One λ at a time: What do we know about presenting human-friendly output from program analysis tools?

A Scoping Review of PLDI Proceedings for HCI Researchers

Titus Barik
North Carolina State University

Chris Parnin
North Carolina State University

Emerson Murphy-Hill
North Carolina State University

Abstract

Program analysis tools perform sophisticated analysis on source code to help programmers resolve compiler errors, apply optimizations, and identify security vulnerabilities. Despite the utility of these tools, research suggests that programmers do not frequently adopt them in practice—a primary reason being that the output of these tools is difficult to understand. Towards providing a synthesis of what researchers know about the presentation of program analysis output to programmers, we conducted a scoping review of the PLDI conference proceedings from 1988-2017. The scoping review serves as interim guidance for advancing collaborations between research disciplines. We discuss how cross-disciplinary communities, such as PLATEAU, are critical to improving the usability of program analysis tools.

1 Introduction

In 1983, Brown [12] lamented that one of the most neglected aspects of the human-machine interface was the quality of the error messages produced by the machine. Today, it appears that many of Brown's laments still hold true with regard to program analysis tools—tools that are intended to help programmers resolve defects in their code. For example, interview and survey studies conducted at Microsoft reveal that poor error messages remain one of the top pain points when using program analysis tools [16], and other studies show similar frustration with error messages in tools [7, 33, 53]. In academia, the situation seems even more dire. As Hanenberg [27] notes in his essay on programming languages research: "developers, which are the main audience for new language constructs, are hardly considered in the research process." And Danas et al. [19] note that in some cases, the output of program analysis tools, such as in model-finders and SAT-solvers, are generated arbitrarily and in an unprincipled way, without regard to the friendliness towards the programmer who might actually use them.

In prior work [8], we have modeled the interaction of programmers with their program analysis tools in terms of an interaction framework, conceptualized by Abowd and Beale [1] and adapted to tools by Traver [59] (Figure 1). The

PLATEAU'17 Workshop on Evaluation and Usability of Programming Languages and Tools, October 23, 2017, Vancouver, CA

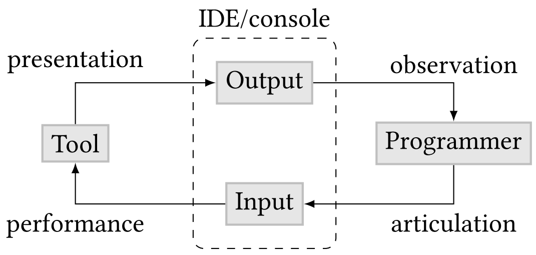

Figure 1. The interaction framework.

interaction framework describes the different interactions between the tool and the programmer, with the tool performing some sophisticated analysis, presenting the information to the programmer for observation through a console or IDE, and then allowing the programmer to articulate their intentions back to the tool. In this paper, we are interested specifically in the *presentation* aspect of the framework, and what we know about presenting human-friendly output from program analysis tools.

Towards the longer-term goal of providing a comprehensive knowledge synthesis about program analysis output, we conducted an interim scoping review of the proceedings from Programming Language Design and Implementation (PLDI), from 1988-2017. The scoping review is intended to be accessible to human-computer interaction (HCI) researchers who want to understand how the PL community is currently applying program analysis output, in order to eventually bridge HCI research with program analysis tools. Consequently, while PLDI papers are typically written to emphasize the formal properties of their program analysis tools as their primary goal, our scoping review reframes these papers in terms of the their *program analysis output* as the primary investigation.

The contributions of this scoping review are:

- A *quasi-gold set* of manually-identified papers from PLDI that relate to program analysis output, to bootstrap future, comprehensive literature reviews on the subject of human-friendly program analysis output.
- A knowledge synthesis of the features of program analysis output that researchers employ to present output to programmers, instantiated as a taxonomy (Section 3). Our taxonomy is agnostic to a particular mode of output, such as text or graphics.

2 Methodology

2.1 What is a Scoping Review?

In this study, we conduct a *scoping review*—a reduced form of the traditional systematic literature review [5, 35]. Scoping reviews have many of the same characteristics of traditional literature reviews: their purpose is to collect, evaluate, and present the available research evidence for a particular investigation. However, because of their reduced form, they can also be executed more rapidly than traditional literature reviews [52]. For example, reductions to scoping reviews include limiting the types of literature databases, constraining the date range under investigation, or eliding consistency measures such as inter-rater agreement. A notable weakness of scoping reviews is that they are not a final output; instead, they provide interim guidance towards what to expect if a comprehensive literature review were to be conducted. Scoping reviews are particularly useful in this interim stage for soliciting guidance on conducting a more formal review, as is our intention in this paper.

2.2 Execution of SALSA Framework

We conducted our scoping review using the traditional SALSA framework: *S*earch, *A*ppraisa*L*, *S*ynthesis, and *A*nalysis. Here, we discuss the additional constraints we adopted in using SALSA for our scoping review.

Search. We scoped our search to all papers within a single conference: Programming Language Design and Implementation (PLDI), for all years (1988-2017). As HCI researchers, we selected PLDI because it is considered to be a top-tier conference for programming languages research, because it contains a variety of program analysis tools, and because these tools tend to have formal properties of soundness and completeness that are not typically found in prototype tools within HCI. Discussions with other researchers within PLDI also revealed that researchers are interested in having their tools adopted by a broader community, but confusing program analysis output hinders usability of the tools to users outside their own research groups.

Appraisal. We manually identified papers through multiple passes. In the first pass, we skimmed titles and abstracts and included any papers which mentioned a program analysis tool and indicated output intended to be consumed by a programmer other than the authors of the tool. In this pass, our goal was to be liberal with paper inclusion, and to minimize false negatives. We interpreted program analysis tools in the broadest sense, to include model checkers, verifiers, static analysis tools, and dynamic analysis tools. In the second pass, we examined the contents of the paper to identify if the paper contributed or discussed its output. Finally, we removed papers that were purely related to reducing false positives, unless those papers used false positives as part of their output to provide additional information to the programmer. For some papers, the output was measured in terms of manual patches submitted to bug repositories. We excluded such papers since the output was manually constructed, and not obtained directly from the tool.

Table 1. Taxonomy of Presentation

Feature	Section
Alignment	Section 3.1
Clustering and Classification	Section 3.2
Comparing	Section 3.3
Example	Section 3.4
Interactivity	Section 3.5
Localizing	Section 3.6
Ranking	Section 3.7
Reducing	Section 3.8
Tracing	Section 3.9

Synthesis and Analysis. We synthesized the papers into a taxonomy of presentation features (Section 3). For analysis, we opted for a narrative-commentary approach [24] in which we summarized the contributions of each of the papers with respect to human-friendly presentations.

2.3 Limitations

As a form of interim guidance, a scoping review has several important limitations. First, the review is biased in several ways. Being scoped only to PLDI means that the identified taxonomy is likely to be incomplete. Second, the scoping review by definition misses key contributions found in other conferences, such as the International Conference on Software Engineering (ICSE), Foundations of Software Engineering (FSE), and the Conference on Human Factors in Computing Systems (CHI), just to name a few. Third, the paper summaries are intended to be accessible to HCI researchers who may not have formal PL experience. As a result, in the interest of being broadly accessible, some of the summaries of the papers may be oversimplified in terms of their PL contributions. Finally, any conclusions made from this interim work should be treated as provisional and subject to revision as more comprehensive reviews are conducted.

3 Taxonomy of Presentation

In this section, we classify and summarize all of the papers from PLDI from 1988-2017 that discuss or contribute to program analysis output intended for programmers. Intentionally, we labeled the taxonomy features such that they do not commit to a particular textual or visual affordance. For example, in a text interface, the feature of ranking (Section 3.7) may be implemented as an enumerated list of items in the console, with a prompt for selection if interactivity is required. In a graphical interface, ranking might instead be implemented using a drop-down, through which the programmer would select their desired option.

The identified taxonomy of presentation features are summarized in Table 1. Some papers describe output that use multiple features; in these cases, we selected the feature which we felt best represented the contribution of the output.

3.1 Alignment

In alignment, program analysis output is presented in a representation that is already familiar to the programmer.

Within this feature, Pombrio and Krishnamurthi [48] tackle the problem of syntactic sugar: programming constructs that make things easier to express, but are ultimately reducible to alternative constructs. For example, in C, the array access notation `a[i]` is syntactic sugar for (the sometimes less convenient notation) `*(a + i)`. Unfortunately, syntactic sugar is eliminated by many transformation algorithms, making the resulting program unfamiliar to the programmer. Pombrio and Krishnamurthi [48] introduce a process of *resugaring* to allow computation reductions in terms of the surface syntax. With similar aims, the AutoCorres tool uses a technique of *specification abstraction*, to present programmers with a representation of the program at a human-readable abstraction while additionally producing a formal refinement of the final presentation [25].

Notions of natural language and readability find their place in several PLDI papers. Qiu et al. [49] propose *natural proofs*, in which automated reasoning systems restrict themselves to using common patterns found in human proofs. Given a reference implementation, and an error model of potential corrections, Singh et al. [55] introduce a method for automatically deriving *minimal corrections* to students' incorrect solutions, in the form of a itemized list of changes, expressed in natural language. And the AFix tool uses a variety of static analysis and static code transformations to design bug fixes for a type of concurrency bug, *single-variable atomicity violations* [31]. The bug fixes are human-friendly in that they attempt to provide a fix that, in addition to other metrics, does not harm code *readability*. To support readability, the authors manually evaluated several possible locking policies to determine which ones were most readable.

Issues of alignment and representation become important to programmers during understanding of optimizations in source-level debugging of optimized code [2]; in their approach, Adl-Tabatabai and Gross [2] detect *engendered variables* that would cause the programmer to draw incorrect conclusions as a result of internal optimizations by the compiler. Earlier work by Brooks et al. [11] and Coutant et al. [18] also provide techniques that allow programmers to reason about optimized code through mapping the state of the optimized execution back to the original source. For example, Brooks et al. [11] use highlighting, boxing, reverse video, grey-scale shading, boxing, and underlining to animate and convey runtime program behavior, overlaid on the original source code, to the programmer.

3.2 Clustering and Classification

Clustering and classification output aims to organize or separate information in a way that reduces the cognitive burden for programmers. For example, Narayanasamy et al. [43] focus on a dynamic analysis technique to automatically classify *data races*—a type of concurrency bug in multi-threaded programs—as being potentially benign or potentially harmful. Furthermore, the tool provides the programmer with a reproducible scenario of the data race to help the programmer understand how it manifests.

Liblit et al. [39] present a statistical dynamic debugging technique that *isolates* bugs in programs containing multiple undiagnosed bugs; importantly, the algorithm separates the effects of different bugs and identifies predictors that are associated with individual bugs. An earlier technique using statistical sampling is also presented by the authors [38]. Ha et al. [26] introduce a classification technique in CLARIFY, a system which classifies *behavior profiles*—essentially, an application's behavior—for black box software components where the source code is not available. And Ammons et al. [3] consider the problem of specifications on programs in that the specifications themselves need methods for debugging; they present a method for debugging formal, temporal specifications through *concept analysis* to automatically group traces into highly similar clusters.

3.3 Comparing

Comparisons occur in program analysis tools when the programmer has a need to examine or understand differences between two or more versions of their code. Within this feature, Hoffman et al. [28] introduce a technique of *semantic views* of program executions to perform trace analysis; they apply their technique to identify regressions in large software applications. Through a differencing technique, their RPRISM tool outputs a semantic "diff" between the original and new versions, to allow potential causes to be viewed in their full context. Similarly, early work by Horwitz [29] identifies both semantic and textual differences between two versions of a program, in contrast to traditional diff-tools that treat source as plain text.

3.4 Example

Examples and counterexamples are forms of output that provide evidence for why a situation can occur or how a situation can be violated. Examples are usually provided in conjunction with other presentation features.

The Alive-Infer tool infers preconditions to ensure the validity of a peephole compiler optimization [42]. To the user, it reports both a *weakest* precondition and a set of "succinct" partial preconditions. For wrong optimizations, the tool provides counterexamples. Zhang et al. [64] apply a technique of *skeletal program enumeration* to generate small test programs for reporting bugs about in GCC and clang compilers; the generated test programs contain fewer than 30 lines on average. Still other work with test programs devise a test-case reducer for C compiler bugs to obtain small and valid test-cases consistently [50]; the underlying machinery is based on *generic fixpoint computations* which invokes a *modular reducer*.

Padon et al. [46] hypothesize that one of the reasons automated methods are difficult to use in practice is because they are opaque. As Padon et al. [46] state, "they fail in ways that are difficult for a human user to understand and to remedy." Their system, Ivy, graphically displays concrete counterexamples to induction, and allows the user to interactively guide generation from these counterexamples. Nguyễn and Van Horn [44] implement a tool in Racket to generate counterexamples for erroneous modules and Isradisaikul and Myers [30] design an algorithm that generates helpful counterexamples for parsing ambiguities; for every parsing conflict, the algorithm generates a compact counterexample illustrating the ambiguity.

PSketch is a program synthesis tool that helps programmers implement concurrent data structures; it uses a *counter example guided inductive synthesis algorithm* (CEGIS) to converge to a solution within a handful of iterations [56]. Given a partial program example, or a *sketch*, PSketch outputs a completed sketch that matches a given correctness criteria.

For type error messages, Lerner et al. [36] pursue an approach in which the type-checker itself does not produce error messages, but instead relies on an oracle for a search procedure that finds similar programs that *do* type-check; to bypass the typically-inscrutable type error messages, their system provides examples of code (at the same location) that would type check.

And for memory-related output, Cherem et al. [15] implement an analysis algorithm for detecting memory leaks in C programs; their analysis uses *sparse value-flows* to present *concise* error messages containing only a few relevant assignments and path conditions that cause the error to happen.

3.5 Interactivity

We identified several papers whose tools support interactivity. That is, the programmer can interact with the tool either before the output is produced, in order to customize the output—or work with the output of the tool in a *mixed-initiative* fashion, where both the programmer and the tool collaborate to arrive at a solution.

Within this feature, Parsify is a program synthesis tool that synthesizes a parser from input and output examples. The tool interface provides immediate visual feedback in response to changes in the grammar being refined, as well as a graphical mechanism for specifying example parse trees using only textual selections [37]. As the programmer adds production rules to the grammar, Parsify uses colored regions overlaid on the examples to convey progress to the programmer.

Live programming is a user interface capability that allows a programmer to edit code and immediately see the effect of the code changes. Burckhardt et al. [13] introduce a *type and effect* formalization that separates the *rendering* of UI components as a side effect of the *non-rendering* logic of the program. This formalization enables responsive feedback and allows the programmer to make code changes without needing to restart the debugging process to refresh the display.

Dillig et al. [20] present a technique called *abductive inference*—that is, to find an explanatory hypothesis for a desired outcome—to assist programmers in classifying error reports. The technique computes small, relevant queries presented to a user that capture exactly the information the analysis is missing to either discharge or validate the error.

LeakChaser identifies unnecessarily-held memory references which often result in memory leaks and performance issues in manages languages such as Java [62]. The tool allows an *iterative* process through three *tiers* which assist programmers at different levels of abstraction, from *transactions* at the highest-level tier to *lifetime relationships* at the lowest level tier.

Chameleon assists programmers in choosing an abstract collection implementation in their algorithm [54]. During program execution, Chameleon computes trace metrics using *semantic profiling*, together with a set of collection selection rules, to present recommended collection adaptation strategies to the programmers. Similarly, the PetaBricks tool makes algorithm choice a first-class construct of the language [4].

von Dincklage and Diwan [61] identify how tools can benefit from guidance from the programmer in cases where incorrect tool results would otherwise compromise its usefulness. For example, many refactoring operations in the Eclipse IDE are optimistic, and do not fully check that the result is fully legal. They propose a method to produce necessary and sufficient reasons, that is, a *why* explanation, for a potentially undesirable result; the programmer can then—through applying predicates—provide feedback on whether the given analysis result is desirable.

Finally, MrSpidey is a user-friendly, static debugger for Scheme [22]; the program analysis computes *value set descriptions* for each term in the program and constructs a *value flow graph* connecting the set descriptions; these flows are made visible to the programmer through a value flow browser which overlays arrows over the program text. The programmer can interactively expose portions of the value graph.

3.6 Localizing

Tools present the relevant program locations for an error, or *localize* errors, through two forms: 1) a *point* localization, in which a program analysis tool tries to identify a single region or line as relevant to the error, and 2) as *slices*, where multiple regions are responsible for the error.

Point. Zhang et al. [63] implement, within the GHC compiler, a simple Bayesian type error diagnostic that identifies the *most likely* source of the type error, rather than the *first source* the inference engine "trips over." The BugAssist tool implements an algorithm for error cause localization based on a reduction to *the maximal satisfiability problem* to identify the *cause* of an error from failing execution traces [34]. The Breadcrumbs tool uses a *probabilistic calling context* (essentially, a stack trace) to identify the root cause of bug, by recording extra information that *might* be useful in explaining a failure [10].

Slices. Program slicing identifies parts of the program that may affect a point of interest—such as those related to an error message; Sridharan et al. [57] propose a technique called *thin slicing* which helps programmers better identify bugs because it identifies more relevant lines of code than traditional slicing. Analogous to thin slicing, Zhang et al. [65] developed a strategy for pruning dynamic slices to identify subsets of statements that are likely responsible for producing an incorrect value; for each statement executed in the dynamic slice, their tool computes a confidence value, with higher values corresponding to greater likelihood that the execution of the statement produced a correct value.

3.7 Ranking

Ranking is a presentation feature that orders the output of the program analysis in a systematic way. For example, random testing tools, that is, *fuzzers*, can be frustrating to use because they "indiscriminately and repeatedly find bugs that may not be severe enough to fix right away" [14]. Chen et al. [14] propose a technique that *orders* test cases in a way that diverse, interesting cases (defined through a machine technique called *furthest point first*) are highly ranked. And the AcSpec tool prioritizes alarms for automatic program verifiers through *semantic inconsistency detection* in order to report high-confidence warnings to the programmer [9].

Coppa et al. [17] present a profiling methodology and toolkit for helping programmers discover asymptotic inefficiencies in their code. The output of the profiler is, for each executed routine of the program, a set of tuples that aggregate performance costs by input size—these outputs are intended to be used as input to performance plots. The Kremlin tool makes recommendations about which parts of the program a programmer should spend effort parallelizing; the tool identifies these regions through a *hierarchical critical path analysis* and presents to the programmer an ordered (by speedup) parallelism plan as a list of files and lines to modify [23].

Perelman et al. [47] provide ranked expressions for completions in API libraries through a language of *partial expressions*, which allows the programmer to leave "holes" for the parts they do not know.

3.8 Reduction

Reduction approaches take a large design space of allowable program output and reduce that space using some systematic rule. Within this feature, Logozzo et al. introduce a static analysis technique of *Verification Modulo Versions* (VMV), which reduces the number of alarms reported by verifiers while maintaining semantic guarantees [41]. Specifically, VMV is designed for scenarios in which programmers desire to fix *new* defects introduced since a previous release.

3.9 Tracing

Tracing is a form of slicing that involves flows of information, and understanding how information propagates across source code. As one example, Ohmann et al. [45] present a system that answers control-flow queries posed by programmers as formal languages. The tool indicates whether the query expresses control flow that is *possible* or *impossible* for a given failure report. As another example, PIDGIN is a program analysis and understanding tool that allows programmers to interactively explore *information flows*—through *program dependence graphs* within their applications—and investigate counterexamples [32]. Taint analysis is another information-flow analysis that establishes whether values from unstructured parameters may flow into security-sensitive operations [60]; implemented as TAJ, the tool additionally eliminates redundant reports through hybrid thin slicing and *remediation logic* over library local points. Other techniques, such as those by Rubio-González et al. [51], use data-flow analysis techniques to track errors as they propagate through file system code.

To support algorithmic debugging, Faddegon and Chitil [21] developed a library in Haskell, that, after annotating suspected functions, presents a detailed *computational tree*. Computational trees are essentially a trace to help programmers understand how a program works or why it does not work. The tool TraceBack provides debugging information for production systems by providing execution history data about program problems [6]; it uses *first-fault diagnosis* to discover what went wrong the *first* time the fault is encountered.

MemSAT helps programmers debug and reason about memory models: given an *axiomatic* specification, the tool outputs a *trace*—sequences of reads and writes—of the program in which the specification is satisfied, or a *minimal* subset of the memory model and program constraints that are unsatisfiable [58].

The Merlin security analysis tool infers *information flows* in a program to identify security vulnerabilities, such as cross-site scripting and SQL inject attacks [40]. Internally, the inference is based on modeling a *data propagation graph* using *probabilistic constraints.*

4 Discussion

Lack of user evaluations in PL. Although we identified and classified papers within PLDI in terms of a taxonomy of presentation, our investigation confirms that papers either perform no usability evaluation with programmers, or the claims of usability of the tool are made through intuition—using the authors of the paper as subjects. For example, consider the presentation feature of alignment (Section 3.1), in which several assumptions are made about how output should be presented in familiar representations to the programmer. All of these assumptions appear to be intuitive—give output in the same level of syntactic sugar as their source code for consistency, use proof constructions commonly found in human proofs, and support readability. Unfortunately, none of these assumptions are tested with actual programmers, reminding us of the concerns noted by Hanenberg and others in the introduction. It seems likely that some of these assumptions *are* actually incorrect, which may explain the lack of adoption in practice and the confusing tool output programmers report for many of these sophisticated program analysis tools.

Lack of operational tools in HCI. At the same time, HCI researchers perform usability studies on user interfaces, yet the experiments they conduct are understandably evaluated against representative tool experiences, rather than the multiplicity of corner cases that occur in practice. Consequently, even if the user interfaces are found to be effective or usable for some measures, the tools themselves cannot actually be used in practice. Regrettably, this means that user interface advances remain within academic papers, and do not ever make it to actual programmers without significant investment in tools that may not even be possible to build due to fundamental, technical limitations.

Bridging PL and HCI. In our view, both deficiencies in PL and HCI can be reduced by fostering collaborations between the disciplines. A cross-disciplinary approach to tool development would enable usable program analysis tools, by having a pipeline from program analysis tools to user evaluations in HCI. HCI contributions could then feedback to PL to further improve the output of program analysis tools. But doing so requires a cross-disciplinary community that can provide such opportunities for collaboration. We suggest that PLATEAU has the potential to become this community.

5 Conclusions

In this paper, we conducted a scoping review of PLDI from the period of 1988-2017. In the review, we identified and cataloged papers for program analysis tools that discussed or made contributions to the presentation of output towards programmers. Admittedly, a scoping review is only a starting point for investigation, and can only provide interim guidance. Nevertheless, our hope is that the scoping review we have conducted can serve to bootstrap future, comprehensive systematic literature reviews. We are open to feedback on practical methods to realizing that goal.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1714538.

References

[1] Gregory D Abowd and Russell Beale. 1991. Users, systems and interfaces: A unifying framework for interaction. In *People and Computers VI*. 73–87.

[2] Ali-Reza Adl-Tabatabai and Thomas Gross. 1996. Source-level debugging of scalar optimized code. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation (PLDI '96)*. ACM, 33–43. https://doi.org/10.1145/231379.231388

[3] Glenn Ammons, David Mandelin, Rastislav Bodík, and James R. Larus. 2003. Debugging temporal specifications with concept analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. ACM, 182–195. https://doi.org/10.1145/781131.781152

[4] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. PetaBricks: A language and compiler for algorithmic choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, 38–49. https://doi.org/10.1145/1542476.1542481

[5] H Arksey and L O'Malley. 2005. Scoping studies: Towards a methodological framework. *Int J Soc Res Methodol* 8 (2005).

[6] Andrew Ayers, Richard Schooler, Chris Metcalf, Anant Agarwal, Junghwan Rhee, and Emmett Witchel. 2005. TraceBack: First fault diagnosis by reconstruction of distributed control flow. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, 201–212. https://doi.org/10.1145/1065010.1065035

[7] Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. 2017. Do developers read compiler error messages?. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 575–585. https://doi.org/10.1109/ICSE.2017.59

[8] Titus Barik, Jim Witschey, Brittany Johnson, and Emerson Murphy-Hill. 2014. Compiler error notifications revisited: An interaction-first approach for helping developers more effectively comprehend and resolve error notifications. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. ACM, 536–539. https://doi.org/10.1145/2591062.2591124

[9] Sam Blackshear and Shuvendu K. Lahiri. 2013. Almost-correct specifications: A modular semantic framework for assigning confidence to warnings. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, 209–218. https://doi.org/10.1145/2491956.2462188

[10] Michael D. Bond, Graham Z. Baker, and Samuel Z. Guyer. 2010. Breadcrumbs: Efficient context sensitivity for dynamic bug detection analyses. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, 13–24. https://doi.org/10.1145/1806596.1806599

[11] Gary Brooks, Gilbert J. Hansen, and Steve Simmons. 1992. A new approach to debugging optimized code. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92)*. ACM, 1–11. https://doi.org/10.1145/143095.143108

[12] P. J. Brown. 1983. Error messages: The neglected area of the man/machine interface. *Commun. ACM* 26, 4 (April 1983), 246–249. https://doi.org/10.1145/2163.358083

[13] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. 2013. It's alive! Continuous feedback in UI programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, 95–104. https://doi.org/10.1145/2491956.2462170

[14] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, 197–208. https://doi.org/10.1145/2491956.2462173

[15] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, 480–491. https://doi.org/10.1145/1250734.1250789

[16] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*. ACM Press, 332–343. https://doi.org/10.1145/2970276.2970347

[17] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. 2012. Input-sensitive profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, 89–98. https://doi.org/10.1145/2254064.2254076

[18] D. S. Coutant, S. Meloy, and M. Ruscetta. 1988. DOC: A practical approach to source-level debugging of globally optimized code. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI '88)*. ACM, 125–134. https://doi.org/10.1145/53990.54003

[19] Natasha Danas, Tim Nelson, Lane Harrison, Shriram Krishnamurthi, and Daniel J Dougherty. 2017. User studies of principled model finder output. In *International Conference on Software Engineering and Formal Methods*. Springer, 168–184. https://doi.org/10.1007/978-3-319-66197-1_11

[20] Isil Dillig, Thomas Dillig, and Alex Aiken. 2012. Automated error diagnosis using abductive inference. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, 181–192. https://doi.org/10.1145/2254064.2254087

[21] Maarten Faddegon and Olaf Chitil. 2016. Lightweight computation tree tracing for lazy functional languages. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, 114–128. https://doi.org/10.1145/2908080.2908104

[22] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. 1996. Catching bugs in the web of program invariants. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation (PLDI '96)*. ACM, 23–32. https://doi.org/10.1145/231379.231387

[23] Saturnino Garcia, Donghwan Jeon, Christopher M. Louie, and Michael Bedford Taylor. 2011. Kremlin: Rethinking and rebooting gprof for the multicore age. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, 458–469. https://doi.org/10.1145/1993498.1993553

[24] Maria J. Grant and Andrew Booth. 2009. A typology of reviews: An analysis of 14 review types and associated methodologies. *Health Information & Libraries Journal* 26, 2 (June 2009), 91–108. https://doi.org/10.1111/j.1471-1842.2009.00848.x

[25] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. 2014. Don't sweat the small stuff: Formal verification of C code without the pain. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, 429–439. https://doi.org/10.1145/2594291.2594296

[26] Jungwoo Ha, Christopher J. Rossbach, Jason V. Davis, Indrajit Roy, Hany E. Ramadan, Donald E. Porter, David L. Chen, and Emmett Witchel. 2007. Improved error reporting for software that uses black-box components. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, 101–111. https://doi.org/10.1145/1250734.1250747

[27] Stefan Hanenberg. 2010. Faith, hope, and love: An essay on software science's neglect of human factors. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, 933–946. https://doi.org/10.1145/1869459.1869536

[28] Kevin J. Hoffman, Patrick Eugster, and Suresh Jagannathan. 2009. Semantics-aware trace analysis. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, 453–464. https://doi.org/10.1145/1542476.1542527

[29] Susan Horwitz. 1990. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*. ACM, 234–245. https://doi.org/10.1145/93542.93574

[30] Chinawat Isradisaikul and Andrew C. Myers. 2015. Finding counterexamples from parsing conflicts. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, 555–564. https://doi.org/10.1145/2737924.2737961

[31] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, 389–400. https://doi.org/10.1145/1993498.1993544

[32] Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. 2015. Exploring and enforcing security guarantees via program dependence graphs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, 291–302. https://doi.org/10.1145/2737924.2737957

[33] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 672–681. https://doi.org/10.1109/ICSE.2013.6606613

[34] Manu Jose and Rupak Majumdar. 2011. Cause clue clauses: Error localization using maximum satisfiability. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, 437–446. https://doi.org/10.1145/1993498.1993550

[35] Barbara Kitchenham. 2004. Procedures for performing systematic reviews. *Keele, UK, Keele University* 33, 2004 (2004), 1–26.

[36] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. 2007. Searching for type-error messages. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, 425–434. https://doi.org/10.1145/1250734.1250783

[37] Alan Leung, John Sarracino, and Sorin Lerner. 2015. Interactive parser synthesis by example. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, 565–574. https://doi.org/10.1145/2737924.2738002

[38] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. 2003. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. ACM, 141–154. https://doi.org/10.1145/781131.781148

[39] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, 15–26. https://doi.org/10.1145/

1065010.1065014

[40] Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. 2009. Merlin: Specification inference for explicit information flow problems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, 75–86. https://doi.org/10.1145/1542476.1542485

[41] Francesco Logozzo, Shuvendu K. Lahiri, Manuel Fähndrich, and Sam Blackshear. 2014. Verification modulo versions: Towards usable verification. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, 294–304. https://doi.org/10.1145/2594291.2594326

[42] David Menendez and Santosh Nagarakatte. 2017. Alive-Infer: Data-driven precondition inference for peephole optimizations in LLVM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)*. ACM, 49–63. https://doi.org/10.1145/3062341.3062372

[43] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. 2007. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, 22–31. https://doi.org/10.1145/1250734.1250738

[44] Phúc C. Nguyen and David Van Horn. 2015. Relatively complete counterexamples for higher-order programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, 446–456. https://doi.org/10.1145/2737924.2737971

[45] Peter Ohmann, Alexander Brooks, Loris D'Antoni, and Ben Liblit. 2017. Control-flow recovery from partial failure reports. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)*. ACM, 390–405. https://doi.org/10.1145/3062341.3062368

[46] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: Safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, 614–630. https://doi.org/10.1145/2908080.2908118

[47] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. 2012. Type-directed completion of partial expressions. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, 275–286. https://doi.org/10.1145/2254064.2254098

[48] Justin Pombrio and Shriram Krishnamurthi. 2014. Resugaring: Lifting evaluation sequences through syntactic sugar. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, 361–371. https://doi.org/10.1145/2594291.2594319

[49] Xiaokang Qiu, Pranav Garg, Andrei Ştefănescu, and Parthasarathy Madhusudan. 2013. Natural proofs for structure, data, and separation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, 231–242. https://doi.org/10.1145/2491956.2462169

[50] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, 335–346. https://doi.org/10.1145/2254064.2254104

[51] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. 2009. Error propagation analysis for file systems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, 270–280. https://doi.org/10.1145/1542476.1542506

[52] Holger J Schünemann and Lorenzo Moja. 2015. Reviews: Rapid! Rapid! Rapid! ...and systematic. *Systematic Reviews* 4, 1 (2015), 4. https://doi.org/10.1186/2046-4053-4-4

[53] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' build errors: A case study (at Google). In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*. ACM Press, 724–734. https://doi.org/10.1145/2568225.2568255

[54] Ohad Shacham, Martin Vechev, and Eran Yahav. 2009. Chameleon: Adaptive selection of collections. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, 408–418. https://doi.org/10.1145/1542476.1542522

[55] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. *SIGPLAN Not.* 48, 6 (June 2013), 15–26. https://doi.org/10.1145/2499370.2462195

[56] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. 2008. Sketching concurrent data structures. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, 136–148. https://doi.org/10.1145/1375581.1375599

[57] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. 2007. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, 112–122. https://doi.org/10.1145/1250734.1250748

[58] Emina Torlak, Mandana Vaziri, and Julian Dolby. 2010. MemSAT: Checking axiomatic specifications of memory models. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, 341–350. https://doi.org/10.1145/1806596.1806635

[59] V. Javier Traver. 2010. On compiler error messages: What they say and what they mean. *Advances in Human-Computer Interaction* 2010 (2010), 1–26. https://doi.org/10.1155/2010/602570

[60] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: Effective taint analysis of web applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, 87–97. https://doi.org/10.1145/1542476.1542486

[61] Daniel von Dincklage and Amer Diwan. 2008. Explaining failures of program analyses. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, 260–269. https://doi.org/10.1145/1375581.1375614

[62] Guoqing Xu, Michael D. Bond, Feng Qin, and Atanas Rountev. 2011. LeakChaser: Helping programmers narrow down causes of memory leaks. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, 270–282. https://doi.org/10.1145/1993498.1993530

[63] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2015. Diagnosing type errors with class. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, 12–21. https://doi.org/10.1145/2737924.2738009

[64] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)*. ACM, 347–361. https://doi.org/10.1145/3062341.3062379

[65] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Pruning dynamic slices with confidence. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, 169–180. https://doi.org/10.1145/1133981.1134002