# Debugging by asking questions about program output

**Conference Paper** · January 2006

DOI: 10.1145/1134471 · Source: DBLP

**1 author:**

Andrew Jensen Ko
University of Washington Seattle
**114** PUBLICATIONS **3,757** CITATIONS

# Debugging by Asking Questions About Program Output

Andrew Ko
Human-Computer Interaction Institute
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh PA 15218
ajko@cs.cmu.edu

## ABSTRACT

One reason debugging is the most time-consuming part of software development is because developers struggle to map their questions about a program's behavior onto debugging tools' limited support for analyzing code. Interrogative debugging is a new debugging paradigm that allows developers to ask questions directly about their programs' output, helping them to more efficiently and accurately determine what parts of the system to understand. An interrogative debugging prototype called the Whyline is described, which has been shown to reduce debugging time by a factor of eight. Several extensions and generalizations to it are proposed, including plans for evaluating their effectiveness.

## Categories and Subject Descriptors
D.2.5. [**Testing and Debugging**]: Debugging aids, H.5.2. [**User Interfaces**]: Interaction styles, user-centered design

## General Terms
Reliability, Human Factors.

## Keywords
Debugging, Whyline, slicing, questions, assumptions, hypotheses, direct manipulation, errors, faults, program understanding, program comprehension.

## 1. INTRODUCTION

Numerous studies have shown that half or more of all software development effort is spent debugging [4, 5, 7, 9]. One reason for this may be that commercial debugging tools have not changed substantially for over 30 years: developers' primary tools for finding errors are still breakpoint debuggers and print statements. Recent efforts in automated debugging [1] are very powerful, but they require both successful and failed runs and do not support programs with interactive input. Other more interactive approaches such as slicing [10] and query languages [8] have potential, but none have been shown to significantly reduce debugging time relative to existing tools. I have done several studies [2, 3, 4, 5] that suggest this is because debugging always begins with a question about a program's *behavior*, and to use these tools, developers must map their question onto a tool's support for analyzing *code*.

I propose to remove this hurdle by allowing developers to directly ask the questions they naturally want to ask. This paper summarizes my approach, called *interrogative debugging* (ID), which aims to allow developers to ask questions about a program's output using direct manipulation (not natural language), and obtain answers in terms of the execution events and code that were responsible the behavior in question. In this paper I describe the claims behind ID in detail, and then describe a prototype that I have developed, called the Whyline [2]. I then propose several extensions and generalizations to the Whyline, and plans to evaluate their effectiveness.

## 2. INTERROGATIVE DEBUGGING

The central claim of ID is that support for asking questions about a program's *output*, instead of about *code*, will significantly reduce debugging time relative to existing tools. Why would this be true? To use current tools, developers must *guess* what code is responsible for a program's failure in order to place a breakpoint, to insert a print statement, to request a static or dynamic slice [10], or to form a query [8]. When the guess is wrong, the time spent investigating the false hypothesis is essentially lost (although there may be some side benefit from the understanding gained while investigating). Across several empirical studies of developers of varying expertise, I have demonstrated several trends that contribute to this lost time:

- Developers generally form *multiple* false hypotheses about what code caused a failure before forming a correct hypothesis [4], each taking considerable time to investigate.
- False hypotheses that go unchecked result in an incorrect understanding of the program that impacts future tasks [3].
- As part of investigating false hypotheses, developers mistakenly modify code that is not broken. In one study, about half of *all* errors were inserted for this reason [5].

How could a tool help developers make more accurate judgments about program behavior? The insight behind ID is that developers are more accurate in identifying inappropriate behavior *itself* than they are in identifying the *causes* of inappropriate behavior [5]. Therefore, ID tools allow developers to specify a program's inappropriate output. Then, the tool performs program analyses on the specified output in order to determine a set of relevant code fragments for the developer to examination. This set should prove to be more relevant than the set of fragments a developer would have determined by guessing. As part of these analyses, the tool also reveals any assumptions that the developer made in asking the question. For example, if a developer asks why something didn't happen when in fact it did, the tool can point out the discrepancy, synchronizing the developer's *understanding* of a program's execution with its *actual* execution.

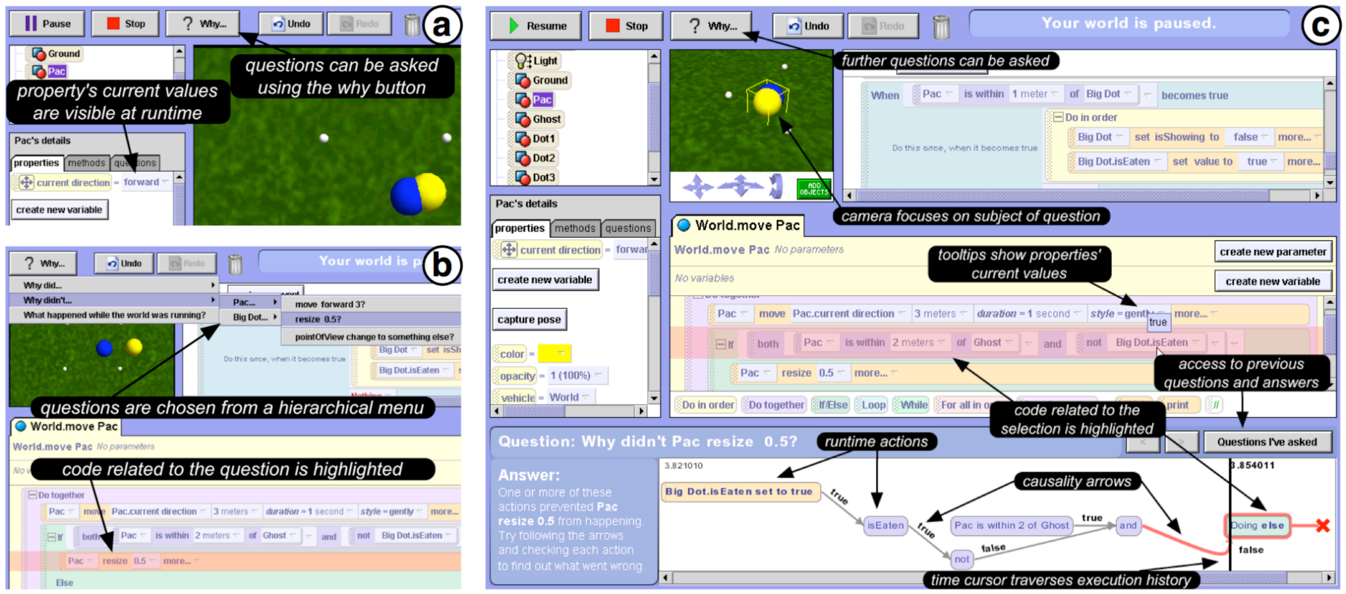
a
property's current values are visible at runtime
questions can be asked using the why button
b
questions are chosen from a hierarchical menu
code related to the question is highlighted
c
further questions can be asked
camera focuses on subject of question
tooltips show properties' current values
access to previous questions and answers
code related to the selection is highlighted
runtime actions
causality arrows
time cursor traverses execution history
Question: Why didn't Pac resize 0.5?
Answer:
One or more of these actions prevented Pac resize 0.5 from happening. Try following the arrows and checking each action to find out what went wrong.

**Figure 1. The Whyline prototype, implemented in the Alice 3D programming environment. When developers see failures at runtime, they can press the "Why" button (a) and get a menu of "Why did..." and "Why didn't..." questions (b) that correspond to output statements in the program. When a question is chosen, an answer (c) is provided in terms of the runtime events that caused or prevented the behavior from happening and the Whyline points out any assumptions the developer may have made.**

ID tools must have several characteristics to be successful. First, they should support both "Why did..." and "Why didn't..." questions, because programs fail by either *not* doing something they are supposed to do, or *doing* something they are *not* supposed to do [5]. Second, studies of how people describe software problems suggest that the ability to specify the context of a behavior is important [6]. Third, developers should be able to *choose* a question from a set of questions, rather than have to *construct* a question. This avoids the numerous issues with introducing a new query language [8] or developing a natural language interface. Fourth, the answer presented by an ID tool should directly associate the output in question with the code and the execution history responsible for causing or preventing the output. This is central to helping developers understand the causal relationships between code and behavior, which in turn will help developers conceive of more accurate repairs to the program.

## 3. THE WHYLINE

To test the feasibility of this approach, I developed the Whyline for Alice (www.alice.org) [2], shown in Figure 1. Alice is an object-oriented, multithreaded, imperative language and development environment for creating 3D simulations and games.

The example in Figure 1 shows a question being asked by a developer, who was implementing a Pac Man game and trying to get `PacMan` (the yellow sphere) to resize when he touches the `Ghost` (the blue sphere). When an Alice program is being executed, as in Figure 1a, the developer can ask a question by pressing the "Why" button, which pauses the program. The menu shown at the top of Figure 1b appears, which allows the developer to choose from a set of "Why did" and "Why didn't" questions that correspond to output statements in the program (which, in Alice, are animations such as resize and move). The questions are sorted by the objects that they refer to and the arguments they use. As the developer hovers over questions in the menu, the Whyline highlights the output statement in the program that corresponds to the question. In this example, the developer expected `PacMan` to resize after intersecting the `Ghost`, so he asks, "Why didn't `PacMan` resize 0.5?" When the developer chooses the question, the area shown at the bottom of Figure 1c appears, providing a textual answer to the question as well as a timeline visualization of the chain of execution events that caused or prevented the output statement from executing. In this case, the `BigDot.isEaten` flag was true, which prevented the resize animation from executing.

In another example (not shown), a developer placed two conditionals inside of a `DoTogether` construct (which allows statements to execute concurrently): one to make `PacMan` "eat" the `BigDot` by setting its `isShowing` flag to false, and one to shrink `PacMan` if he touches the `Ghost` *without* having eaten the `BigDot`. When the developer tested the code, `PacMan` ate the `BigDot`, but also shrank when he touched the `Ghost`. The developer asked, "Why did `PacMan` resize 0.5?" and the Whyline showed that he touched the `Ghost` while the `BigDot isShowing` was true. This confused the developer, and so he asked, "Why didn't `BigDot isShowing` change to false?" The Whyline responded by combining the answers to the two questions, revealing that the two conditionals had been interleaved.

In a study comparing Alice to Alice with the Whyline, developers with the Whyline spent a factor of 8 less time debugging and got 40% further through their tasks [2]. These gains were due largely to the fact that developers without the Whyline generated multiple false hypotheses about what code was causing failures, and spent considerable time investigating them, often inserting new errors in the program as they tried to fix existing ones. Developers with the Whyline simply asked about the output they expected, and were led directly to the code responsible for the output or lack thereof. This saved the time that would have been spent investigating false hypotheses, as well as the time spent repairing any errors that would be been inserted.

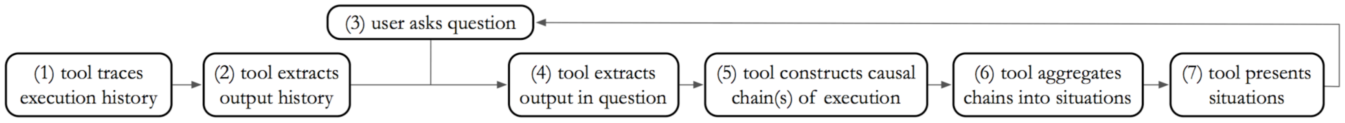
(3) user asks question
(1) tool traces execution history
(2) tool extracts output history
(4) tool extracts output in question
(5) tool constructs causal chain(s) of execution
(6) tool aggregates chains into situations
(7) tool presents situations

**Figure 2. Actions performed by the user and the system in an ID tool, and the data produced at each step.**

## 4. GENERALIZING THE WHYLINE

Although the Whyline for Alice demonstrated the feasibility of ID in the small, it raised several important issues regarding its applicability to more general-purpose languages and more complex programs. In particular, what is "output" in the general case? Can a reasonably sized question menu be generated for programs, in general? What kinds of "Why didn't" questions can be answered effectively? How large are the answers to questions about more complex programs?

To address these issues, my thesis work will involve designing and building a Whyline for desktop Java programs. My proposed solution is illustrated in Figure 2. The solution treats a developer's question as a query on a program's output statements and output history. The system analyzes the statements and execution events resulting from the query to produce chains of causality that are based on the program's execution history. In the rest of this section, I discuss the challenges posed in each of these steps and offer preliminary design ideas and their tradeoffs.

### 4.1 Tracing the Execution History

In order to allow developers to ask questions about a program's output in a general way, the tool must record a complete history of every "output" statement executed. Furthermore, to accurately answer questions about causality, we must record the data dependencies in the program's execution. Although recorded execution histories can be very large, recent projects have begun to address this issue by using compression and by only recording unrecoverable runtime state [10]. I also expect to develop new optimizations that are specific to any analyses that I develop.

### 4.2 Generating an Output History

Once an execution history is available, the tool needs to extract the "output" produced by the program. For Alice, there is a limited set of animations and properties for each object that have a visible effect or representation on-screen. In the general case, however, a developer might interpret any code fragment as producing "output," depending on the behavior being debugged.

I will first focus on supporting questions about the most general forms of output, namely, graphical primitives (shapes, lines, rectangles, etc.) and text (such as debug statements printed to a console) since they constitute nearly all of a typical desktop program's output. For Java, I could trace all calls on instances of `java.io.PrintStream` and `java.awt.Graphics2D` (such as `println` and `fillRect`). I can then use these output histories to reconstruct an interactive history of the graphical and textual output produced by the program. These reconstructed histories will (1) allow developers to freely navigate the output history produced by the program, simulating reverse execution, and (2) provide a user interface foundation for querying output.

In addition to these basic forms of output, I will also allow developers to customize the meaning of "output" and build their own interfaces for navigating and interrogating the output. For example, if a team debugging a web application wants to ask about queries sent to a database, they could have the Whyline instrument the method call that sends the query and build a custom interface for interrogating histories of queries. One problem with this approach is that developers are poor at guessing what to inspect because of the sheer number of possible explanations for a problem [5]. For example, the problem might not be with queries themselves, but with the transmission, or the database's interpretation of the queries, which are both independent of the query formation.

### 4.3 Asking Questions

Once the system has an output history, how can a developer ask a question about it? I performed a linguistic analysis of how people describe software problems [6] in order to determine the various ways that developers restrict the domain of their questions:

- *Why did or why didn't?* This determines the type of causality analysis to be formed, and also allows the tool to point out assumptions about what did or did not occur.
- *Referential scope.* A question might refer to all circles painted by a program, all circles painted by statement *s*, or the circle painted by statement *s* at time *t*.
- *Relative scope.* A question may be phrased relative to input events or other output events ("Why didn't a circle paint after this click?").
- *Argument scope.* Each type of output has different features that a developer may want to use to restrict a question's domain ("Why wasn't this circle green?").

In the Alice version of the Whyline, many of these scopes were ignored: each question referred to the most recent execution of a specific output statement, and questions could not be asked relative to other events. In the prototype I envision, each of these different scopes will be specified using direct-manipulation. For example, developers could specify the referential scope by clicking on a graphical or textual primitive displayed in the interactive history of output (this would not work for "Why didn't" questions, however, since they refer to code that was not executed; instead, these can be formed through hierarchical menu as in the Whyline for Alice, or by demonstrating the desired behavior). Developers could ask questions relative to an input event by clicking on a visual representation of the input event in the interactive history, dragging a time cursor to the time of interest, and then supplying other arguments for the question.

### 4.4 Applying Queries

Once a developer supplies a question, the system will use its scoping arguments to select a set of the output statements from the program and a set of execution events from output history. These statements and events will be analyzed in the next step to produce an answer to the developer's question.

At this point, the tool will also check for any false assumptions implicit in the developer's question. For example, if the developer asks, "Why didn't anything paint after this click?" when in fact something did, part of the tool's answer will reveal the output that the developer did not notice.

### 4.5 Analyzing Causality

The goal of this step is to produce one or more *causal chains* of execution events for the developer to inspect (like the one shown in Figure 1c). For each of the output events that is the subject of a "Why did" question, the system will determine what caused the event to occur. To perform these analyses, I will utilize the existing techniques, primarily slicing algorithms [10].

For each output statement that is the subject of a "Why didn't" question, the system will determine the execution events that prevented the output statement from executing, or executing with the desired arguments. In some cases, this is trivial to answer. For example, in the Whyline for Alice, an analysis simply determined the predicate that prevented the queried output statement from executing. This same analysis can work in the Java version, with added support for analyzing the reachability of the method that contains the predicate. For situations where an output statement *was* executed, but not with the appropriate values or at the appropriate time, I will devise new analyses that can identify alternate histories that could have resulted in the correct values.

### 4.6 Aggregating Chains

If the previous step resulted in multiple chains of causality, it is likely that many will have been executed in similar contexts and will have produced similar output. It would be helpful to aggregate and summarize them, to minimize the amount of information that a developer must understand to interpret the answer. These could then be presented as different *situations* in which the queried output did or did not occur.

### 4.7 Presenting Answers

The final step is to present the situations produced in the previous step in a user interface that helps the developer conceive of a modification that can correct the program's failure. The Whyline for Alice offered several features to this end: (1) a time cursor, which, when moved, changes the code shown and the state of the output history, to help developers relate the code to the behavior that it caused, and (2) support for asking elaborative questions about an answer, which combined multiple answers. Because the answers for larger more complex programs are likely to be larger and more complex themselves, new interaction techniques will be necessary to help developers efficiently navigate the information:

- *Incremental answer navigation*, which would allow dynamic slices to be computed incrementally and on demand, affording more immediate answer feedback.
- *Situation workspaces*, which maintain state for each situation such as its time cursor position, the visible code, and the interactive state of the chain visualization. This way, developers could stop inspecting a situation and return to it later without having to remember where they were.
- *Multiple selection of answer elements*, which would allow developers to compare multiple program states and code fragments side by side.

## 5. EVALUATION

To evaluate the effectiveness of the prototype, I will perform an experimentally controlled comparison of the *debugging efficiency* and *effectiveness* of Java developers using my prototype, against Java developers using breakpoint debuggers and state of the art research prototypes, such as those by Clevel and Zeller [1] and Lencevicius et al. [8]. The test programs that developers debug in the experiment will be varied in size, and the failures that they diagnose will be naturally occurring, rather than artificially created. In order to determine what particular features of the prototype contribute to its effectiveness, I may include conditions in the experiment involving crippled versions of the prototype. For example, one such version might only allow developers to ask about code, and not output, in order to assess the importance of asking about output. I also plan to widely deploy the prototype for Eclipse in order to gather data on long-term use and effectiveness.

## 6. CONCLUSION

Debugging and program understanding are fundamental bottlenecks in software development, largely because developers struggle to effectively translate their questions about program *behavior* into primitive analyses of *code*. Interrogative debuggers will allow developers to inquire directly about program behavior, helping them to more quickly and accurately determine the relevant code fragments in a system, as compared to existing debugging tools. This work will have several other contributions, including new interaction techniques for interrogating program output, new program analyses for answering "Why didn't" questions about program output, and new techniques for identifying similar "situations" in an execution history.

## 7. ACKNOWLEDGMENTS

This work is supported by the National Science Foundation under NSF grant IIS-0329090 and the EUSES consortium under NSF grant ITR CCR-0324770. The author is also supported by an NDSEG fellowship.

## 8. REFERENCES

[1] H. Cleve and A. Zeller, Locating Causes of Program Failures, *ICSE* 2005, St. Louis, MI.

[2] A. J. Ko and B. A. Myers, Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior, *CHI* 2004, Vienna, Austria, 151-158.

[3] A. J. Ko, B. A. Myers, and H. Aung, Six Learning Barriers in End-User Programming Systems, *VL/HCC* 2004, Rome, Italy, 199-206.

[4] A. J. Ko, H. Aung, and B. A. Myers, Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks, *ICSE* 2005, St. Louis, MI, 126-135.

[5] A. J. Ko and B. A. Myers, A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems, *JVLC*, 16, 1-2, 41-84, 2005.

[6] A. J. Ko, B. A. Myers, and D. H. Chau, A Linguistic Analysis of How People Describe Software Problems in Bug Reports, *Submitted for publication* 2006.

[7] T. LaToza, G. Venolia, and R. DeLine, Maintaining Mental Models: A Study of Developer Work Habits, *ICSE* 2005, Shanghai, China, to appear.

[8] R. Lencevicius, U. Holzle, and A. K. Singh, Dynamic Query-Based Debugging of Object-Oriented Programs, *J. of Automated Soft. Engr.*, 10, 1, 367-370, 2003.

[9] G. Tassey, The Economic Impacts of Inadequate Infrastructure for Software Testing, National Institute of Standards and Technology RTI Project Number 7007.011, 2002.

[10] X. Zhang and R. Gupta, Cost Effective Dynamic Program Slicing, *PLDI* 2004, Washington, D.C.