution and Process*, vol. 28, no. 10, pp. 840–862, 2016.

[7] C. Gouveia, J. Campos, and R. Abreu, "Using HTML5 visualizations in software fault localization," in *Proc. VISSOFT'13*, 2013, pp. 1–10.

[8] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: an eclipse plug-in for testing and debugging," in *Proc. ASE'12*, 2012, pp. 378–381.

[9] A. Perez and R. Abreu, "Cues for scent intensification in debugging," in *Proc. IWPD'13*, 2013, pp. 120–125.

[10] N. Alhindawi, N. Dragan, M. L. Collard, and J. I. Maletic, "Improving feature location by enhancing source code with stereotypes," in *Proc. ICSME'13*, Sep. 2013, pp. 300–309.