How Should Static Analysis Tools Explain Anomalies to Developers?

Titus Barik
North Carolina State University, USA
tbarik@ncsu.edu

ABSTRACT

Despite the advanced static analysis tools available within modern integrated development environments (IDEs) for detecting anomalies, the error messages these tools produce to describe these anomalies remain perplexing for developers to comprehend. This research postulates that tools can computationally expose their internal reasoning processes to generate assistive *error explanations* that more closely align with how developers explain errors to themselves. My work demonstrates that tools stand to significantly benefit if they incorporate explanation principles in their design.

1. INTRODUCTION

Modern software development typically occurs within an integrated development environment (IDE), such as Eclipse, Visual Studio, and IntelliJ.[1] One task developers perform within this IDE is understanding and fixing anomalies that static analysis tools identify within the environment. These anomalies are presented to the developer through the IDE as *error messages*. Despite the sophisticated reasoning processes of static analysis tools such as clang [1] and Coverity [10] for surfacing program analysis details, developers continue to have difficulty understanding the messages the tools produce [12, 19, 5]. Consequently, static analysis anomalies remain perplexing for developers to resolve [25].

The *objective* of my research is to identify and address the difficulties that developers face during error comprehension and resolution tasks. I argue that static analysis error messages should be reframed as assistive *error explanations* to better align and support the self-generated explanations [13], or *self-explanations*, that developers use to understand anomalies (Figure 1) [7]. The *significance* of this work is that static analysis explanations have the potential to substantially improve developer comprehension of static analysis anomalies. The expected *contribution* of this work is to advance a theoretical framework that guides how static analysis tools should explain anomalies to developers.

[1]http://pypl.github.io/IDE.html

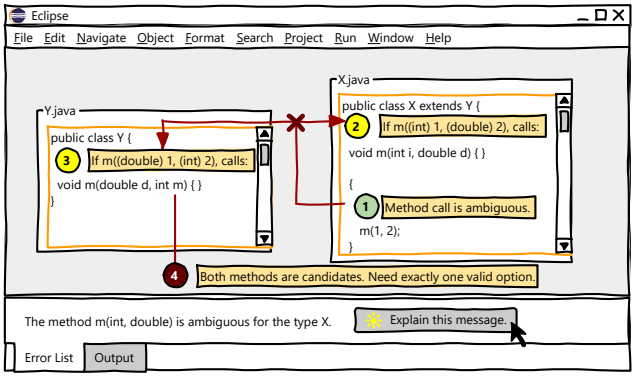

Figure 1: A mock-up of an Eclipse IDE that provides assistive explanations to the developer for an ambiguous method anomaly.

2. BACKGROUND

Self-explanation as a cognitive mechanism. Chi and colleagues found that self-explanation is an essential cognitive process that provides the "cognitive bridge" through which humans translate declarative facts to *understanding* [13]. Since the original finding, self-explanation has been replicated in a variety of domains [4, 15, 24, 27], including computer programming tasks [26, 11]; the cognitive process of self-explanation appears to be not only essential, but ubiquitous [6]. Subsequent work by Chi and colleagues have found that self-explanation can be elicited through explicit prompts [14], whether by humans or by computers [4], and that self-explanation effects are significantly enhanced when diagrams are used over text alone [3].

Computationally supporting self-explanations. Lim and colleagues found that when intelligent systems explain *why* the system behaved a certain way, participants report better understanding and stronger feelings of trust with the system [22]. Kulesza and colleagues also investigated explanations for intelligent systems, focusing on how properties of soundness and completeness affect the fidelity of end users' mental models [18]. Their findings suggest that completeness is more useful than soundness. The Whyline system is a prototype interrogative debugging interface in which developers can ask the system "Why" or "Why not" questions to obtain an explanation of runtime events [17]. Results from the Theseus tool for visualizing runtime behavior showed that users quickly adopted these visualizations and incorporated the information in their own self-explanations [21].

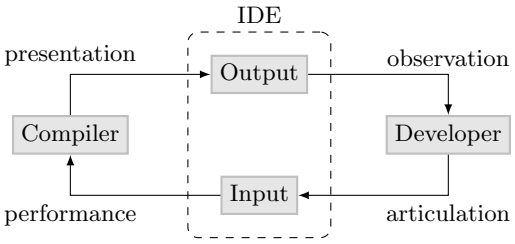

Figure 2: The interaction framework, instantiated for IDEs. Coverage over the framework provides support towards a theory of self-explanation.

3. APPROACH

My approach to building a theory of self-explanation for static analysis anomalies is actualized by providing *coverage* over Abowd and Beale's interaction framework (Figure 2) [25, 2, 9]. This framework comprises four components: a compiler, a developer, an input, and an output. The framework components of input (e.g., entering source code) and output (e.g., an error message) are encapsulated as the IDE. The compiler *presents* an error to the developer through the IDE, and the IDE can augment this presentation by, e.g., visually underlining the offending code. This error must then be *observed* and comprehended by the developer. The developer must then *articulate* a resolution through the IDE, either as a manual edit to the code or through an automated tool in the IDE. The IDE then invokes the compiler, which *performs* the compilation process, and the cycle repeats.

4. RESULTS

Presentation and Observation. To understand how existing presentations in IDEs do not align with the way developers self-explain errors to themselves, I conducted a pencil-and-paper think-aloud study in which developers self-explained source code listings and error messages as presented by Eclipse [7].[2] I asked participants to explain each compiler anomaly while making annotations on the source code during their explanation. *Findings.* First, the availability of error explanations (Figure 1) yields significantly better judgments by developers about the causes of the error. Second, developers adopt conventions from these visualizations in their own explanations, suggesting that they are useful. Third, a cognitive dimensions questionnaire [16] reveals that error explanations are beneficial to developers because they explicate the relationships between different program elements.

Performance and Articulation. The traditional abstraction of compilers is that they are sophisticated black boxes [20]. Our work in the development of FixBugs (Figure 3) demonstrates that it is feasible to retrofit existing compilers to expose explanatory features to the developer.[3] To understand how developers might benefit when toolsmiths move beyond speed as a primary metric for applying fixes to anomalies, we conducted an experiment using FixBugs [8]. In contrast with one-shot *Quick Fixes* provided by IDEs, FixBugs balances the automation provided by development tools with the developers' need to explore and comprehend the design space of resolutions. An analytical, heuristic evaluation [23] identified trade-offs between the design of Quick Fix and FixBugs along several dimensions. Table 1 summarizes the relative preference of FixBugs to Quick Fix from expert evaluators across computer science, psychology, and industry software engineering. *Findings.* From the evaluation feedback, evaluators reported that Quick Fix is preferred when minimal or no design exploration is necessary to apply the fix. FixBugs excels in situations where fixes can be provided, but the permutation of possible design choices within that space have more variation. A descriptive coding of the feedback identified that a) enabling exploration of alternative design spaces, b) providing direct feedback, and c) offering structured manipulation for applying changes, facilitate self-explanation.

[2]http://go.barik.net/errviz
[3]http://go.barik.net/fixbugs

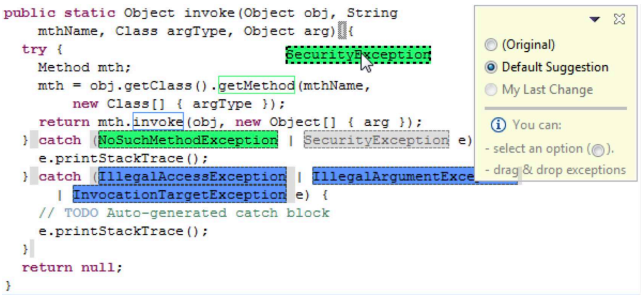

Figure 3: In contrast to Quick Fixes, FixBugs enables design space exploration. Here, the developers can throw or catch exceptions through drag-and-drop actions for resolving a missing exception anomaly.

Table 1: Evaluators' (E1-E6) positive scores indicate preference towards FixBugs and negative scores indicating preference towards Quick Fix. A raised dot (·) indicates that the evaluator did not find the criteria applicable.

	CS		PSY		IND	
Heuristic	E1	E2	E3	E4	E5	E6
Visibility	+2		+1	+1	−4	+1
User control/Freedom	+1	+2	+1	+2	+2	+1
Recognition	+2	+2		+4		+1
Aesthetic		+1		+2	−3	−1
Recognize/Diagnose	+1	·	+1	+2	−2	+2

5. CONCLUSION

The comprehensibility and utility of error messages for static analysis anomalies can be significantly improved by reframing error messages as *error explanations* that more closely align with how developers explain anomalies to themselves. The results of this work contribute towards a theory of how static analysis tools should explain anomalies to developers.

6. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1217700.

7. REFERENCES

[1] Clang Static Analyzer. http://clang-analyzer.llvm.org/.

[2] G. D. Abowd and R. Beale. Users, systems and interfaces: A unifying framework for interaction. In *People and Computers VI*, pages 73–87, 1991.

[3] S. Ainsworth and A. Th Loizou. The effects of self-explaining when learning with text or diagrams. *Cognitive Science*, 27(4):669–681, Aug. 2003.

[4] V. Aleven. An effective metacognitive strategy: learning by doing and explaining with a computer-based Cognitive Tutor. *Cognitive Science*, 26(2):147–179, Apr. 2002.

[5] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *PASTE '07*, pages 1–8, June 2007.

[6] R. Azevedo and V. Aleven, editors. *International Handbook of Metacognition and Learning Technologies*, volume 28. Springer, 2013.

[7] T. Barik, K. Lubick, S. Christie, and E. Murphy-Hill. How developers visualize compiler messages: A foundational approach to notification construction. In *VISSOFT '14*, pages 87–96, Sept. 2014.

[8] T. Barik, Y. Song, B. Johnson, and E. Murphy-Hill. From quick fixes to slow fixes: Reimagining static analysis resolutions to enable design space exploration. In *ICSME '16*, 2016.

[9] T. Barik, J. Witschey, B. Johnson, and E. Murphy-Hill. Compiler error notifications revisited: An interaction-first approach for helping developers more effectively comprehend and resolve error notifications. In *ICSE NIER '14*, pages 536–539, May 2014.

[10] A. Bessey, D. Engler, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, and S. McPeak. A few billion lines of code later. *Communications of the ACM*, 53(2):66–75, Feb. 2010.

[11] K. Bielaczyc, P. L. Pirolli, and A. L. Brown. Training in self-explanation and self-regulation strategies: Investigating the effects of knowledge acquisition activities on problem solving. *Cognition and Instruction*, 13(2):221–252, Jan. 1995.

[12] P. J. Brown. Error messages: The neglected area of the man/machine interface. *Communications of the ACM*, 26(4):246–249, Apr. 1983.

[13] M. T. Chi, M. Bassok, M. W. Lewis, P. Reimann, and R. Glaser. Self-explanations: How students study and use examples in learning to solve problems. *Cognitive Science*, 13(2):145–182, Apr. 1989.

[14] M. T. Chi, N. De Leeuw, M.-H. Chiu, and C. Lavancher. Eliciting self-explanations improves understanding. *Cognitive Science*, 18(3):439–477, July 1994.

[15] K. J. Crippen and B. L. Earl. The impact of web-based worked examples and self-explanation on performance, problem solving, and self-efficacy. *Computers & Education*, 49(3):809–821, Nov. 2007.

[16] T. Green and M. Petre. Usability analysis of visual programming environments: A 'Cognitive Dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996.

[17] A. J. Ko and B. A. Myers. Designing the Whyline. In *CHI '04*, pages 151–158, Apr. 2004.

[18] T. Kulesza, S. Stumpf, M. Burnett, S. Yang, I. Kwan, and W.-K. Wong. Too much, too little, or just right? Ways explanations impact end users' mental models. In *VL/HCC '13*, pages 3–10, Sept. 2013.

[19] S. K. Kummerfeld and J. Kay. The neglected battle fields of syntax errors. pages 105–111, Jan. 2003.

[20] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

[21] T. Lieber, J. R. Brandt, and R. C. Miller. Addressing misconceptions about code with always-on programming visualizations. In *CHI '14*, pages 2481–2490, Apr. 2014.

[22] B. Y. Lim, A. K. Dey, and D. Avrahami. Why and why not explanations improve the intelligibility of context-aware intelligent systems. In *CHI '09*, pages 2119–2129, Apr. 2009.

[23] J. Nielsen. Heuristic evaluation. *Usability inspection methods*, 17(1):25–62, 1994.

[24] P. Reimann and C. Neubert. The role of self-explanation in learning to use a spreadsheet through examples. *Journal of Computer Assisted Learning*, 16(4):316–325, Oct. 2008.

[25] V. J. Traver. On compiler error messages: What they say and what they mean. *Advances in Human-Computer Interaction*, 2010:1–26, 2010.

[26] K. VanLehn and M. R. Johnes. What mediates the self-explanation effect? Knowledge gaps, schemas or analogies? In *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society*, pages 1034–1039, 1993.

[27] M. B. W. Wolfe and S. R. Goldman. Relations Between Adolescents' Text Processing and Reasoning. *Cognition and Instruction*, 23(4):467–502, Dec. 2005.