VISUFLOW: a Debugging Environment for Static Analyses

Lisa Nguyen Quang Do
Fraunhofer IEM
lisa.nguyen@iem.fraunhofer.de

Stefan Krüger
Paderborn University
stefan.krueger@upb.de

Patrick Hill
Paderborn University
pahill@campus.uni-paderborn.de

Karim Ali
University of Alberta
karim.ali@ualberta.ca

Eric Bodden
Paderborn University & Fraunhofer IEM
eric.bodden@upb.de

ABSTRACT

Code developers in industry frequently use static analysis tools to detect and fix software defects in their code. But what about defects in the static analyses themselves? While debugging application code is a difficult, time-consuming task, debugging a static analysis is even harder. We have surveyed 115 static analysis writers to determine what makes static analysis difficult to debug, and to identify which debugging features would be desirable for static analysis. Based on this information, we have created VISUFLOW, a debugging environment for static data-flow analysis. VISUFLOW is built as an Eclipse plugin, and supports analyses written on top of the program analysis framework SOOT. The different components in VISUFLOW provide analysis writers with visualizations of the internal computations of the analysis, and actionable debugging features to support debugging static analyses. A video demo of VISUFLOW is available online: https://www.youtube.com/watch?v=BkEfBDwiuH4

CCS CONCEPTS

•Software and its engineering → Software testing and debugging; •Theory of computation → Program analysis; •Human-centered computing → Empirical studies in visualization;

KEYWORDS

Debugging, Static analysis, IDE, Survey, User Study, Empirical Software Engineering

1 INTRODUCTION

As more and more complex software is written every day, ensuring its functionality, quality, and security becomes increasingly important and difficult to achieve. Static analysis is particularly useful in that regard, because it allows developers to reason even about partial/incomplete programs. Researchers and practitioners are continuously contributing to various static analysis frameworks [3, 4, 9, 13]. Yet, as application code gets more sophisticated, writing a static analysis for it becomes increasingly harder as well, and, as we show in this paper, debugging the analysis can prove more complicated than debugging the analyzed code.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICSE '18 Companion, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s). 978-1-4503-5663-3/18/05…$15.00
DOI: 10.1145/3183440.3183470

We have conducted a large-scale survey [11] with 115 analysis writers to determine the difficulties of debugging static analysis compared to general application code. Our findings show that bugs found in static analyses are quite different from those generally found in application code. Current debugging tools do not support debugging those specific bugs (e.g., wrong assumptions on how the static analysis framework interprets the analyzed code), making it much harder for analysis writers to find the cause of an error. The main cause of this problem is the inability to visualize how the analysis behaves at a given point in time.

Based on the responses we received from our survey participants, we have identified the debugging features required to provide analysis writers with helpful visuals. In this paper, we present VISUFLOW, a debugging environment for static analysis built in Eclipse, an integrated development environment (IDE). VISUFLOW is designed to help analysis writers debug their analyses through comprehensive visualizations. In particular, it provides the following features:

- Access to the intermediate representation of the analyzed code.
- Interactive graph visualizations of the analyzed code.
- Overview of the intermediate results of the analysis.
- Breakpoint and stepping functionalities for both the analysis code and the analyzed code.

2 MOTIVATION

Our survey explores how analysis writers debug static analysis and which features are most useful when debugging static analyses compared to general application code. The 115 participants cover different branches of static analysis, including the analyzed language, the types of static analysis (e.g., data-flow, abstract interpretation), and the analysis frameworks. We have made the anonymized responses available online [10].

We observed that 5.3× more participants find static analyses harder to debug than application code, because debugging static analyses requires one to comprehend two different codebases (the analysis code and the analyzed code) instead of just the application code, resulting in more complex corner cases. Additionally, the correctness of an analysis is not directly verifiable, as opposed to the output of application code: "Static analysis code usually deals with massive amounts of data. [...] It is harder to see where a certain state is computed, or even worse, why it is not computed."

Those specific properties of static analysis directly influence the types of bugs that are found in static-analysis code. In our study, 81.3% of the participants report that the main cause of bugs in application code is programming errors such as "wrong conditions, wrong loops statements". While those bugs exist in static analyses,

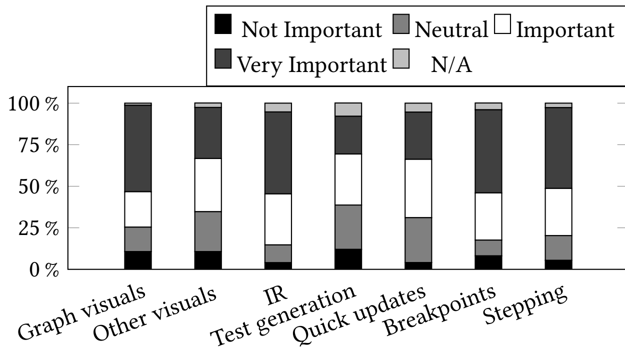

Figure 1: Importance of the features for debugging static analysis. IR denotes Intermediate Representation.

they are only mentioned by 41.7% of the participants. Corner cases, algorithmic errors, and handling the analysis semantics and infrastructure are significantly more prevalent in static analysis than in application code.

Since the bugs found in static analyses are different from the ones found in application code, one would expect different debugging tools to be used for those two types of code. However, our survey showed that regardless of the type of code that participants write, they use the same techniques: breakpoints and stepping, variable inspection, printing intermediate results, debugging tools such as gdb or the Eclipse integrated debugger, and coding support such as auto-completion. Interestingly, participants expressed dissatisfaction with their debugging tools in general. This shows that existing debugging tools are not sufficient to fully support static analysis writers: "While the IDE can show a path through [my] code for a symbolic execution run, it doesn't show analysis states along that path." Overall, current debugging tools miss one crucial component to properly support static analysis: visibility of what is (not) computed in the analysis at a given point of its execution.

We asked the participants which debugging features would be useful in a debugging environment and noticed a significant difference between the features requested to debug static analysis and application code (($p = 0.04 \leq 0.05$) for a $\mathcal{X}^2$ test). For application code, participants requested better coding support and hot-code replacement. For static analysis, participants requested better visualizations of the analysis constructs such as the intermediate representation of the code or intermediate results of the analysis (18.4%), graph visualizations of the analysis (23.7%), omniscient debugging (13.2%) [8], and better breakpoints and stepping functionalities that would allow them to step through both the analysis code and the analyzed code at the same time. Figure 1 details the relative importance of those debugging features.

Through our survey, we see that current debugging tools are designed for application code, and while helpful, they are not sufficient to fully support debugging static analysis. We identified the following debugging features that a static analysis debugger should provide: graph visualizations, access to analysis intermediate results, and better conditional breakpoints.

3 VISUFLOW

Based on the features identified in the survey, we present VisuFlow, a debugging environment for static analyses. We have implemented VisuFlow on top of the Eclipse IDE, and it supports debugging static data-flow analyses written on top of the Soot [14] analysis framework. Our survey shows the importance of displaying both the analysis code and the analyzed code, and in particular, highlighting how the former handles the latter. With this goal in mind, we have designed VisuFlow provide such information, presented in an understandable and usable manner.

We detail below the main functionalities of VisuFlow, illustrated in Figure 2, using the corresponding numbers.

1. **Java Editor:** We used the standard Eclipse Java Editor and its functionalities (e.g., Eclipse breakpoints) to display the analysis code, since providing users with familiar views and functionalities allows for a better integration of VisuFlow into the Eclipse IDE. We extended the Java Editor to add navigation functions between this view and the other views, as detailed below.
2. **Jimple View:** Soot converts Java code to an intermediate representation called Jimple, which it then analyzes. To show analysis writers how the analysis handles Jimple code, we introduced a Jimple View that shows the analyzed code in Jimple format. The Jimple code is not editable, because it is automatically generated by Soot. Similar to the Java Editor, the Jimple View offers navigation functionalities to other views.
3. **Unit Breakpoints:** In the Jimple View, analysis writers can set special breakpoints called *Unit Breakpoints* that stop the execution of the analysis at a given *unit* (i.e., Jimple statement). Once the execution stops, VisuFlow highlights the Jimple statement being currently analyzed, and the user may step through the intermediate representation of the analyzed code to debug their analysis. By jointly using the stepping functionalities of the Unit Breakpoints and the Java Editor's breakpoints, analysis writers can step through both code bases at the same time, without needing to write complex conditional breakpoints in the standard Eclipse debugger.
4. **Graph View:** By default, this view displays the call graph of the analyzed code. The user can explore the control flow graph (CFG) of a particular method by selecting its node in the call graph. One can also navigate between the CFGs of different methods through navigation menus as shown in Figure 2. On the edges of the CFGs, VisuFlow displays the information propagated by the analysis. Such access to the intermediate results of the analysis allows users to follow the data flows and locate miscalculations more easily. When stepping through the Jimple code, VisuFlow highlights the corresponding unit in the Graph View. Tooltips give access to more information about the different units.

   The Graph View is intended to give the user a better understanding of the structure of the analyzed code, while also providing a more visual approach to debugging. It combines different information about the analysis, the analyzed code, and the intermediate results, thus providing an easy way to keep track of multiple things without cluttering the interface with code. These characteristics also cause the graphs to act as navigational hubs from which the user can jump to the different views based on what is observed in the graph to gather more information. The

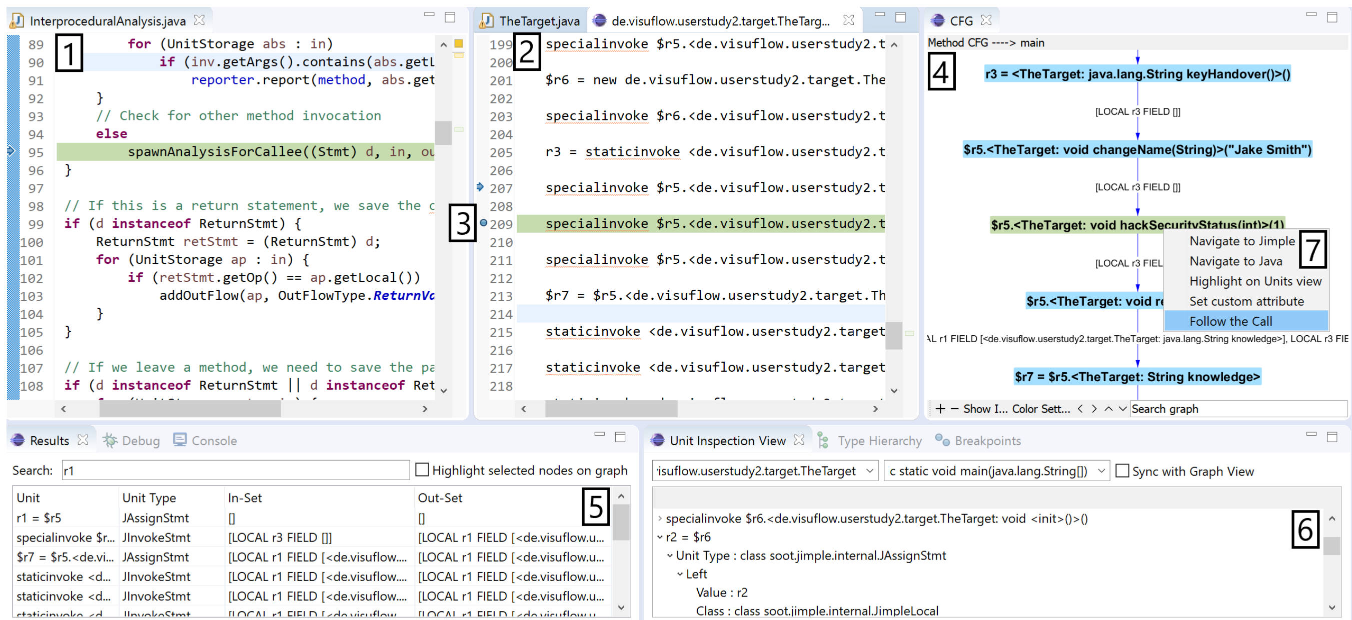

Figure 2: The graphical user interface of VISUFLOW. We describe the labeled views in Section 3.

graph information, retrieved from the data model, is displayed using the open source framework GraphStream [2], which scales to graphs with a large number of nodes.

5. **Results View:** The intermediate results are also shown in the Results View, along with additional information such as editable tags that the developer can use to mark specific units. This view also contains searching and filtering functionalities, which is especially helpful for methods with a large number of statements.
6. **Unit Inspection View:** This view enables analysis writers who are unfamiliar with SOOT and Jimple to inspect Jimple statements and see how they are constructed. Since SOOT analyses manipulate Jimple statements, understanding Jimple units helps analysis writers design appropriate analysis rules, or judge the correctness of existing ones.
7. **Synchronization and Navigation:** VISUFLOW allows users to synchronize the Unit Inspection View, the Graph View and the Results View so that statements that are selected in a view are also shown in the other views. In order to connect the different views, we enriched VISUFLOW with navigation features that allow developers to switch between views through drop-down menus, as illustrated in Figure 2. The synchronization functionalities provide a smooth navigation between all views that does not disrupt the analysis writer by forcing them to manually look for a specific statement when switching between views.

The different views and breakpoints of VISUFLOW are implemented as an extension of the standard Eclipse UI components. To populate the views with analysis information, VISUFLOW maintains an internal data model of the intermediate representation of the analyzed code. The tool hooks into Eclipse's builder and uses Eclipse's OSGi-Event Model to populate and update this data model at every change in the code base. The intermediate results of the analysis are updated every time the analysis is re-run, using a Java agent to instrument the analysis to collect the information at runtime.

4 EVALUATION

To evaluate the usefulness of VISUFLOW, we conducted a user study with 20 participants. We prepared faulty analyses that do not produce correct results for a given piece of analyzed code. Each participant debugged two such analyses, each containing three errors, and used VISUFLOW for one analysis, and the standard Eclipse debugging environment (hereafter named *Eclipse*) for the other. Half of the participants used VISUFLOW first, and the other half Eclipse first. We recorded how many errors the participants could identify and fix, how long they spent using each feature of the tools, and asked them to fill a comparative questionnaire and to discuss their impressions of the two tools. The results are available online [10].

Figure 1 shows that the IDE features that were most used by the participants include many of VISUFLOW's features (i.e., the Graph View, the Jimple View, the breakpoints, and the Results View). With VISUFLOW, participants spent 25.6% less time using the Java Editor, and 44.4% less time stepping through the code. Instead, they spent this time using the Graph View and the Results View. This shows that graphs, special breakpoints, and access to the intermediate representation and to the intermediate results are desirable features in a debugging environment for static analyses, confirming the findings of our survey. Moreover, Eclipse's breakpoints editor was used 88.2% less often in VISUFLOW than in Eclipse. We attribute this to the special breakpoint features in VISUFLOW, which relieve users from defining complex conditional breakpoints.

Overall, participants identified 25% and fixed 50% more errors with VISUFLOW than with Eclipse. For Task 1, they identified and fixed 1.4× more errors, and for Task 2, they identified 1.1× and fixed

Table 1: Main features of VISUFLOW and Eclipse that participants used, and the average time spent using each feature.

	VISUFLOW		Eclipse	
	#users	Time (s)	#users	Time (s)
Java Editor	14	486	14	653
Graph View	14	201	n/a	n/a
Jimple View	11	58	12	60
Breakpoints / Stepping	11	174	11	313
Variable Inspection	3	78	8	67
Results View	8	50	n/a	n/a
Console	5	24	7	40
Drop Frame	5	12	3	5
Breakpoints View	3	13	2	110
Unit View	3	7	n/a	n/a

1.6× more errors. Out of the 20 participants in our study, 12 participants are used to debugging their analyses using Eclipse. However, 7 of them still found and fixed more errors using VISUFLOW.

In general, VISUFLOW was positively received by the participants of our user study. In the questionnaire, they gave it a Net Promoter Score [12] of 9.1/10 compared to Eclipse (standard deviation $\sigma = 1.1$), and 8.3/10 compared to their own coding environment ($\sigma = 1.7$). In the questionnaire and interviews, participants stressed the usefulness of the Graph, Jimple, and Results View, and VISUFLOW's special breakpoints, especially in understanding both the analysis and the analyzed code, and how they interacted: "I think [VISUFLOW] is helpful because of the linkage between the Java code, the Jimple code and the graphic visualization: all that I had to keep in my mind [earlier].".

5 RELATED WORK

While we are not aware of any debugging tool that targets writers of static analyses as VISUFLOW does, a few tools have been published that offer at least subsets of VISUFLOW's features. By means of integrated Eclipse views, the software-analysis platform Atlas [5] is able to visualize data-flows through a given program. Path Projection [6] supports users of static analysis in understanding error messages and locating the related lines of code. However, neither of these tools target analysis developers, and therefore, they do not have the comprehensive feature set that VISUFLOW provides for debugging static analyses.

VISUFLOW is tightly integrated into Eclipse. Consequently, its users can use all of Eclipse's integrated debugging functionalities for Java, such as breakpoints and stepping. While these are general-purpose debugging features that lack the necessary focus on static analysis, they still prove useful in any scenario involving debugging. VISUFLOW does not support more complex paradigms for debugging such as delta [15], omniscient [8], and interrogative debugging [7]. We leave the exploration of how VISUFLOW could benefit from their integration to future work.

6 CONCLUSION

We presented VISUFLOW, a debugging environment for static analysis. Through a user study, we demonstrated that the features in VISUFLOW, designed to help visualize the internal computations of the analysis, help analysis writers identify and fix more errors than with the standard Eclipse IDE. In future work, we plan to explore better visualizations for the analysis of large code bases, especially in terms of collapsable graphs and quick response to user modifications in either of the code bases. VISUFLOW instantiates several of the debugging features identified in our survey for SOOT-based, static data-flow analysis. It would be interesting to explore how to adapt the features for other types of static analysis. Other features suggested in the survey (e.g., omniscient debugging and quick updates) are not currently offered by VISUFLOW. We plan to explore adding those features to VISUFLOW in the future. The anonymized answers to the survey and user study are available online [10]. VISUFLOW is open source [10], and we welcome contributions under the Apache 2.0 licence [1].

ACKNOWLEDGEMENTS

We thank Henrik Niehaus, Shashank Basavapatna Subramanya, Kaarthik Rao Bekal Radhakrishna, Zafar Habeeb Syed, Nishitha Shivegowda, Yannick Kouotang Signe, and Ram Muthiah Bose Muthian for their work on the implementation of VISUFLOW. This research was supported by a Fraunhofer Attract grant as well as the Heinz Nixdorf Foundation. This work has also been partially funded by the DFG as part of project E1 within the CRC 1119 CROSSING, and was supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] 2017. Apache License 2.0. https://www.apache.org/licenses/LICENSE-2.0. (2017).
[2] 2017. GraphStream. http://graphstream-project.org/. (2017).
[3] Eric Bodden. 2012. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *International Workshop on State of the Art in Java Program Analysis (SOAP)*. 3–8. https://doi.org/10.1145/2259051.2259052
[4] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods (NFM) (Lecture Notes in Computer Science)*, Vol. 6617. 459–465.
[5] Tom Deering, Suresh Kothari, Jeremias Sauceda, and Jon Mathews. 2014. Atlas: A New Way to Explore Software, Build Analysis Tools *(ICSE Companion 2014)*. ACM, New York, NY, USA, 588–591. https://doi.org/10.1145/2591062.2591065
[6] Yit Phang Khoo, Jeffrey S. Foster, Michael Hicks, and Vibha Sazawal. 2008. Path Projection for User-centered Static Analysis Tools *(PASTE '08)*. ACM, New York, NY, USA, 57–63. https://doi.org/10.1145/1512475.1512488
[7] Andrew Jensen Ko and Brad A. Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *CHI 2004, Vienna, Austria, April 24 - 29, 2004*. 151–158.
[8] Bil Lewis. 2003. Debugging Backwards in Time. *CoRR* cs.SE/0310016 (2003). http://arxiv.org/abs/cs.SE/0310016
[9] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. 2017. Just-in-time Static Analysis *(ISSTA 2017)*. ACM, New York, NY, USA, 307–317.
[10] Lisa Nguyen Quang Do, Stefan Krüger, Patrick Hill, Karim Ali, and Eric Bodden. 2017. VisuFlow. https://blogs.uni-paderborn.de/sse/tools/visuflow-debugging-static-analysis/. (2017).
[11] Lisa Nguyen Quang Do, Stefan Krüger, Patrick Hill, Karim Ali, and Eric Bodden. 2018. *Debugging Static Analysis*. Technical Report. arXiv:cs.SE/1801.04894
[12] Frederick F Reichheld. 2003. The one number you need to grow. *Harvard Business Review* 81, 12 (2003), 46–55.
[13] Haihao Shen, Jianhong Fang, and Jianjun Zhao. 2011. Efindbugs: Effective error ranking for findbugs. In *International Conference on Software Testing, Verification and Validation (ICST)*. 299–308.
[14] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. 2000. Optimizing Java Bytecode Using the Soot Framework: Is It Feasible?. In *CC*. 18–34. https://doi.org/10.1007/3-540-46423-9_2
[15] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200.