Triaging Checklists: a Substitute for a PhD in Static Analysis

Khoo Yit Phang   Jeffrey S. Foster   Michael Hicks   Vibha Sazawal

University of Maryland, College Park

{khooyp,jfoster,mwh,vibha}@cs.umd.edu

Abstract

Static analysis tools have achieved great success in recent years in automating the process of detecting defects in software. However, these sophisticated tools have yet to gain widespread adoption, since many of these tools remain too difficult to understand and use. In previous work, we discovered that even with an effective code visualization tool, users still found it hard to determine if warnings reported by these tools were true errors or false warnings. The fundamental problem users face is to understand enough of the underlying algorithm to determine if a warning is caused by imprecision in the algorithm, a challenge that even experts with PhDs may take a while to achieve. In our current work, we propose to use *triaging checklists* to provide users with systematic guidance to identify false warnings by taking into account specific sources of imprecision in the particular tool. Additionally, we plan to provide *checklist assistants*, which is a library of simple analyses designed to aid users in answering checklist questions.

1. Introduction

In recent years, the research and industrial communities have made great strides in developing sophisticated software defect detection tools based on *static analysis*. Such tools analyze program source code with respect to some explicit or implicit specification, and report potential errors in the program. Static analysis tools show great promise in automating defect detection: new analysis techniques and tools are now regularly reported in the research literature as having found bugs in significant open-source software [Ayewah et al. 2007; Engler and Ashcraft 2003; Engler et al. 2000; Foster et al. 2002; Hovemeyer and Pugh 2004; Naik et al. 2006; Shankar et al. 2001]. Microsoft routinely uses tools to find bugs in production software [CSE; Das 2006] and other large software houses, such as Google [Ayewah et al. 2007;

Ruthruff et al. 2008] and EBay [Jaspan et al. 2007], are beginning to follow suit.

Despite these successes, most static analysis tools remain limited in formal adoption, particularly tools that use sophisticated algorithms [Ayewah et al. 2008; Ayewah and Pugh 2008]. In our opinion, one of the key reasons is that many tool designers today fail to appreciate that the human user is an essential component of the defect detection process. A tool can output a list of possible errors, but the user has to determine if a reported warning is actually an error and, if so, how to fix it. In fact, we consider a tool to be effective only if it can successfully collaborate with the user to locate actual errors and fix them. To do so, we believe that tools must be able to convey their results to the user efficiently and with sufficient information for the user to correctly and quickly arrive at a conclusion.

In our previous work (Section 2), we developed a code visualization tool that is designed to efficiently explain the often long and complicated errors reported by static analysis tools. While this reduces the users' effort to understand error reports, we discovered that users face a more fundamental difficulty in understanding how false warnings may arise from specific sources of imprecision in static analysis algorithms. Training is an impractical solution to this problem; even static analysis experts such as ourselves find that it can often take some time to study a particular static analysis tool to truly appreciate and internalize all the intricacies in the underlying algorithm.

Instead, we believe that we can provide systematic guidance to the user in the form of *triaging checklists* (Section 3). Checklists are very practical devices to guide users in triaging, since users simply follow the instructions on the checklist to answer each question and to determine the conclusions. Checklists can also be very specific, since they can be designed by tool developers to point out known sources of imprecision in their tools and instruct users how to look for them. To be most effective, we want checklists to be customized to individual warnings such that users will only need to answer exactly the minimum number of questions to triage each warning. In this paper, we describe our ongoing efforts to explore how sources of imprecision may be traced through various static analysis algorithms, and how to con-
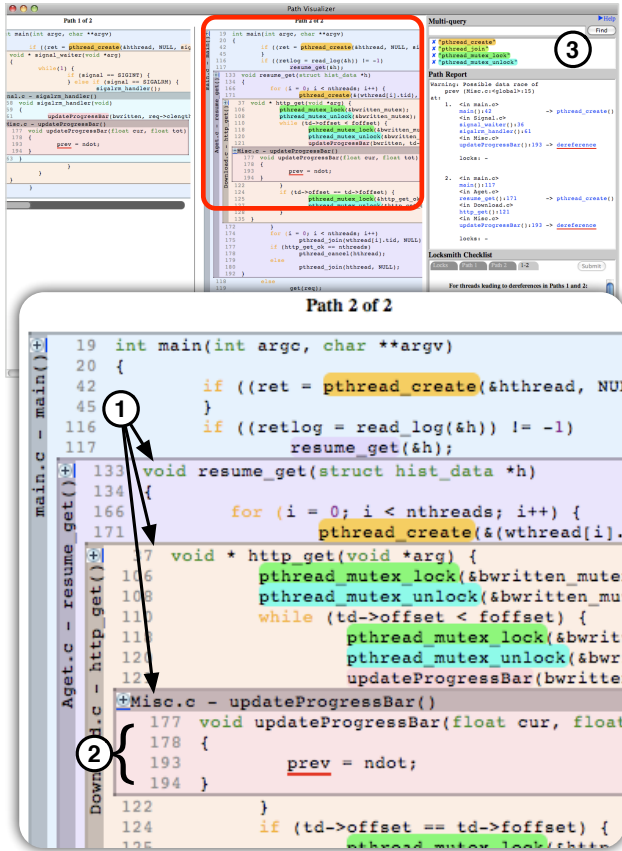
Figure 1: Path Projection (top) and a close-up which shows (1) function call inlining, (2) code folding, and (3) multiple keyword searches.

struct clear, easy-to-understand checklists with this information.

2. Previous Work: Visualizing Program Paths with *Path Projection*

In previous work, we developed *Path Projection*, a general user interface toolkit for visualizing and navigating *program paths* [Khoo et al. 2008a]. Program paths are lists of program statements, and are used by many static analysis tools to report potential errors. For example, CQual [Greenfieldboyce and Foster 2004] and MrSpidey [Flanagan et al. 1996] report paths corresponding to data flow; BLAST [Beyer et al. 2004] and SDV [Ball and Rajamani 2002] provide counterexample traces; Code Sonar [GrammaTech, Inc. 2007] provides a failing path with pre- and post-conditions; and Fortify SCA [Fortify Software Inc. 2007] provides control flow paths that could induce a failure.

However, manually tracing program paths to understand a warning is a tedious task: the user has to jump between different functions in different files and sift through many lines of source code, while trying to work out how relevant statements along the path may or may not lead to the error.

Most source code editors are inefficient for this task as they are designed to view one section of a file at a time, whereas a program path may span multiple functions across several files. Many editors provide hyperlinks for users to quickly navigate between different sections of the path, but even with hyperlinks, there is still a significant cognitive burden on the user to remember fragments of code for each path section, or alternatively, to manually arrange multiple editor windows to see all of the path at once.

With these issues in mind, we designed Path Projection to visualize program paths in a way that helps users see the entire path at once. Path Projection uses two main techniques—*function call inlining* and *code folding*—to "project" the source code onto the error path. An example is shown in Figure 1. In the shown path, main calls resume_get, and the body of resume_get is inlined directly below the call site (1). Then resume_get calls http_get (via pthread_create), so the latter's body is inlined, and so on. Inlining ensures the code is visually arranged in path order, which removes the need to jump around the program to trace a path. Code folding is used to hide away irrelevant code, indicated by discontinuous line numbers, so that the user is initially shown as much of the path as will fit in one screen (2). We show only lines that are implicated in the error report, and the function names or conditional guards of enclosing lexical blocks (including matching braces). The highlighted keywords pthread_create, pthread_mutex_lock, etc. would normally be folded, but are here revealed through multiple keyword searches (3).

We evaluated Path Projection's utility with a controlled experiment in which users triaged reports produced by Locksmith [Pratikakis et al. 2006], a static data race detection tool for C. Locksmith reports data races by listing the call stacks that access the shared variable for each conflicting thread, so, we use Path Projection to visualize the call stacks side-by-side. We measured users' completion time and accuracy in triaging Locksmith reports, comparing Path Projection to a "standard" viewer that we designed to include the textual error report along with commonly used IDE features. We found that Path Projection improved the time it takes to triage a bug by roughly one minute, an 18% improvement, and that accuracy remained the same. Moreover, participants reported they preferred Path Projection over the standard viewer. Users spent little time looking at the error report in the Path Projection interface, which suggests that Path Projection succeeds in making paths easy to see and understand in the source code view.

3. Current Work: Checklists for Triaging Static Analysis

We find from our experiments that, although a good visualization such as Path Projection reduces the effort to triage the result of static analysis tools, many users still find the triaging task to be very difficult. To triage a reported error,

For threads leading to dereferences in Paths i and j:

Are they parent-child (or child-parent), or child-child?

⚪ Parent-child / ⚪ Child-child

Parent-child (or child-parent) threads.	Y	N
Does the parent's dereference occur after the child is spawned?	⚪	⚪
Before its dereference, does the parent wait (via `pthread_join`) for the child?	⚪	⚪
If no, there is likely a race. Are there reasons to show otherwise?	⚪	⚪
Explain:		

Child-child threads.	Y	N
Are the children mutually exclusive (i.e., only one can be spawned by their common parent/ancestor)?	⚪	⚪
If no, there is likely a race. Are there reasons to show otherwise?	⚪	⚪
Explain:		

Figure 2: Checklist for triaging Locksmith reports

the user has to know enough about the analysis performed—in particular, its sources of imprecision—to determine if the error is a false positive. For example, a static analysis may be *path insensitive*, meaning it assumes that all conditional branches could be taken both ways. Thus the tool may falsely report errors on unrealizable paths. As another example, a static analysis tool may be *flow insensitive*, meaning it does not pay attention to statement ordering. Thus the tool may decide that some source data might reach a target location even if an intermediate assignment statement kills the flow of data, making this impossible at run time.

In our experience, reasoning about imprecision to detect false positives is out of reach for most users. We found that even with extensive tutorials, participants had trouble triaging the results from static analysis tools [Khoo et al. 2008b]. Their triaging procedures were usually ad hoc and inconsistent, often neglecting some sources of imprecision (and thus sometimes wrongly concluding a report to be a true bug) or assuming non-existent sources of imprecision (and therefore wasting time verifying conditions certain to hold).

3.1 Triaging checklists

We believe we can greatly improve the effectiveness of static analysis tools by providing users with checklists to guide them through the triaging process. A *triaging checklist* enumerates a series of questions that the user has to answer in order to triage a particular error. In our Path Projection study, we developed a partial checklist to help users triage error reports from Locksmith. One common source of false positives from Locksmith is its path insensitivity, so this checklist fo-

cuses on verifying the realizability of paths implicated in a data race.

A section of the checklist is shown in Figure 2. This section is shown when Locksmith reports that two threads, i and j, access a shared variable without holding a common lock, which would lead to a data race. The user has to examine the call stacks of the conflicting threads in the Locksmith report to determine if the variables may be accessed simultaneously from threads i and j.

The user first has to decide whether threads i and j are in a parent-child or other (child-child) relationship. If the user selects parent-child, the user then needs to determine if the dereference in the parent occurs after the child thread is created (otherwise there is no race) and if there are no blocking thread joins preventing the parent from dereferencing the shared variable until the child has joined (Locksmith ignores calls to `pthread_join`). The child-child case is analogous—the user must check whether the two children are mutually exclusive (e.g., spawned in disjoint branches of an **if** statement), which would preclude a race. For both parent-child and child-child races there is a catch-all checklist question for other reasons that could preclude the race, e.g., due to branching logic along the given path.

In our final Path Projection study (described in Section 2), we provided our users with the same checklists for both user interface conditions. Prior to that study, we ran a pilot study without checklists, and found that users took much longer to complete the given tasks. Although the two studies are not strictly comparable, we observed that users triaged error reports roughly 40% (or four minutes) faster with checklists than without them, and their conclusions were more reliable. A key reason for improved accuracy is that checklists make clear exactly what users need to look for, so they can be systematic and not miss important indicators. The reason for improved efficiency is that the checklist enumerates exactly what must be done, and no more. For example, the Locksmith checklist has no mention of verifying whether a listed lock is actually held—in this case Locksmith's algorithm is perfectly precise, so its conclusions are trustworthy. But in our earlier pilot study, many users would get distracted examining if locks were or were not held.

We think there is much promise in checklists, so we plan to study their use more clearly and systematically. First, we are working to generalize the use of checklists to other tools and other types of imprecision in static analysis. For example, Locksmith is also imprecise because it uses a flow-insensitive alias analysis, which means that while paths to dereferences in different threads may be simultaneously realizable, the dereferences may actually be to different memory locations (and thus not a race), contrary to what the alias analysis thinks.

Second, we would like to build static analyses that efficiently track sources of imprecision, and use this information to construct checklists that are specific to each reported error.

For example, for the statement **if** $(\times)$ p = q, we know p and q are aliases only if $\times$ is non-zero, but a *path-insensitive* alias analysis would conservatively ignore the conditional and simply assume, unconditionally, that p and q are aliases. If this assumption is used to generate an error report, the report will be a false positive if $\times$ is always zero. Thus, the analysis should keep track of when it takes this imprecise step. If the assumption leads to an error, a checklist item can be constructed to ask the user to check whether $\times$ may indeed be non-zero. This basic idea is similar to client-driven pointer analysis [Guyer and Lin 2003], which attempts to selectively remedy the imprecision of its pointer analysis based on feedback from subsequent client static analyses. While useful, automated remedies are not always possible, nor can they always be identified cheaply or reliably. Checklists take advantage of the human's expertise and computational ability to verify well-defined problems that may have no satisfactory automated solution.

Finally, we plan to measure the efficacy of checklists and checklist assistants through controlled user studies for bug triage, of the flavor of the one used to evaluate Locksmith's existing checklist [Khoo et al. 2008b]. As we gain further insight and experience in developing checklists, we will move to automate the generation of tool-specific checklists as well, drawing on the basic theory of abstract interpretation (which expresses the ways in which an analysis domain is conservative) [Cousot and Cousot 1977]. We will also consider means to allow users to construct their own checklists that can take advantage of accumulated analysis information.

3.2 Checklist assistants

We are also investigating the use of *checklist assistants*, which are simple analyses to help answer specific questions in triaging checklists. Unlike the core analyses of tools, these simple analyses need not be sound; they will simply point the user in the right direction, ultimately relying on his/her judgment. For example, our Path Projection interface contains a rudimentary assistant in the form of a multi-keyword search that highlights and reveals matching text even if they had been hidden by code folding. Consider the Locksmith checklist again: the user is asked in one case to check if a parent joins a child thread before an access to a common shared variable; if so, there is no race, since at that point the child thread has exited. To assist in this task, the user may enter " pthread_join " into the multi-keyword search to highlight all matching occurrences of that text in the displayed path. The user can then visually scan for a matching occurrence of pthread_join between the accesses in the parent and child threads. If there is a such occurrence, the user can quickly determine that there is no race. A more sophisticated assistant may recursively search all functions called between the parent and child threads for occurrences of pthread_join , further simplifying the user's effort to answer the checklist.

Rather than "baking in" these sorts of analyses into a given tool, we are looking into providing a generic library of checklist assistants that can be reused across different types of static analysis. These may be developed in the style of ASTLog (later, PREFast) [Crew 1997], for simple syntactic queries, or a more general data flow analysis framework parameterizable by the lattice, transfer functions, and so on [Chambers et al. 1996; Duesterwald et al. 1997; Dwyer and Clarke 1996; Hall et al. 1993]. Ideally, these checklist assistants should have access to the internal results of the tools' core analyses (e.g., the control flow graph, points-to graph, etc.); however, we are also exploring the possibility of working with just the information available from the tools' error reports, to make checklist assistants applicable to any tool. We also imagine allowing users to indicate that a heuristic analysis be used automatically, once it becomes sufficiently trusted.

4. Related Work

Checklists have attained widespread adoption in a variety of fields [Hales and Pronovost 2006], including emergency room triage [Berman et al. 1989], aviation [Degani and Wiener 1990], and ergonomics [Brodie and Wells 1997]. In software engineering, checklists play an important role in software inspection tasks. Anderson et al. [2003] demonstrate how CodeSurfer can be used to answer questions in NASA's Code Inspection Checklist. Ayewah and Pugh [2009] developed a checklist for Findbugs to help users rate the severity of reported warnings. The successful adoption of checklists in many fields gives us confidence that we can greatly improve the usability of static analysis tools by giving users checklists.

Several tools exist to query code facts, such as Ciao [Chen et al. 1995], JQuery [Janzen and Volder 2003], and Semmle-Code [Semmle Limited]. Lencevicius et al. [2003] propose using querying for interactive debugging, and Ko and Myers [2008] built a debugger called Whyline that allows programmers to ask "why" and "why not" questions about a program trace. Partiqle lets users express relational queries over program traces [Goldsmith et al. 2005]. In contrast to these approaches, our checklist assistants are specifically intended to tackle imprecision in static analysis tools. Martin et al. [2005] propose PQL (Program Query Language), a simple language for writing static analyses that implemented via compilation to datalog programs that work with bddbddb [Whaley and Lam 2004]. We may be able to use ideas from PQL in developing our checklist assistants, but we hope to provide a more flexible system that employs a range of static analysis techniques rather than one approach.

5. Conclusion

In this paper, we propose to use *triaging checklists* as one key tool to make static analysis tools easier to use. While a good visualization is useful to explain a warning efficiently, a good triaging checklist provides users with clear and complete instructions to decide if a warning is truly an error or

false warning. We are investigating how checklists can be applied to a variety of static analyses, as well as how to trace sources of imprecision in static analysis to construct checklists that are highly tool- and error-specific. Additionally, we are also exploring *checklist assistants*, which are lightweight analyses designed to help users answer checklist questions.

Acknowledgments

References

Paul Anderson, Thomas Reps, Tim Teitelbaum, and Mark Zarins. Tool support for fine-grained software inspection. *IEEE Software*, 20(4):42–50, July/August 2003.

Nat Ayewah, Hovemeyer David, J.D. Morgenthaler, J. Penix, and William Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, September 2008.

Nathaniel Ayewah and William Pugh. A report on a survey and study of static analysis users. In *DEFECTS '08: Proceedings of the 2008 workshop on Defects in large software systems*, pages 1–5, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-051-7. doi: http://doi.acm.org/10.1145/1390817.1390819.

Nathaniel Ayewah and William Pugh. Using checklists to review static analysis warnings. In *DEFECTS '09: Proceedings of the 2nd International Workshop on Defects in Large Software Systems*, pages 11–15, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-654-0. doi: http://doi.acm.org/10.1145/1555860.1555864.

Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8, 2007.

Thomas Ball and Sriram K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, Portland, Oregon, January 2002.

D. A. Berman, S. T. Coleridge, and T. A. McMurry. Computerized algorithm-directed triage in the emergency department. *Annals of Emergency Medicine*, 18(2), February 1989.

Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The Blast query language for software verification. In Roberto Giacobazzi, editor, *Static Analysis, 11th International Symposium*, volume 3148 of *Lecture Notes in Computer Science*, pages 2–18, Verona, Italy, August 2004. Springer-Verlag.

David Brodie and Richard Wells. An evaluation of the utility of three ergonomics checklists for predicting health outcomes in a car manufacturing environment. In *Proc. of the 29th Annual Conference of the Human Factors Association of Canada*, 1997.

Craig Chambers, Jeffrey Dean, and David Grove. Frameworks for Intra- and Interprocedural Dataflow Analysis. Technical Report 96-11-02, Department of Computer Science and Engineering, University of Washington, November 1996.

Yih-Farn R. Chen, Glenn S. Fowler, Eleftherios Koutsofios, and Ryan S. Wallach. Ciao: A graphical navigator for software and document repositories. In *Proceedings of the International Conference on Software Maintenance*, pages 66–75, 1995.

Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977.

Roger F. Crew. ASTLOG: A Language for Examining Abstract Syntax Trees. In *Proceedings of the Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997.

CSE. Microsoft center for software excellence. http://www.microsoft.com/windows/cse/default.mspx.

Manuvir Das. Formal specifications on industrial-strength code: From myth to reality. In *Proceedings of the 18th International Conference on Computer Aided Verification*, page 1, August 2006.

Asaf Degani and Earl L. Wiener. Human factors of flight-deck checklists: The normal checklist, 1990. NASA Contractor Report 177549.

Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A Practical Framework for Demand-Driven Interprocedural Data Flow Analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992–1030, November 1997.

Matthew B. Dwyer and Lori A. Clarke. A Flexible Architecture for Building Data Flow Analyzers. In *Proceedings of the 18th International Conference on Software Engineering*, pages 554–564, Berlin, Germany, March 1996.

Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 237–252, Bolton Landing, New York, October 2003.

Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Fourth symposium on Operating System Design and Implementation*, San Diego, California, October 2000.

Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching Bugs in the Web of Program Invariants. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 23–32, Philadelphia, Pennsylvania, May 1996.

Fortify Software Inc. Fortify Source Code Analysis, 2007. http://www.fortifysoftware.com/products/sca/.

Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-Sensitive Type Qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, Berlin, Germany, June 2002.

Simon F. Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 385–402, New York,

NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: http://doi.acm.org/10.1145/1094811.1094841.

GrammaTech, Inc. CodeSonar, 2007. http://www.grammatech.com/products/codesonar/overview.html.

David Greenfieldboyce and Jeffrey S. Foster. Visualizing Type Qualifier Inference with Eclipse. In *Workshop on Eclipse Technology eXchange*, Vancouver, British Columbia, Canada, October 2004.

Samuel Z. Guyer and Calvin Lin. Client-Driven Pointer Analysis. In Radhia Cousot, editor, *Static Analysis, 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 214–236, San Diego, CA, USA, June 2003. Springer-Verlag.

B. M. Hales and P. J. Pronovost. The checklist – a tool for error management and performance improvement. *Journal of Critical Care*, 21(3), 2006.

Mary Hall, John M. Mellor-Crummey, Alan Carle, and René Rodrıguez. FIAT: A Framework for Interprocedural Analysis and Transformation. In *Proceedings of the 6th Annual Workshop on Parallel Languages and Compilers*, August 1993.

David Hovemeyer and William Pugh. Finding bugs is easy. In John M. Vlissides and Douglas C. Schmidt, editors, *OOPSLA Companion*, pages 132–136. ACM, 2004.

Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 178–187, New York, NY, USA, 2003. ACM. ISBN 1-58113-660-9. doi: http://doi.acm.org/10.1145/643603.643622.

Ciera Christopher Jaspan, I-Chin Chen, and Anoop Sharma. Understanding the Value of Program Analysis Tools. In *OOPSLA'07 Practitioner Reports*, 2007.

Yit Phang Khoo, Jeffrey S. Foster, Michael Hicks, and Vibha Sazawal. Path projection for user-centered static analysis tools. In *Proceedings of the ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, November 2008a.

Yit Phang Khoo, Jeffrey S. Foster, Michael Hicks, and Vibha Sazawal. Path Projection for User-Centered Static Analysis Tools. Technical Report CS-TR-4919, Department of Computer Science, University of Maryland, College Park, August 2008b.

Andrew J. Ko and Brad A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 301–310, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: http://doi.acm.org/10.1145/1368088.1368130.

Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. Dynamic query-based debugging of object-oriented programs. *Automated Software Engg.*, 10(1):39–74, 2003. ISSN 0928-8910. doi: http://dx.doi.org/10.1023/A:1021816917888.

Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: a program query language. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 365–383, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: http://doi.acm.org/10.1145/1094811.1094840.

Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, 2006.

Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: Context-Sensitive Correlation Analysis for Race Detection. In *PLDI '06*, pages 320–331, 2006.

Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. Predicting accurate and actionable static analysis warnings. In *International Conference on Software Engineering*, 2008. To appear.

Semmle Limited. Semmle — Query Technologies. http://semmle.com.

Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th Usenix Security Symposium*, Washington, D.C., August 2001.

John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 131–144, Washington, D.C., June 2004.