Interactive Graph Exploration for Comprehension of Static Analysis Results

Rafael Toledo
University of Waterloo
rftoledo@uwaterloo.ca

Abstract—Static analysis results can be overwhelming depending on their complexity and the total number of results. Interactive graph visualization can help engineers explore the connections between different code entities while visually supporting insights about the code's behaviour. In our doctoral research, we aim to investigate how a graphical model of a program and its analysis results can support the engineer’s understanding. We expect that a graphical interface can ease the diagnosis of faults and reduce the cognitive load required to comprehend reported control and data flows present in the codebase.

Index Terms—Static Analysis, Program Comprehension, Graph Visualization

I. INTRODUCTION

Engineers use static analysis to learn facts about their program without necessarily running it. However, massive heterogeneous distributed systems (e.g., automotive software) can be too large to analyze, resulting in highly-complex and overwhelming results. To deal with those systems, we extract a lightweight graphical model of the program data, comprising a collection of facts of the source code. Such a graphical model can be derived automatically by parsing the source code and extracting facts from the code’s abstract syntax tree (AST). Extracted facts include code entities (e.g., variables, classes, and functions), their relationships (e.g., calls, reads, and writes), and their attributes (e.g., location in the source code, indicators of control flow influence, presence conditions) [15]. Therefore, an attributed graph can represent the resulting fact base, and graph queries can express the static analyses of the program. Moreover, graph databases (e.g., Neo4j database) storing the software model accommodate large fact bases, flexible query language, and optimized queries and graphs.

Engineers pose questions about their software to support the comprehension of program behaviour (e.g., control-flow and data-flow based queries) and detection of unexpected interactions among disjoint software components. Flow-based queries on a large software system can result in large numbers of complicated analyses’ results, and it may be unrealistic to expect the engineer to understand and triage them all. For example, a control or data-flow analysis of a single electronic control unit (ECU) of 1 million lines of automotive software can output 25,000 paths for the engineer to inspect. To ease the comprehension and inspection of analysis results, we seek to adapt graph visualization techniques by using the structure of program data to design effective visual abstractions (e.g., based on different software units like functions, modules, components) and using the problem domain of analysis to triage results (e.g., based on the analysis query and the relative importance of query results).

Specifically, we plan to study visual abstractions and interactive methods, and assess their support for incremental exploration of a large graphical program model. Our goal is to leverage graph visualization technologies to support the engineer’s mental model of the software and their understanding of how the analysis results over the graphical model map back to the code. Our research plan has the following steps: (i) We will perform a literature review of program-comprehension questions to identify queries of interest that can be applied to graphical models of program data, particularly graph-based queries such as variants of control-flow and data-flow analyses that help with the comprehension of program behaviour and detection of interactions among components; and (ii) We will apply the science of design methodology to develop an interactive visualization tool to support the incremental exploration of query results.

II. RELATED WORK

There have been a number of early exploratory user studies [10, 13, 18], and recent follow-up works [2, 11, 12, 19] that categorize program comprehension questions and engineers’ information needs. We will perform a literature review of this field to identify questions of interest to engineers that can be applied to graphical models of program data, especially those supporting program comprehension and interaction detection.

Previous works using graph visualizations on program data encoded a program’s data structures [1, 5, 23], software architecture [6, 16, 20–22], and control and data-flow [3, 8, 14]. Clustering, semantic zooming, neighbourhood highlighting, and view distortion are some of the visualization operations used by those applications to support the graphical exploration. Herman et al. [7] define that incremental exploration of a large graph must have a strategy to generate new logical frames and rearrange the content of the current view after each change. In our case, we must consider the current engineer’s understanding of the analysis results and their interest in editing data (e.g., adding function calls, grouping variable nodes, changing visual abstraction) from the visualization to accomplish the goal of their exploration.

Most researchers and companies using interactive methods for the triage and visualization of static analysis results still rely on code navigation as the preferable means of program
comprehension. Path Projection [9] project code excerpts that correspond to the nodes of a reported call graph and include a checklist to help triage race condition reports. CodeSonar [4] also present code projections of analysis results and call graphs of selected functions. The analysis results are sorted based on a score representing the true-positive likelihood, severity level, and potential security threats. REACHER [14] provides upstream and downstream searches along the program’s control flow, resulting in interactive call graphs with visual cues encoding order, repetition, and conditionality of calls. VarXplorer [17] presents feature interaction graphs and removes the benign interactions indicated by the engineer to direct the engineer’s focus to those requiring further inspection. The information provided by the engineer is also used to reduce the size of subsequent queries.

Our work differs from others in terms of the data being represented and the overall purpose of the visualization. We focus on the analyses of a graphical model of a program to help engineers understand the program’s behaviour and to detect interactions among components. We want to take advantage of the graphical structure of program data and the optimized path-based queries of a graph database to pose queries about the program’s control and data flow. Therefore, our tool should use graph visualization techniques (e.g., abstractions, filters, triage) to reduce the number and complexity of query results, and allow focused exploration. The engineer can use flexible query languages provided by the graph databases to express ad-hoc queries about the fact base, enabling a broad range of user-defined checks. Therefore, the relevance of the query results may vary depending on their domain-specific triage strategies. The investigation of appropriate triage strategies and the development of ways to express them are also part of our work.

III. HYPOTHESIS AND METHODOLOGY

Our research methodology is composed of two stages. First, we will investigate the matches between code-comprehension and model-comprehension questions. Second, we will use the discovered matches to design a tool to increase the efficiency of the developer’s comprehension and triaging of static-analysis results.

A. Program and model comprehension

Several studies categorize program comprehension questions, but they mostly apply to the software source code and need to be adapted to questions about the graphical program model. Based on that, we define our first hypothesis as:

**Hypothesis 1:** The adaptation of code-comprehension questions to model-comprehension questions is sufficient to identify control-flow and data-flow analysis queries that help with the comprehension of program behaviour and detection of interactions among software components.

To validate our hypothesis, we will perform a literature review to identify the categories of questions asked during program comprehension that can be addressed by a graphical model of the program data. Given the number of studies regarding program comprehension questions, we believe that it would be redundant to run a new exploratory study to identify comprehension questions of interest to engineers. Moreover, a review of those works can help us identify the nature of the engineers’ information needs and their alignment with the intuition that graphical software models can provide.

If our adaptation is found to be deficient, we will test the quality of the identified graph-based queries with users. Moreover, we will perform a formative user study to identify the static analysis queries over graphical model of program data of most interest to the engineer.

B. Interactive exploration of analyses results

In the second stage, we will investigate primitive operations of interactive large graph visualization that can support the first stage questions. Based on that, we define our second hypothesis as:

**Hypothesis 2:** An interactive graph exploration tool can improve the efficiency of the engineer’s understanding of the analysis results.

We will validate this hypothesis by applying the design science methodology to develop a graph exploration tool iteratively. We will identify primitive operations (e.g., clustering, filtering, distortion, visual cues) that can serve as the engineer’s vocabulary for exploring large graphs representing analysis results.

The alignment between program comprehension questions and graphical software data models from the first stage will guide our experimental prototypes’ development. The prototype evaluations will use quantitative and qualitative methods to measure the difference in time taken to understand the analysis results, the cognitive load demanded to achieve such understanding, and the correctness of the user’s comprehension of the program behaviour. Those efficient gains are expected to improve engineers’ capacity to determine whether an analysis result indicates a fault to be fixed or irrelevant control and data paths.

IV. EXPECTED CONTRIBUTION

The thesis will investigate how interactive graph representation of program data can support the engineer’s analysis comprehension. The expected contributions of the thesis are:

- The identification of software-comprehension questions that engineers can ask a graphical model of program data
- A tool to support the incremental exploration of static-analysis results over a graphical model of program data, enabling a more efficient comprehension and inspection of the system

The interactive experience of visualizing graphical program models of static-analyses results should ease the investigation of a program’s reported control and data flows. Once the engineer understands the dynamics of code artifacts reported by the analyses, they can determine whether an analysis result represents a program fault. This comprehension of potential faults contributes to the design of necessary fixes.
REFERENCES

[1] Edward E Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L Su, and Samuel Z Geyer. Heapviz: interactive heap visualization for program understanding and debugging. In Proceedings of the 5th international symposium on Software visualization, pages 53–62, 2010.

[2] Abdullah Al-Nayeem, Krzysztof Ostrowski, Sebastian Pueblas, Christophe Restif, and Sai Zhang. Information needs for validating evolving software systems: An exploratory study at google. In 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), pages 544–545. IEEE, 2017.

[3] Andrew Bragdon, Steven P Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J LaViola Jr. Code bubbles: rethinking the user interface paradigm of integrated development environments. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, pages 455–464, 2010.

[4] CodeSonar. https://www.grammatech.com/codesonar-cc.

[5] Wim De Pauw and Gary Sevitsky. Visualizing reference patterns for solving memory leaks in java. In European Conference on Object-Oriented Programming, pages 116–134. Springer, 1999.

[6] Carsten Gutwenger, Michael Jünger, Karsten Klein, Joachim Kupke, Sebastian Leipert, and Petra Mutzel. A new approach for visualizing uml class diagrams. In Proceedings of the 2003 ACM symposium on Software visualization, pages 179–188, 2003.

[7] Ivan Herman, Guy Melançon, and M Scott Marshall. Graph visualization and navigation in information visualization: A survey. IEEE Transactions on visualization and computer graphics, 6(1):24–43, 2000.

[8] Takashi Ishio, Shogo Etsuda, and Katsuro Inoue. A lightweight visualization of interprocedural data-flow paths for source code reading. In 2012 20th IEEE International Conference on Program Comprehension (ICPC), pages 37–46. IEEE, 2012.

[9] Yit Phang Khoo, Jeffrey S Foster, Michael Hicks, and Vibha Szawawal. Path projection for user-centered static analysis tools. In Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pages 57–63, 2008.

[10] Andrew J Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In 29th International Conference on Software Engineering (ICSE’07), pages 344–353. IEEE, 2007.

[11] Jacob Krüger and Regina Hebig. What developers (care to) recall: An interview survey on smaller systems. In 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 46–57. IEEE, 2020.

[12] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. Live programming and software evolution: questions during a programming change task. In 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), pages 30–41. IEEE, 2019.

[13] Thomas D LaToza and Brad A Myers. Hard-to-answer questions about code. In Evaluation and Usability of Programming Languages and Tools, pages 1–6. Association for Computing Machinery, 2010.

[14] Thomas D LaToza and Brad A Myers. Visualizing call graphs. In 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pages 117–124. IEEE, 2011.

[15] Bryan J Muscedere, Robert Hackman, Davood Anbarian, Joanne M Atlee, Ian J Davis, and Michael W Godfrey. Detecting Feature-Interaction Symptoms in Automotive Software Using Lightweight Analysis. In Proc. of SANER’19. IEEE, 2019.

[16] Andreas Noack and Claus Lewerentz. A space of layout styles for hierarchical graph models of software systems. In Proceedings of the 2005 ACM symposium on Software visualization, pages 155–164, 2005.

[17] Larissa Rocha, Ivan Machado, Eduardo Almeida, Christian Kästner, and Sarah Nadi. A semi-automated iterative process for detecting feature interactions. In 34th Brazilian Symposium on Software Engineering (SBES ’20), 2020.

[18] Jonathan Sillito, Gail C Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, pages 23–34, 2006.

[19] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pages 248–259, 2015.

[20] M-AD Storey and Haudi A Muller. Manipulating and documenting software structures using shrimp views. In Proceedings of International Conference on Software Maintenance, pages 275–284. IEEE, 1995.

[21] Nikita Synytskyy, Richard C Holt, and Ian Davis. Browsing software architectures with Isedit. In 13th International Workshop on Program Comprehension (IWPC’05), pages 176–178. IEEE, 2005.

[22] Xhevahirë Tërnava, Johann Mortara, and Philippe Collet. Identifying and visualizing variability in object-oriented variability-rich systems. In Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A, pages 231–243, 2019.

[23] Thomas Zimmermann and Andreas Zeller. Visualizing memory graphs. In Software Visualization, pages 191–204. Springer, 2002.