VISSOFT 2017

Method Execution Reports: Generating Text and Visualization to Describe Program Behavior

Fabian Beck*, Hafiz Ammar Siddiqui†, Alexandre Bergel‡ and Daniel Weiskopf†
*University of Duisburg-Essen, Germany
†University of Stuttgart, Germany
‡University of Chile

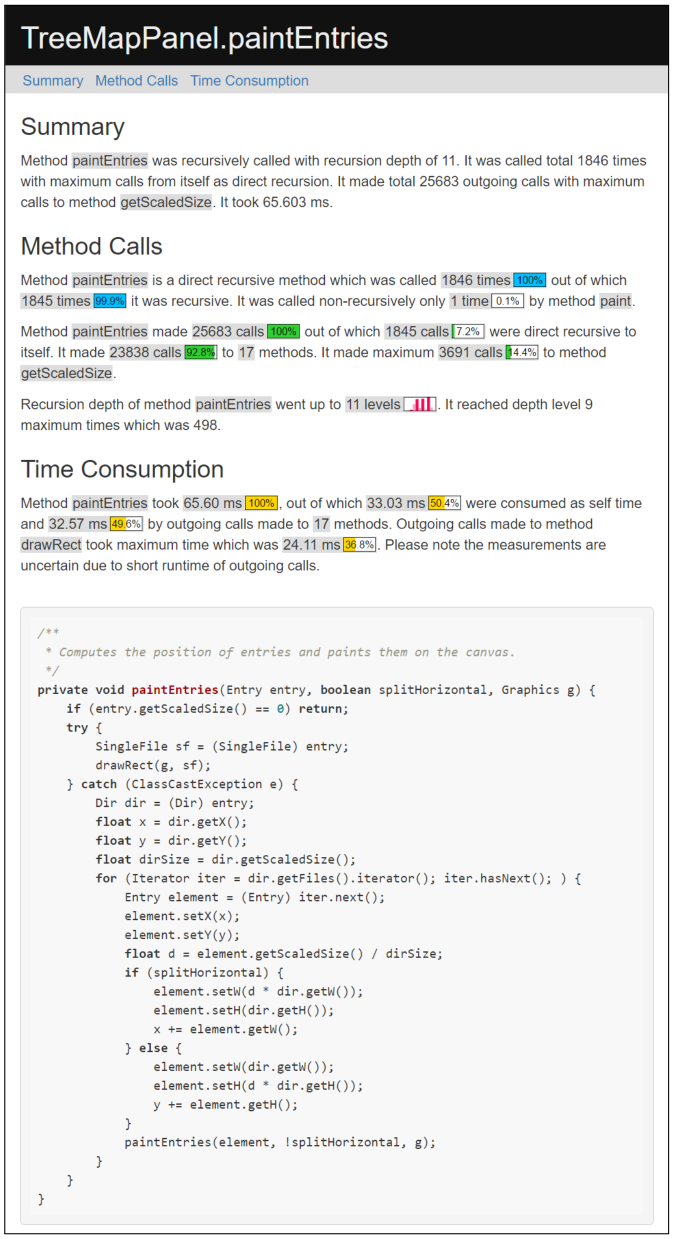
***Abstract*—To obtain an accurate understanding of program behavior, developers use a set of tools and techniques such as logging outputs, debuggers, profilers, and visualizations. These support an in-depth analysis of the program behavior, each approach focusing on a different aspect. What is missing, however, is an approach to get an overview of a program execution. As a first step to fill this gap, this paper presents an approach to generate Method Execution Reports. Each report summarizes the execution of a selected method for a specific execution of the program using natural-language text and embedded visualizations. A report provides an overview of the dynamic calls and time consumption related to the selected method. We present a framework to generate these reports and discuss the specific instantiation and phrasing we have chosen. Our results comprise feedback from developers discussing the understandability and usefulness of our approach and a task-based comparison to state-of-the-art solutions.**

I. Introduction

Programmers frequently need to understand the runtime behavior of an unknown or partly known program. Documentation and expressive identifiers support the programmer developing a basic understanding of the code, but do not provide information about a specific execution. In contrast, logging, breakpoint debugging, profilers, and visualization facilitate the programmer to learn about an execution, but come along with a detailed and often time-consuming analysis.

The *Method Execution Reports* we introduce in this work provide a lightweight alternative that can be generated for a specific program execution. Such reports are generic and describe different aspects of the runtime behavior of the analyzed system. The reports are automatically created from data. In contrast to visualizations, their primary mean of communication is text, not diagrams. Visualizations and interactively explorable details are only added if larger sets of numbers or longer lists of entities need to be communicated. In this way, the ability of visualization to make larger quantities of data readable complements the flexibility and rich vocabulary of textual descriptions.

Figure 1 provides an example of a *Method Execution Report* for `paintEntries`, the main drawing method of a Java program to generate a treemap of a file directory [1] (*i.e.,* a space-filling hierarchy visualization using nested boxes). The report is structured in three sections and lists the code of the method below. In the summary, we learn about basic data: whether the method was called recursively, about the number of incoming and outgoing calls, and how long the summed



TreeMapPanel.paintEntries

Summary Method Calls Time Consumption

Summary

Method paintEntries was recursively called with recursion depth of 11. It was called total 1846 times with maximum calls from itself as direct recursion. It made total 25683 outgoing calls with maximum calls to method getScaledSize. It took 65.603 ms.

Method Calls

Method paintEntries is a direct recursive method which was called 1846 times 100% out of which 1845 times 99.9% it was recursive. It was called non-recursively only 1 time 0.1% by method paint.

Method paintEntries made 25683 calls 100% out of which 1845 calls 7.2% were direct recursive to itself. It made 23838 calls 92.8% to 17 methods. It made maximum 3691 calls 14.4% to method getScaledSize.

Recursion depth of method paintEntries went up to 11 levels. It reached depth level 9 maximum times which was 498.

Time Consumption

Method paintEntries took 65.60 ms 100%, out of which 33.03 ms 50.4% were consumed as self time and 32.57 ms 49.6% by outgoing calls made to 17 methods. Outgoing calls made to method drawRect took maximum time which was 24.11 ms 36.8%. Please note the measurements are uncertain due to short runtime of outgoing calls.

```
/**
 * Computes the position of entries and paints them on the canvas.
 */
private void paintEntries(Entry entry, boolean splitHorizontal, Graphics g) {
    if (entry.getScaledSize() == 0) return;
    try {
        SingleFile sf = (SingleFile) entry;
        drawRect(g, sf);
    } catch (ClassCastException e) {
        Dir dir = (Dir) entry;
        float x = dir.getX();
        float y = dir.getY();
        float dirSize = dir.getScaledSize();
        for (Iterator iter = dir.getFiles().iterator(); iter.hasNext(); ) {
            Entry element = (Entry) iter.next();
            element.setX(x);
            element.setY(y);
            float d = element.getScaledSize() / dirSize;
            if (splitHorizontal) {
                element.setW(d * dir.getW());
                element.setH(dir.getH());
                x += element.getW();
            } else {
                element.setW(dir.getW());
                element.setH(d * dir.getH());
                y += element.getH();
            }
            paintEntries(element, !splitHorizontal, g);
        }
    }
}
```

Fig. 1. Execution report for the method `TreeMapPanel.paintEntries` of a program that draws a treemap visualization for a file hierarchy encoding file sizes in the size of the nested boxes.

executions of the method took. The *Method Calls* section details the description of the call structure, such as listing the most important callers and callees as well as describing the recursion in more detail. The final section provides further information on runtime consumption. Small visualizations, which the reader can explore interactively, augment parts of the statements. Other reports follow the same structure, but their content—depending on the studied method and program execution—varies significantly (*cf.* Figure 1 and Figure 5). Interactive versions of the reports as well as executables and the source code for generating new reports are part of the online supplemental material[1] for this paper.

This paper makes the following main contributions:

- We designed and implemented a technique to generate text based on decision graphs to incorporate interactive and visual indicators.
- We apply the technique to describe method execution behavior with respect to calls and execution time.

First, we report related work and approaches that provide a basis for our approach (Section II). We then describe a general framework to generate execution reports (Section III). As an instantiation of the framework for Java methods, we introduce *Method Execution Reports* (Section IV). We evaluate the resulting reports in a user study and report feedback of software developers (Section V). Moreover, a task-based analysis contrasts our approach to existing solutions (Section VI).

II. Related Work

Our approach is a specific documentation generation technique related to code summarization. Previous techniques use automatically derived keywords and generated phrases to describe a software artifact. A simple summarization approach is to extract meaningful keywords as labels from the comments and identifier names contained in the source code [2], [3]. Other approaches go a step further and create natural-language-text summaries from existing text fragments, for instance, available in bug reports [4]. In contrast, we cannot rely on existing phrases because there usually does not exist a written documentation of specific runs of the system.

However, natural-language text can be also generated without considering existing phrases from the source code structure or software models. Several approaches produce descriptions of object-oriented class models as a basis for documentation [5], [6], [7], [8]. Moreno *et al.* [9] create summarizations of classes from source code. Sridhara *et al.* [10] uses identifiers present in source code statements to produce descriptions of software methods, McBurney and McMillan [11] add context to such descriptions by explicitly considering call dependencies. Further approaches exist to produce descriptions of crosscutting concerns [4], commit messages [12], or release notes [13] from software changes. In contrast to these approaches, we generate a description of dynamic runtime behavior of software. Not only do call dependencies need to be described, but the description is also required to quantify these dependencies and to add general performance information.

In general, *natural language generation* [14] is a field that investigates the automatic production of natural-language texts. In addition to simple templating, called *mail merge*, this field provides sophisticated techniques to compose text from data, constructing grammatically correct sentences. The natural language generation approach presented in this paper has a mid-level of complexity, being more advanced than filling in variables in a text template, but not using advanced grammar-based text generation methods [14]. We further integrate simple visualizations into the generated text. This can be considered a generation of multi-modal documents [15] and shares some similarity with the generation of text that describes a visualization [16]. A tight, interactive integration of text and graphics for generated multi-modal documents like in our approach, however, has not been studied yet.

From a visualization perspective, the focus is on small visualizations integrated into the text known as sparklines [17], word-scale visualizations [18], or word-sized graphics [19]. In software engineering applications, sparklines and similar small graphics have been already integrated into IDE interfaces, especially the code editor, to show software metrics [20], dynamic call and performance information [21], [22], [23], bad smells [24], numeric variable values [25], and feature location search results [26]. While sparklines have become very popular as part of spreadsheets and tables, the integration of sparklines into natural-language text is not as common [19]. With regard to text integration, Goffin *et al.* study placement options for sparklines [27], [28] and the design space for these graphics to enrich a text [18].

Our technique targets an improved understanding of runtime behavior and shares this goal with debugging tools and profilers. Standard interfaces for these tools are tree-based variable exploration and call stack views. Alternative debugging interfaces include systems like *DDD* [29], which visually represents explorable data structures, or *Whyline* [30], [31], which allows developers to ask *why* questions about the program output. In addition to standard profiling tools, performance information can be shown as an overlay of the code or integrated into the code representation [32], [21], [22]. External performance visualization provides an overview of potential bottlenecks [33], [34] and performance regressions [35]. Isaacs *et al.* [36] survey further performance visualization approaches. To the best of our knowledge, a natural language interface for dynamic software analysis data does not exist yet.

III. Execution Reporting Framework

The core of our approach is a technique to generate natural-language texts. We add visual augmentations to the text to provide details and allow users to further explore the data. Finally, we integrate all parts into a system that records the dynamic information during program execution and builds the interactive reports as Web pages. This section introduces all

[1]https://fabian-beck.github.io/Method-Execution-Reports/

building blocks of the approach as a technical framework and basis for later producing specific reports.

A. *Text Generation*

We use decision graphs and parameterized sentence templates to generate the natural-language text. Decision graphs are inspired by well-known flowcharts [37], which consist of decision and processing vertices. In terms of natural language generation [14], this templating approach can be considered as an advanced *mail merge* technique. In contrast to more complex generation techniques like those discussed by Reiter *et al.* [14], it does not require building a language and grammar model. For our application, this mid-level of complexity provides enough flexibility while, at the same time, it remains easy to handle.

A report consists of different sections, each contributing one or several paragraphs. The sequence of the paragraph is fixed. Each decision graph describes a procedure to generate the content of a paragraph. The result might be empty if certain conditions hold (*e.g.,* if a method is non-recursive, the paragraph describing recursion stays empty). The graph is a directed acyclic graph that consists of the following vertices and edges:

- **Start Vertex** (1×)—The unique entry point of the graph; a single outgoing edge connects a start vertex with the next vertex.
- **Text Vertex**—A vertex that, when visited, adds a parameterized text fragment to the paragraph; a single edge connects it with the next vertex.
- **Decision Vertex**—A conditional vertex with several outgoing edges, each representing a specific case; the edge labels indicate the different discussions partitioning all possible cases (*i.e.,* comparable to a `switch` statement).
- **Stop Vertex** (1×)—The unique exit point of the graph; the vertex does not have any outgoing edges.

Visiting the graph from the start to the stop vertex, the text vertices produce a linear sequence of text. The decision vertices and their contained conditions determine the path through the graph. The path is deterministic because there is always a unique branch selected in each decision. A path always ends in a stop vertex because the graph is acyclic and the stop vertex is the only vertex without outgoing edges. The text template used in a text vertex usually range from a few words to up to a full sentence. The parameters of the templates are numbers (*e.g.,* call frequencies or timing information) and identifier names (*e.g.,* method names of called or calling methods). For changing quantities, some templates require slight modifications of the grammar (*e.g.,* switching between singular and plural) or wording (*e.g.,* adding *"only"*). Beside this, the template is static—we reflect different cases that require larger modification through different text vertices in the decision graph.

Figure 2 provides the decision graph used to produce the *Summary* section of the report. The diagram shows decision vertices marked with rounded corners and text vertices highlighted with colors. Note that it is not feasible due to limited

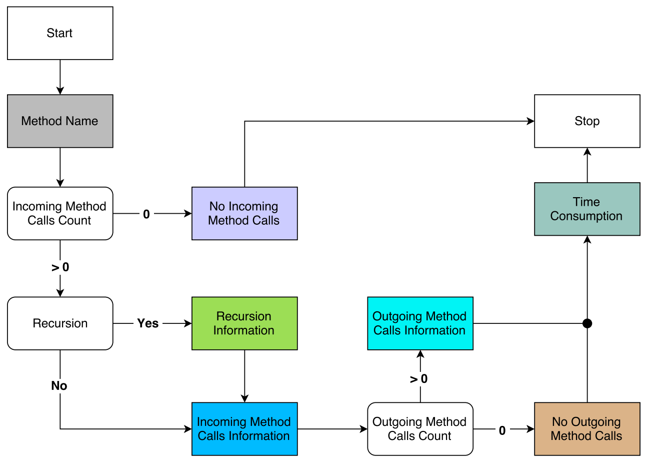


Fig. 2. Decision graph describing the composition of the report summary section. Nodes with rounded corner represent decision vertices. Colored nodes represent text vertices (we use different colors to discern and visually reference the nodes in Sections III and IV).

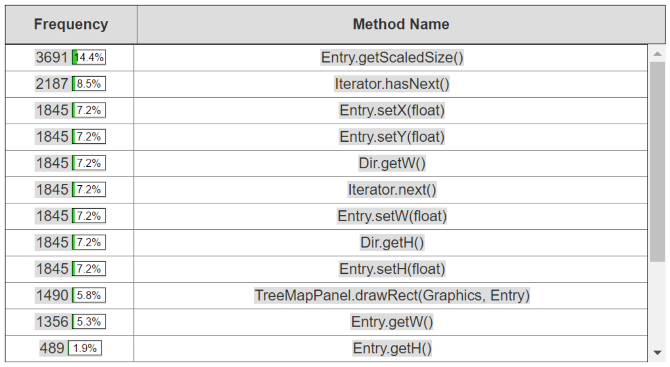
space to include the exact formulation of conditions and text templates in the diagram, but we use descriptive identifiers instead and provide all text templates as part of the online supplemental material. For instance, the text template connected with the *Method Name* vertex ■ is *"Method <method>"*, where *"<method>"* is a placeholder for the shortened method name. A more complex template is assigned to the *Recursion Information* vertex 🟩:

- If recursion depth = 1: *"was recursively called with recursion depth of only 1. It"*
- else: *"was recursively called with recursion depth of <1+>. It"*

In the recursive case, this phrase follows the text produced by the *Method Name* vertex ■. Hence, the two parts are concatenated, for instance, to produce *"Method paintEntries was recursively called with recursion depth of 11. It"* (*cf.* Figure 1). Please note that the examples ends with *"It"*, which already bridges to the next sentence. This solution is appropriate here because, in the non-recursive case, the *Recursion Information* vertex 🟩 is skipped and the next vertex would extend *"Method <method>"*. The specific path in the example of Figure 1 followed through the decision graph (Figure 2) is: *Start* → *Method Name* ■ → *Recursion Information* 🟩 → *Incoming Method Calls Information* 🟦 → *Outgoing Method Calls Information* 🟦 → *Time Consumption* ■ → *Stop*.

B. *Visual Augmentation*

Together with the text, we produce and integrate visual augmentation and interactive details that the user can retrieve on demand. The goal of these augmentations is to provide additional information that, if expressed verbally, would overload the text. We leverage the interactivity of Web documents that allow blending in more information when hovering a text phrase with the mouse. Moreover, we integrate word-



Frequency	Method Name
3691 14.4%	Entry.getScaledSize()
2187 8.5%	Iterator.hasNext()
1845 7.2%	Entry.setX(float)
1845 7.2%	Entry.setY(float)
1845 7.2%	Dir.getW()
1845 7.2%	Iterator.next()
1845 7.2%	Entry.setW(float)
1845 7.2%	Dir.getH()
1845 7.2%	Entry.setH(float)
1490 5.8%	TreeMapPanel.drawRect(Graphics, Entry)
1356 5.3%	Entry.getW()
489 1.9%	Entry.getH()

Fig. 3. Detailed list of 17 methods called by `paintEntries` as available on demand in the report shown in Figure 1.

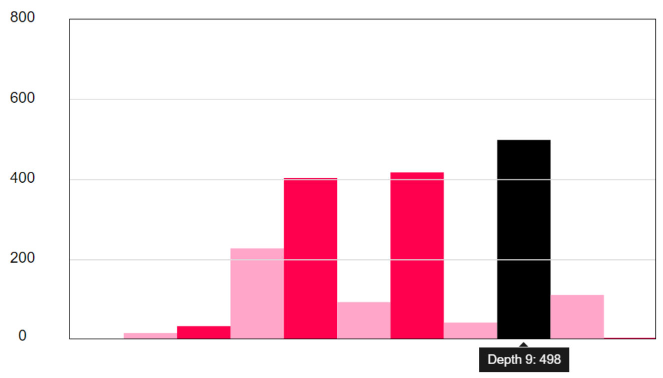
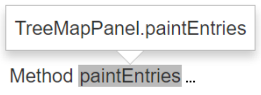


Fig. 4. Enlarged interactive histogram representing recursion depths of `paintEntries` as available on demand in the report shown in Figure 1.

sized graphical representation to illustrate certain quantities. In particular, our reports include the following augmentations:

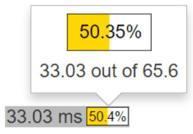
- **Shortened Names**—We refer to other methods using the method name, highlighted in gray to discern the identifier from plain text (*cf.* Figure 1). A longer version of the name also listing the class of the method is available on demand in a tooltip dialog.

  TreeMapPanel.paintEntries

  Method paintEntries ...

  If the identifier name alone is not unique within the report (*e.g.,* several overloaded methods with the same name), we add further details (class name, parameter types, return type, etc.) until the name becomes unique.
- **Fill Bars**—For visualizing relative quantities (*e.g.,* "x out of y"), we add small bars that represent the quantity as a fill level and provide a percentage value x/y (*cf.* Figure 1). We highlight the text the graphic refers to in gray and again provide a tooltip dialog on demand.

  50.35%

  33.03 out of 65.6

  33.03 ms 50.4%

  In the reports, we use different colors to discern different reference quantities y for varying number x. For introducing a new reference quantity y, we first add a 100% bar (y/y) in the respective color.
- **Quantified Method Lists**—When we need to list more than two methods assigned with relative quantities (*e.g.,* all callers with respective call frequencies), we only summarize the total number of methods in the textual report. On click, we provide further details in a pop-up dialog as shown in Figure 3 (and a preview on mouse hover). It contains a table providing the relative quantity as a fill bar and a shortened but unique method name.
- **Histograms**—For representing distributions of quantities, we further provide small bar chart histograms integrated as sparklines into the text. Clicking one of these opens an enlarged version of diagram in a pop-up dialog (Figure 4). Currently, we use this visualization only for summarizing depth levels of recursions, but the diagram easily generalizes to other kinds of histograms or timelines.

The visual augmentation is designed to be visually simple, intuitive, and easy to understand. We did not want to overload the representation and keep the text readable. Interactions are limited to retrieving details on demand, which is a common interaction pattern that users are familiar with.

C. Implementation

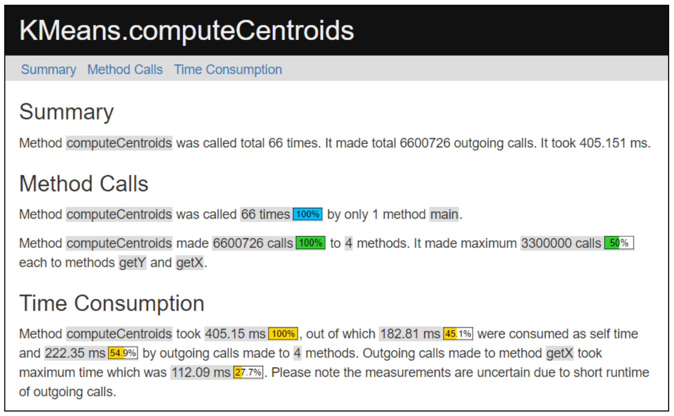
Our implementation integrates text generation and visual augmentation into a processing pipeline for generating reports of Java methods. As a first step, we need to profile a Java program at runtime to record all required information. We focus the recording on a single method only and summarize all executions of this method within a single run. The second step is generating an interactive report from this information.

Profiling: Profiling describes the process of observing and recording program behavior. Since we are interested in specific information on the execution behavior of a single method, we do not need to profile the whole system. We use a simple instrumentation approach based on *Javaassist* and instrument the bytecode at load-time. We implemented a *Java Agent* to instrument an executable *Java* program. As input, the agent requires the name of the method of interest for which the report should be generated. Our agent records the following events including timestamps and involved other methods:

- **Method Start/End**—The execution of the method of interest starts/ends.
- **Incoming Call**—The method of interest is called by another method.
- **Outgoing Call**—The method of interest calls another method.

We store these events together with timestamps in an XML file. This allows us to later calculate method execution times for the method itself and callees of the method. For recursive methods (both direct recursion as well as indirect recursion through other methods), we take care to not count runtime several times when aggregating the runtime information on method level [22]. Depending on how often the method of interest is executed, the runtime overhead produced by profiling varies. Currently, both call structure and timing information are recorded in a single run. To make the measurement more reliable, two separate runs might be used in the future. We chose a simple profiling approach, but the profiling can be easily replaced by a more sophisticated solution without affecting the general approach. We currently do not record which threads executed the method; reflecting this information in the generated reports would be a promising extension.

Generation of Interactive Report: We use Web technologies for the reports—HTML and CSS provide a simple basis to structure and lay out the text, and JavaScripts adds interaction and visual indicators. In particular, we use D3js to produce interactive visualizations. The generator of the reports itself is written in Java. It takes an XML file (and the source code of the method of interest) as input and produces a static HTML file linked with all required libraries and style files. Hence, users can run this process locally and do not need to set up a Web server to generate and view the reports. Currently, the decision graphs and templates are manually transformed to Java code. Future extensions of the implementation might provide a user interface to visually specifying the graphs and templates and generate the code automatically from the graphical representation. Also, a closer integration of the generation process and the presentation of the resulting report into an IDE is a desirable extension.



KMeans.computeCentroids

Summary Method Calls Time Consumption

Summary

Method computeCentroids was called total 66 times. It made total 6600726 outgoing calls. It took 405.151 ms.

Method Calls

Method computeCentroids was called 66 times 100% by only 1 method main.

Method computeCentroids made 6600726 calls 100% to 4 methods. It made maximum 3300000 calls 50% each to methods getY and getX.

Time Consumption

Method computeCentroids took 405.15 ms 100%, out of which 182.81 ms 45.1% were consumed as self time and 222.35 ms 54.9% by outgoing calls made to 4 methods. Outgoing calls made to method getX took maximum time which was 112.09 ms 27.7%. Please note the measurements are uncertain due to short runtime of outgoing calls.

Fig. 5. Execution report for the method `KMeans.computeCentroids`, which computes the new centroid positions in each iteration of a k-means clustering algorithm for clustering 50,000 2D points into $k = 10$ clusters.

IV. Method Execution Reports

To produce reports, we instantiated our execution reporting framework with specific decision graphs and text templates. We started off composing the report with a description of the call structure of a method because (i) method calls form the basic communication between different components of a system, (ii) calls—in particular incoming ones—are hard to retrieve from the source code, and (iii) the dynamic call structure might significantly vary from the static calls retrieved with a static analysis tool. We iterated the templates and examples internally before we asked software developers for preliminary feedback. Based on this, we improved the report and extended it by a description of the runtime consumption. The final report consists of three sections: a short summary, a description of the method calls, and a description of time consumption. In the following, we explain the content of the three paragraphs and discuss the scenarios that need to be differentiated. While the decision graph for the first paragraph is shown in Figure 2, the online supplemental material provides details on all decision graphs and text templates used.

A. Report Summary

The initial paragraph—like in a regular text—is intended to give an introduction and summarize the most important information. The complexity of the summary varies with the complexity of the method's execution to some extent. As illustrated in the decision graph in Figure 2, the shortest path to the *Stop* vertex is taken if there are no incoming calls—the method is not executed. The respective text vertex just describes that the method *"was never called"* ⬜ and all following sections are skipped. However, the standard case is that the method is called at least once and also calls other methods itself. If no recursion is involved, a typical result like the one in Figure 5 consists of three sentences reporting the incoming calls 🟦, the outgoing calls 🟦, and the runtime ⬜. In case of recursive calls (*cf.* Figure 1), another sentence is added describing the recursion 🟩; we inserted this information at the beginning of the summary because recursion is one of the most important characteristics of a method. For both incoming 🟦 and outgoing 🟦 calls, additional information on the source or target of the calls is added if it can be phrased in a simple way. Whereas the example in Figure 5 does not contain such additions, the example in Figure 1 provides more information on both incoming (*"[...] with maximum calls from itself as direct recursion"*) and outgoing calls (*"[...] with maximum calls to method getScaledSize"*).

B. Method Calls

The next report section details the method calls. It consists of three paragraphs: one for incoming calls, one for outgoing calls, and one for recursion. These are only generated if the respective information exists, that is, if there are incoming or outgoing calls or if the method is called recursively. For instance, the example in Figure 1 contains all three paragraphs, whereas the one in Figure 5 skips the paragraph on recursion.

For describing the incoming calls in the first paragraph, again, it makes a difference whether the method was called recursively or not. For recursive calls, direct recursion (*i.e.,* the method calling itself) is discerned from indirect recursion (*i.e.,* the method calling itself through a chain of other methods). The text describes the number of non-recursive, directly recursive, and indirectly recursive calls with blue fill bars to get a quick overview of the numbers. For the non-recursive calls, the paragraph summarizes the callers: If the calls came from only one or two methods, then the callers are named. If there are more callers, the text just names the method which called most frequently (*e.g., "It was called maximum <1+> times by method <method>."*) and considers special cases (*e.g., "It was called only 1 time each by all methods."*). The full list of callers is always available when clicking on the number of non-recursive callers. For *public* methods, we further provide information whether they were called from outside the class they are contained in or not (if not, this might be indicator to change their visibility to *private*). A further special case is the main method that does not have regular incoming calls—we report that it is only invoked once by the virtual machine.

The next paragraph provides equivalent information on the callees of the method of interest. Again, it discerns recursive and non-recursive calls and informs about the callees—either listing their names, or just naming the top ones while all callees are available as a list on demand (*cf.* Figure 3). To discern the call frequencies of callees from those of callers, we use green fill bars instead of blue ones in this paragraph.

The final paragraph of the section describes the recursion in detail if applicable (*cf.* Figure 1). Recursive calls form a tree. The text summarizes the properties of this tree, in particular, its deepest level of nesting and the level that is reached most frequently. An interactively explorable histogram provides more detail on the depth distribution (*cf.* Figure 4). With this information, a programmer can check whether the recursion works as expected. For instance, when drawing a treemap of a file structure like in method `paintEntries` (Figure 1), these statistics directly reflect the nesting structure of the directory that shall be visualized.

C. *Time Consumption*

The last section reports details on the runtime consumption of the method aggregated across all calls. As an important distinction, the text discerns between self-time (i.e., the time consumed by the instructions of the method itself) and runtime consumption that stems from called methods (i.e., waiting for outgoing calls to return). Yellow fill bars visualize these quantities. Similar to the callee with the maximum call frequency, we list here the callee that consumes the maximum runtime. Further details can be explored in a list of all callees comparable to the example in Figure 3. Since measuring performance underlies different measurement uncertainty, if necessary, we append a warning at the end of the paragraph (*e.g., "Please note the measurements are uncertain due to short runtime of outgoing calls."*). To decide what warning needs to be added, we developed a heuristic to estimate uncertainty of measurement. Factors that increase the uncertainty are short overall runtime and short runtime of outgoing calls. When the calls are extremely short and for indirect recursion (which is difficult to describe with respect to runtime consumption of a method), the whole paragraph is shortened and just provides the total runtime.

V. Developer Feedback

The evaluation is focused on practitioner perception. As such, we directly evaluate the result of our approach, *i.e.,* generated reports produced for a set of software systems. We therefore do not directly evaluate the decisions we have taken to produce those reports, such as the decision graph. Two aspects have to be considered:

- The work presented in this paper is the first iteration of our overall effort of bridging the gap between profiling and debugging tools with end users. This mean that we are interested, at that current stage, in (i) identifying potential use cases of our approach and (ii) obtaining feedback for future iterations.
- As far as we are aware of, the *Method Execution Report* technique is unique regarding both the information it provides and the way information is presented. Although some profiling tools are able to produce advanced detailed reports of a runtime behavior, none is able to produce a comprehensive report in the same spirit as our reports. As such, there is no *fair* baseline to which we can directly compare our approach to. For this reason, we exclude carrying out a comparative empirical evaluation, but compare the technique to others in a task-based analysis (Section VI).

The research question we seek to answer is *How Method Execution Reports are perceived by practitioners?* In such a case, surveying a group of practitioners about their perceptions for a given set of reports is therefore the strategy we employ.

A. *Survey Description*

We designed a survey composed of three parts. The first part is about the personal experience and background of the participant. The questions we ask cover (i) the programming languages and environment the participant is familiar with, and (ii) the known debugging and profiling techniques. The second part briefly describes the *Method Execution Report* technique in the same fashion as Section IV. This part is necessary to ensure that all the participants receive the same information.

The third part presents a method execution report to the participant and poses a set of questions about (Q1) the textual content of the report, (Q2) the interaction and visual elements offered by the report, (Q3) usefulness of the report, (Q4) software engineering tasks for which the report may be useful to have, (Q5) alternatives to the report, and (Q6) any other criticism. We use reports produced for six different methods, each belonging to a particular application. The method size ranges from 12 to 111 lines of code (comments are not accounted). The method `paintEntries`, illustrated in Figure 1, is among the method execution reports we employed in this experiment. Answers of the third part are reported in plain English. The survey and all participant answers are available as part of the online supplemental material.

B. *Experiment Execution*

During the execution of our experiment, we considered two aspects: (i) participants results were collected on paper, and (ii) method reports were presented on screen. Participants did not have access to other sources of information, such as a programming environment or the complete code base. Since we aim to obtain feedback on our report while minimizing possible bias, participants were restricted to only explore the reports we provided.

C. Participants

In total, eleven participants took part in our survey: three participants are PhD students in software engineering at the University of Chile and the eight other participants are from three different software development companies based in Chile. The programming experience of the participants ranges from 3 to 14 years, with 7.4 years as average, a median of 7 years, and a standard deviation of 3.3 years. Participants are familiar with Java, and in particular the Eclipse, IntelliJ, and Netbeans programming environments. Eight participants are familiar with one or more dynamically typed languages, including Python, JavaScript, and Pharo. Participants took 19 minutes (both average and median) to fill the survey. The fastest participant took 9 minutes and the slowest took 31 minutes. We employed reports generated from seven different Java software systems. Systems to be analyzed were evenly distributed among the participants.

D. Results

Overall, the participants positively perceived the *Method Execution Reports*. From the eleven participants, nine participants rated the sentence *"Overall, do you feel that such a report is useful?"* (Q3) with *strongly agree* or *agree* (on a five-point scale from *strongly disagree* to *strongly agree*). During the experiment, participant were free to ask any questions about the survey and the report. No participants asked questions. The remaining of this section summarizes participants' answers.

Q1: Textual content of the report—Participants were positive about the textual content of the report. For example, Participants P2 and P4 mentioned: *"The language used is clear"*, *"The report is very useful to know where methods spend time"*, *"the report provides data without redundant text"*, *"[the report is] useful to know if a loop is working correctly"*. On the negative side, two participants raised some English grammar issues.

Q2: Interaction and visual elements—Participants were slightly less enthusiastic about the interaction and the visual elements embedded in the textual content. Giving a meaning to the color was perceived as difficult and not intuitive (*e.g.,* P1, P2, P4). P9 said *"it took me a little bit to understand the percentage bars. However, once understood, they are very useful"*. Participants P3 and P6 are very positive. When referring to the interaction that appears on demand, P6 says: *"We are used to hyperlinks in the web"*.

Q3: Usefulness of the report—Nearly all the participants were very positive on the usefulness of the report. For example, P2 said *"The report is useful because it helps understand what the code is doing and where it spends most of the execution time"*, and P1 said *"I think it would be more useful having the information in the source code [instead of having a webpage]"*.

Q4: Software engineering tasks for which the report may be useful to have—Most participants (*e.g.,* P3, P4, P8, P9) indicate that the reports are useful to find performance problems and identifying bottlenecks. In addition, we obtained the following answers:

- *"The report is also good for people communicating performance problem with other developers"* (P2)
- *"[the report is useful] when a developer needs to make a refactoring or an optimization of the code"* (P4)
- *"useful to assess whether a loop is expensive"* (P6)
- *"detect bottlenecks when long running simulations"* (P7)
- *"useful when developing web applications (with some very short time response)"* (P11)

Q5: Alternative to the report—No participant knew of a code monitoring tool that provides the same information as we do. Nevertheless, some tools and techniques were mentioned as a way to obtain part of the information we report: *"debugging framework"* (P3), *"Java Mission Control"* (P4), *"gprof in C and C++"* (P5), *"Chrome profiler"* (P10).

Q6: General criticism about the report—As a positive criticism, P4 said *"Minimalist design, does not include unnecessary information"*. In contrast, many participants complained that the report is focused on one single method. More than half of the participants felt unnecessarily restricted to one single method. For example, *"[the report should provide] the ability for going deeper into the call tree"* (P6), *"having more visual elements to help understanding the code flow (*e.g., *using a graph or tree)"* (P7). Two participants complained about the unnecessary separation between our report and the programming environment (P1) and the source code (P11).

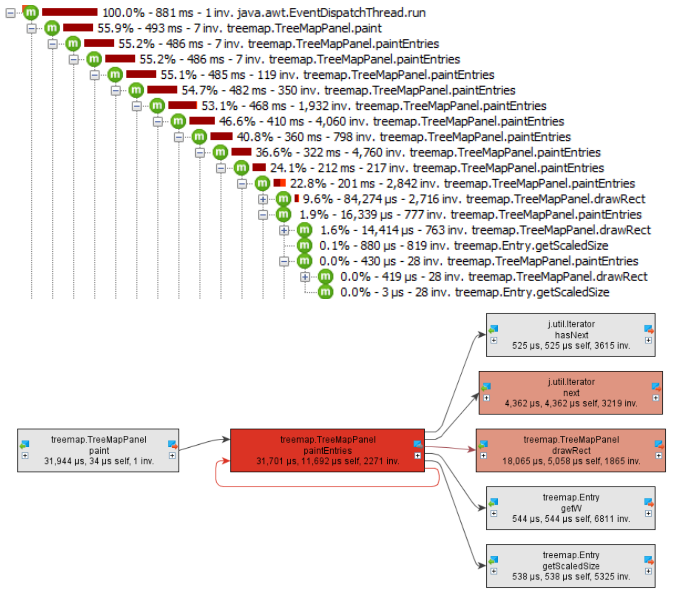
VI. Task-based Comparison

Finally, we compare our approach to state-of-the-art tools. However, it is difficult to decide which are the direct competitors to compare with. We do not argue that *Method Execution Reports* should replace any of the established debugging and profiling tools, but that it fills a gap for understanding the execution of a source code fragment. Hence, we have to contrast our approach to a variety of types of tools. We show that the main tasks our approach supports are not sufficiently covered by the state of the art. Note that we do not list tasks that our approach does not support but might be facilitated by other tools.

A. Selected Approaches

Runtime information can be collected with a variety of approaches from simple manual instrumentation (logging) to sophisticated profiling tools. In particular, we discern the following techniques:

- **Breakpoint Debugger**—Setting breakpoints, stepping through them, and interactively exploring program state like supported by the Eclipse IDE.
- **Logging**—Recording logging outputs using plain system outputs or a logging framework such as Log4j.
- **Profiling Call Tree**—Inspecting an interactively expandable call tree of methods like provided by profiling tools



```
100.0% - 881 ms - 1 inv. java.awt.EventDispatchThread.run
  55.9% - 493 ms - 7 inv. treemap.TreeMapPanel.paint
    55.2% - 486 ms - 7 inv. treemap.TreeMapPanel.paintEntries
      55.2% - 486 ms - 7 inv. treemap.TreeMapPanel.paintEntries
        55.1% - 485 ms - 119 inv. treemap.TreeMapPanel.paintEntries
          54.7% - 482 ms - 350 inv. treemap.TreeMapPanel.paintEntries
            53.1% - 468 ms - 1,932 inv. treemap.TreeMapPanel.paintEntries
              46.6% - 410 ms - 4,060 inv. treemap.TreeMapPanel.paintEntries
                40.8% - 360 ms - 798 inv. treemap.TreeMapPanel.paintEntries
                  36.6% - 322 ms - 4,760 inv. treemap.TreeMapPanel.paintEntries
                    24.1% - 212 ms - 217 inv. treemap.TreeMapPanel.paintEntries
                      22.8% - 201 ms - 2,842 inv. treemap.TreeMapPanel.paintEntries
                        9.6% - 84,274 µs - 2,716 inv. treemap.TreeMapPanel.drawRect
                        1.9% - 16,339 µs - 777 inv. treemap.TreeMapPanel.paintEntries
                          1.6% - 14,414 µs - 763 inv. treemap.TreeMapPanel.drawRect
                          0.1% - 880 µs - 819 inv. treemap.Entry.getScaledSize
                          0.0% - 430 µs - 28 inv. treemap.TreeMapPanel.paintEntries
                            0.0% - 419 µs - 28 inv. treemap.TreeMapPanel.drawRect
                            0.0% - 3 µs - 28 inv. treemap.Entry.getScaledSize
```

treemap.TreeMapPanel paint 31,944 µs, 34 µs self, 1 inv.
treemap.TreeMapPanel paintEntries 31,701 µs, 11,692 µs self, 2271 inv.
j.util.Iterator hasNext 525 µs, 525 µs self, 3615 inv.
j.util.Iterator next 4,362 µs, 4,362 µs self, 3219 inv.
treemap.TreeMapPanel drawRect 18,065 µs, 5,058 µs self, 1865 inv.
treemap.Entry getW 544 µs, 544 µs self, 6811 inv.
treemap.Entry getScaledSize 538 µs, 538 µs self, 5325 inv.

Fig. 6. Expanded JProfiler call tree (top) and call graph (bottom) for Java method `TreeMapPanel.paintEntries` in a similar execution to the one described in the report in Figure 1.

such as JProfiler; methods are represented multiple times if executed in different parts of the tree (Figure 6, top).

- **Profiling Call Graph**—Inspecting an interactively expandable call graph provided by profiling tools such as JProfiler (calls of the same method are aggregated into a single node); the nodes and edges are annotated with call frequency and runtime information (Figure 6, bottom).

This list is not complete but should cover the most widely used tools to retrieve runtime information on method level. To review a group of tools, we keep the following discussion on a generic level. Still, to make the comparison more specific, the named examples of tools act as representatives for each a group of tools in the comparison.

B. Comparison Results

We manually played through three tasks regarding the dynamic call structure of the methods (T1–T3) and two tasks concerned with runtime consumption (T4–T5). Table I lists the results, describing how each of the tasks can be addressed with each of the approaches. We summarize the level of support on a qualitative scale ranging from *supported* ++ to *not supported* −−. In addition, we explain the steps that need to be taken for every rating. Note that, although we tried to be as objective and fair as possible, our ratings and explanations might still be influenced by our subjective opinion. The analysis was conducted by a single author.

Regarding the tasks related to method calls (T1–T3), it is hard to retrieve the required information through breakpoint debugging and logging, for non-trivial cases even infeasible in practical application. Incoming calls are easier to observe with these approaches than outgoing one—one sets a breakpoint or adds a logging instruction at the beginning of the method. In contrast, outgoing calls are harder to trace because the method would need to be paused or instrumented at each instruction calling another method, which results in an unrealistic effort. Profiling call trees and graphs better support tasks T1–T3. In call trees, however, the different incoming calls produce separate nodes—if there are multiple incoming calls, information needs to be assembled across the tree (*cf.* Figure 6, top). In contrast, all calls of the same method are already aggregated in a node of a call graph and tasks T1 and T2 are easy to answer (*cf.* Figure 6, bottom). Note that profilers such as JProfiler also allow for aggregating nodes in the call tree for selected nodes—we consider this to be a variant of a call graph representation. Information on recursion (T3) can be better explored in call trees, but the levels need to be expanded manually; in the call graph, recursion is aggregated to self-edges and cycles, but cannot be explored in detail.

For profiling a program (T4–T5), breakpoint debugging and logging are not suitable because breakpoint debugging invalidates the measurement by pausing the program and manual logging instrumentation for profiling are only feasible on a coarse granularity (*e.g.*, measuring the total runtime of a benchmark). Of course, the representations of profilers are much better suited for these tasks. Again, the problem of a call tree is that we are interested in aggregated information for all calls of a method, which needs to be manually collected in the tree. The call graph comes much closer to our textual representation because it is based on this aggregation. Like shown in Figure 6 (bottom), time information can be quickly read from the color coding applied to the node (T4) and the edges (T5).

C. Discussion

In general, the comparison shows that call graph representations are most similar to our textual reports—they aggregate calls for the same method and show call frequencies as well as time consumption. The main difference between these representations is that we currently focus on a single method only, but the graph representation provides an overview of a wider call context. However, in the call graph, information on the characteristics of the recursion is hard to acquire. Also, it would be more difficult to extend graphical representation with further information as suggested by developers in the user study. Also, graph representations run into scalability issues when representing larger quantities of incoming or outgoing calls. JProfiler therefore limits the number of initially shown adjacent nodes (*cf.* Figure 6). In contrast, our reports easily scale to larger numbers of calls because we aggregate them into lists that are only shown on demand. We believe that the two representations complement each other and consider integrating a call graph representation to our reports. Interactively connecting it with the information described in the text could improve understandability further.

For now, the presented reports are limited to describing methods individually. This is a useful starting point, but as also indicated by developer feedback, for a versatile practical application, it would be helpful to create such reports at least

TABLE I

HOW TASKS CAN BE ANSWERED WITH METHOD EXECUTION REPORTS IN COMPARISON TO EXISTING SOFTWARE DEVELOPMENT TOOLS. LEGEND – ++ SUPPORTED; + SUPPORTED, BUT COMPLICATED AND TIME-CONSUMING; − ONLY PARTIALLY SUPPORTED; −− NOT SUPPORTED.

Task	**Execution Report**	**Breakpoint Debugger**	**Logging**	**Profiling Call Tree**	**Profiling Call Graph**
Method Calls					
T1 *Which methods called the method of interest (i.e., callers) and what was the distribution of calls?*	++ Read the first paragraph of the *Method Calls* section and explore the list of callers.	+ Set a breakpoint at the method definition and iterate through all calls.	+ Set up a map of counters (caller $\rightarrow$ integer) and increase at method entrance the respective counter.	+ Find all instances of the method in the tree and retrieve the callers as parent nodes in the tree.	++ Follow incoming edges of the method node and compare quantities of weighted edges.
T2 *Which other methods were called by the method of interest (i.e., callees) and what was the distribution of calls?*	++ Read the second paragraph of the *Method Calls* section and explore the list of callees.	− Step through the instructions and observe the outgoing calls; usually, too much effort to do for all calls.	− Instrument instructions containing outgoing calls; usually, too much effort to do for all calls.	+ Find all instances of the method in the tree and retrieve the callees as child nodes in the tree.	+ Follow outgoing edges of the method node; call frequencies are not visualized and only available on demand.
T3 *Was the method called recursively and what were the characteristics of the recursion (direct/indirect recursion, depth levels)?*	++ Read the *Summary* and *Method Calls* section, in particular, the third paragraph of the *Method Calls* section.	− Set breakpoints at a direct recursive calls; recursion statistics and information on indirect recursion not to acquire with reasonable effort.	− Direct: insert a logging at all direct recursive calls to see whether they are executed; indirect: not supported with reasonable effort.	− Find all instances of the method in the tree and manually expand the tree; recursion statistics and indirect recursion infeasible to investigate.	− Find self-edges for direct recursion and circles for indirect recursion; depth levels cannot be retrieved.
Time Consumption					
T4 *What is the aggregated method call time and self time?*	++ Read the *Runtime Consumption* section.	−− Pausing the program interferes with time measurement.	− Record start and end time of a call to measure aggregated call time; self time not feasible to measure.	+ Find all instances of the method in the tree and investigate the incoming runtime consumption.	++ Read the color and the annotation of the method node.
T5 *How does the time consumption propagate through outgoing calls to other methods?*	++ Read the *Runtime Consumption* section and explore the list of callees with annotated runtime information.	−− Pausing the program interferes with time measurement.	−− Usually not feasible because all callees would need to be manually instrumented.	+ Find all instances of the method in the tree and investigate the outgoing runtime consumption.	++ Follow outgoing edges of the method and read time consumption from the color of the edges.

for sets of methods and to connect the reports by hyperlinks. Future research could also explore whether it is possible and useful to create comparable reports on higher level of abstraction, such as class or package level. The ultimate challenge would be to create a meaningful report for a full execution or even sets of executions. Together with broadening the scope of the reports, a better integration of the reports with IDEs and profilers becomes indispensable. In the IDE, a developer could execute a program in a special profiling mode (comparable to the debug mode) and then study the generated reports for a selected set of methods. In a profiler, the report could act as a detail view for a method that appears when selecting a method in a call tree or graph.

VII. CONCLUSION

We presented *Method Execution Reports*, which provide a quick summary of the behavior of a method within a specific execution of the program. Feedback from developers provides evidence on the value and use cases of the approach as well as suggests specific extensions. Our task-based comparison shows which questions about program behavior the reports answer already. *Method Execution Reports* complement debugging and logging approaches and might act as an entry point of a detailed analysis using call tree and graph representations of profilers. We consider our work as a proof of concept to demonstrate the potential of textual description of program behavior augmented by visualizations and hope it will trigger further research and tooling leveraging such approaches.

ACKNOWLEDGMENT

We are deeply grateful to the development team of INRIA Chile and Object Profile for having participated in our experiment. Also, Fabian Beck is indebted to the Baden-Württemberg Stiftung for the financial support of this research project within the Postdoctoral Fellowship for Leading Early Career Researchers.

REFERENCES

[1] J. J. van Wijk and H. van de Wetering, "Cushion Treemaps: Visualization of hierarchical information," in *Proceedings of the 1999 IEEE Symposium on Information Visualization*, ser. InfoVis. IEEE, 1999, pp. 73–78.

[2] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proceedings of the 32nd International Conference on Software Engineering*, ser. ICSE, vol. 2. IEEE, 2010, pp. 223–226.

[3] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Using IR methods for labeling source code artifacts: Is it worthwhile?" in *Proceedings of the 20th International Conference on Program Comprehension*, ser. ICPC. IEEE, 2012, pp. 193–202.

[4] S. Rastkar, G. C. Murphy, and G. Murray, "Automatic summarization of bug reports," *IEEE Transactions on Software Engineering*, vol. 40, no. 4, pp. 366–380, 2014.

[5] B. Lavoie, O. Rambow, and E. Reiter, "The ModelExplainer," in *Demonstration Notes of the International Natural Language Generation Workshop*, ser. INLG, 1996, pp. 9–12.

[6] ——, "Customizable descriptions of object-oriented models," in *Proceedings of the Fifth Conference on Applied Natural Language Processing*, ser. ANLC. Association for Computational Linguistics, 1997, pp. 253–256.

[7] F. Meziane, N. Athanasakis, and S. Ananiadou, "Generating natural language specifications from UML class diagrams," *Requirements Engineering*, vol. 13, no. 1, pp. 1–18, 2008.

[8] H. Burden and R. Heldal, "Natural language generation from class diagrams," in *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*, ser. MoDeVVa. ACM, 2011, p. 8.

[9] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for Java classes," in *Proceedings of the IEEE 21st International Conference on Program Comprehension*, ser. ICPC. IEEE, 2013, pp. 23–32.

[10] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE. ACM, 2010, pp. 43–52.

[11] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in *Proceedings of the 22nd International Conference on Program Comprehension*, ser. ICPC. ACM, 2014, pp. 279–290.

[12] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk, "On automatically generating commit messages via summarization of source code changes," in *Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM. IEEE, 2014, pp. 275–284.

[13] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, A. Marcus, and G. Canfora, "Automatic generation of release notes," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE. ACM, 2014, pp. 484–495.

[14] E. Reiter, R. Dale, and Z. Feng, *Building Natural Language Generation Systems*. MIT Press, 2000.

[15] W. Wahlster, E. André, W. Finkler, H.-J. Profitlich, and T. Rist, "Plan-based integration of natural language and graphics generation," *Artificial Intelligence*, vol. 63, no. 1-2, pp. 387–427, 1993.

[16] V. O. Mittal, S. F. Roth, J. D. Moore, J. Mattis, and G. Carenini, "Generating explanatory captions for information graphics," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, ser. IJCAI, 1995, pp. 1276–1283.

[17] E. R. Tufte, *Beautiful Evidence*, 1st ed. Graphics Press, 2006.

[18] P. Goffin, J. Boy, W. Willett, and P. Isenberg, "An exploratory study of word-scale graphics in data-rich text documents," *IEEE Transactions on Visualization and Computer Graphics*, 2016 (online first).

[19] F. Beck and D. Weiskopf, "Word-sized graphics for scientific texts," *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, no. 6, pp. 1576–1587, 2017.

[20] M. Harward, W. Irwin, and N. Churcher, "In situ software visualisation," in *Proceedings of the 21st Australian Software Engineering Conference*, ser. ASWEC. IEEE Computer Society, 2010, pp. 171–180.

[21] D. Röthlisberger, M. Härry, W. Binder, P. Moret, D. Ansaloni, A. Villazón, and O. Nierstrasz, "Exploiting dynamic information in IDEs improves speed and correctness of software maintenance tasks," *IEEE Transactions on Software Engineering*, vol. 38, no. 3, pp. 579–591, 2012.

[22] F. Beck, O. Moseler, S. Diehl, and G. D. Rey, "In situ understanding of performance bottlenecks through visually augmented code," in *Proceedings of the 21st IEEE International Conference on Program Comprehension*, ser. ICPC. IEEE, 2013, pp. 63–72.

[23] S. Baltes, O. Moseler, F. Beck, and S. Diehl, "Navigate, understand, communicate: How developers locate performance bugs," in *Proceedings of the 9th International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM. IEEE, 2015, pp. 1–10.

[24] E. Murphy-Hill and A. P. Black, "An interactive ambient visualization for code smells," in *Proceedings of the 5th International Symposium on Software Visualization*, ser. SOFTVIS. ACM, 2010, pp. 5–14.

[25] F. Beck, F. Hollerich, S. Diehl, and D. Weiskopf, "Visual monitoring of numeric variables embedded in source code," in *Proceedings of the 1st IEEE Working Conference on Software Visualization*, ser. VISSOFT. IEEE, 2013, pp. 1–4.

[26] F. Beck, B. Dit, J. Velasco-Madden, D. Weiskopf, and D. Poshyvanyk, "Rethinking user interfaces for feature location," in *Proceedings of the 23rd IEEE International Conference on Program Comprehension*, ser. ICPC. IEEE, 2015, pp. 151–162.

[27] P. Goffin, W. Willett, J.-D. Fekete, and P. Isenberg, "Exploring the placement and design of word-scale visualizations," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2291–2300, 2014.

[28] P. Goffin, W. Willett, A. Bezerianos, and P. Isenberg, "Exploring the effect of word-scale visualizations on reading behavior," in *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA. ACM, 2015, pp. 1827–1832.

[29] A. Zeller and D. Lütkehaus, "DDD—a free graphical front-end for UNIX debuggers," *ACM Sigplan Notices*, vol. 31, no. 1, pp. 22–27, 1996.

[30] A. J. Ko and B. A. Myers, "Finding causes of program output with the Java Whyline," in *Proceedings of the 27th International Conference on Human Factors in Computing Systems*, ser. CHI. ACM, 2009, pp. 1569–1578.

[31] A. J. Ko and B. Myers, "Debugging reinvented: Asking and answering why and why not questions about program behavior," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE. ACM, 2008, pp. 301–310.

[32] S. G. Eick and J. L. Steffen, "Visualizing code profiling line oriented statistics," in *Proceedings of the 3rd Conference on Visualization*, ser. VIS. IEEE Computer Society Press, 1992, pp. 210–217.

[33] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang, "Visualizing the execution of Java programs," in *Software Visualization*. Springer, 2002, pp. 151–162.

[34] A. Bergel, F. Bañados, R. Robbes, and W. Binder, "Execution Profiling Blueprints," *Software: Practice and Experience*, vol. 42, no. 9, pp. 1165–1192, 2012.

[35] J. P. Sandoval Alcocer, A. Bergel, S. Ducasse, and M. Denker, "Performance Evolution Blueprint: Understanding the impact of software evolution on performance," in *Proceedings of the 1st IEEE Working Conference on Software Visualization*, ser. VISSOFT. IEEE, 2013, pp. 1–9.

[36] K. E. Isaacs, A. Giménez, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer, "State of the art of performance visualization," in *EuroVis - STARs*, ser. EuroVis. Eurographics Association, 2014, pp. 141–160.

[37] L. A. Schultheiss and E. M. Heiliger, "Techniques of flow-charting," in *Proceedings of the Clinic on Library Applications of Data Processing*. Graduate School of Library Science. University of Illinois at Urbana-Champaign, 1963, pp. 62–78.