Extracting and Answering Why and Why Not Questions about Java Program Output

ANDREW J. KO
University of Washington, Seattle
and
BRAD A. MYERS
Carnegie Mellon University, Pittsburgh

When software developers want to understand the reason for a program's behavior, they must translate their questions about the behavior into a series of questions about code, speculating about the causes in the process. The Whyline is a new kind of debugging tool that avoids such speculation by instead enabling developers to select a question about program output from a set of "why did and why didn't" questions extracted from the program's code and execution. The tool then finds one or more possible explanations for the output in question. These explanations are derived using a static and dynamic slicing, precise call graphs, reachability analyses, and new algorithms for determining potential sources of values. Evaluations of the tool on two debugging tasks showed that developers with the Whyline were three times more successful and twice as fast at debugging, compared to developers with traditional breakpoint debuggers. The tool has the potential to simplify debugging and program understanding in many software development contexts.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids; tracing*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*User-centered design; interaction styles*

General Terms: Reliability, Algorithms, Performance, Design, Human Factors

Additional Key Words and Phrases: Whyline, questions, debugging

This article is a revised and extended version of a paper presented at ICSE 2008 in Leipzig, Germany.

This work was supported by the National Science Foundation under NSF grant IIS-0329090 and CCF-0811610 and the EUSES consortium under NSF grant ITR CCR-0324770. The first author was also supported by NDSEG and NSF Graduate Fellowships. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Authors' addresses: A. J. Ko, The Information School, University of Washington, Seattle, WA 98195; B. A. Myers, Human-Computer Interaction Institute, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

Due to a conflict of interest, Professor D. Rosenblum was in charge of the review of this article.

DOI 10.1145/1824760.1824761 http://doi.acm.org/10.1145/1824760.1824761

ACM Reference Format:
Ko, A. J. and Myers, B. A. 2010. Extracting and answering why and why not questions about Java program output. ACM Trans. Softw. Eng. Methodol. 20, 2, Article 4 (August 2010), 36 pages.
DOI = 10.1145/1824760.1824761 http://doi.acm.org/10.1145/1824760.1824761

1. INTRODUCTION

Software developers have long struggled with understanding the causes of software behavior. Yet, despite decades of knowing that program understanding and debugging are some of the most challenging and time-consuming aspects of software development, little has changed in how developers work: these tasks still represent up to 70% of the time required to ship a software product [Tassey 2002].

A simple problem underlies this statistic: once a person sees an inappropriate behavior, he or she must then translate questions about the program's output into a series of queries about the program's code. In doing this translation, developers must guess which code is responsible [Ko et al. 2006b]. This is worsened by the fact that bugs often manifest themselves in strange and unpredictable ways: a typo in a crucial conditional can dramatically alter program behavior. Even for experienced developers, speculation about the relationship between the symptoms of a problem and their causes is a serious issue. In our investigations, developers' initial guesses were wrong almost 90% of the time [Ko et al. 2006b].

Unfortunately, today's debugging and program understanding tools do not help with this part of the task. Breakpoint debuggers require people to choose a line of code. Slicing tools require a choice of variable [Baowen et al. 2005]. Querying tools require a person to write an executable expression about data [Lencevicius et al. 2003]. As a result, all of these tools are subject to a "garbage-in garbage-out" limitation: if a developer's choice of code is irrelevant to the cause, the tool's answer will be similarly irrelevant. Worse yet, none of today's tools allow developers to ask why not questions about things that did not happen; such questions are often the majority of developers' questions [Ko and Myers 2004; Ko et al. 2006b]. (Of course, lots of things do not happen in a program, but developers tend to only ask about behaviors that a program is designed to do.)

In this article, we present a new kind of program understanding and debugging tool called a *Whyline*, which overcomes these limitations. Rather than requiring people to translate their questions about output into actions on code, the Whyline allows developers to choose a why did or why didn't question about program output and then generates an answer to the question using a variety of program analyses. This avoids the problems noted above because developers are much better at reasoning about program output, since unlike the execution of code, it is observable. Furthermore, in many cases, developers themselves define correctness in *terms* of the output [Ko et al. 2006a].

This work follows earlier prototypes. The Alice Whyline [Ko and Myers 2004] supported a similar interaction technique, but for an extremely simple language with little need for procedures and a rigid definition of output (in a lab

study, the Whyline for Alice decreased debugging time by a factor of 8). The Crystal framework [Myers et al. 2006], which supported questions in end-user applications, applied the same ideas, but limited the scope to questions about commands and events that appear in an application's undo stack (in lab studies of Crystal, participants were able to complete 30% more tasks, 20% faster).

These successes inspired us to extend our ideas to support Java programs, removing many of the limitations of our earlier work. In this article, we contribute (1) algorithms for extracting questions from code that are efficient and output-relevant; (2) algorithms for answering questions that provide nearly immediate feedback; and (3) a visualization of answers that is compact and simple to navigate. We achieve all this with no limitations on the target program, other than that it use standard platform-independent Java I/O APIs and that the program does not run too long before the problem occurs (given our trace-based approach).

This article represents an expanded version of our earlier conference paper on the Java Whyline [Ko and Myers 2008]. There are two major differences between this article and the earlier conference article. First, here we include enough algorithmic and architectural detail for an experienced developer to replicate our work. These details include new algorithm listings, architectural details, details on the Whyline's recording format, and challenges with recreating an interactive output history. Second, we have included an expanded discussion of limitations, implications, and future work, providing a balanced perspective on the strengths and limitations of the Whyline approach. Complete information about the Whyline, including the motivating user studies and summative evaluations, are available in the first author's dissertation [Ko 2008].

2. AN EXAMPLE

To motivate the implementation, let us begin with an example of the Java Whyline in use. A study reported in Ko et al. [2006b] used a painting program that supported colored strokes (see Figure 1(a)). Among the 500 lines of code, there were a few bugs in the program that were inserted unintentionally, which were left in for the study. One problem was that the RGB color sliders did not create the right colors. In this study, participants used the Eclipse IDE for Java and took a median of 10 minutes (from 3 to 38) to find the problem. The high variation in times was largely due to the participants' strategies: most used text searches for "color" to find relevant code, revealing 62 matches over 9 files; others followed data dependencies manually, sometimes using breakpoints.

With the Whyline, the process is greatly simplified (see Figure 1). The user simply demonstrates the behavior he or she want to inquire about (a), in this case by drawing a stroke that exhibits the wrong color. The user then quits the program and the trace is automatically loaded by the Whyline. The user then finds the point in time he wants to ask about by moving the time controller, the black vertical bar at (b). Then, the user clicks on something related to the behavior to pop-up questions about it (c). In this case, a user could click on the stroke with the wrong color and can then ask the question, "why did this line's color = ■ ?"

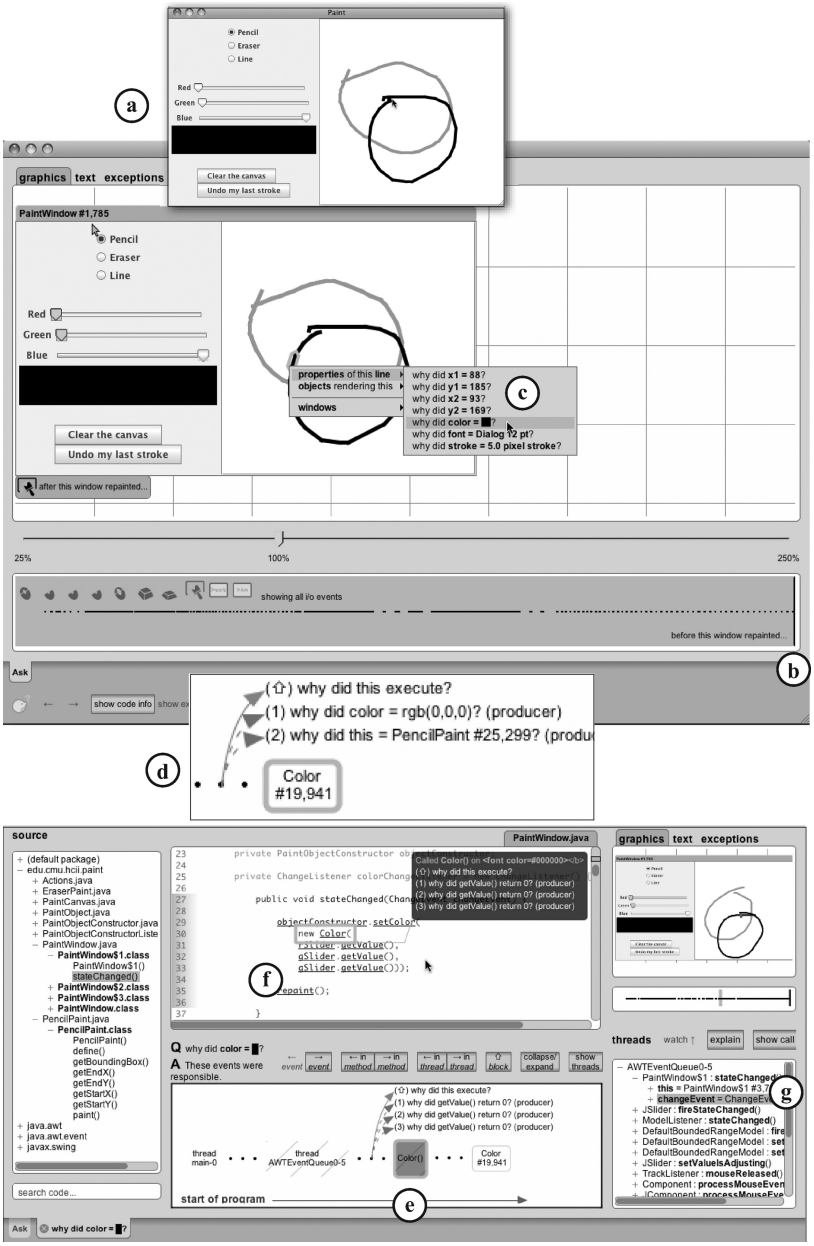

Fig. 1. Using the Whyline (*the highlighting and arrows above are part of the Whyline; the only annotations above are circled letters*): (a) The developer demonstrates the behavior; (b) after the trace loads, the developer finds the output of interest by scrubbing the I/O history; (c) the developer clicks on the output and chooses a question; (d) the Whyline provides an answer (the orange highlight indicating the selected event), which the developer navigates (e) in order to understand the cause of the behavior (f); (g) shows the call stack.

In response, the Whyline shows a visualization explaining the sequence of executions that caused the stroke to have its color (d),(e). This visualization includes assignments, method invocations, branches, and other events that caused the behavior. When the user selects an event, the corresponding source file is shown (f), along with the call stack and locals at the time of the selected execution event (g). In this case, the Whyline selects the most recent event in the answer, which was the color object used to paint the stroke (d). To find out where the color came from, the user could find the source of the value by selecting the label "(1) why did color = rgb(0,0,0)" (d). This causes the selection to go to the instantiation event (e) and the corresponding instantiation code (f). Here, the user would likely notice that the green slider was used for the blue component of the color; the blue slider should have been used.

In a user study of this task (reported elsewhere [Ko 2008]), people using the Whyline took half the time that participants with traditional tools took to debug the problem. This was because participants did not have to guess a search term or speculate about the relevance of various matches of their search terms, nor did they have to set any breakpoints. Instead, they simply pointed to something that they knew was relevant and wrong, and let the Whyline find the relevant execution events.

3. DESIGN AND IMPLEMENTATION

The Whyline is intended to support[1] interactive debugging (unlike automated debuggers, which take a specification of correctness to find potential causes of a problem [Cleve and Zeller 2005]). Therefore, the Whyline needs new incremental and cache-reliant algorithms to ensure near-immediate feedback for most user actions. The Whyline also takes a post-mortem approach to debugging, capturing a trace [Wang and Roychoudhury 2004, Zhang and Gupta 2005] and then analyzing it after the program has stopped, like modern profilers. This choice was based on evidence that bug-fixing is generally a collaborative process [Ko 2007], which could benefit from the ability to share executions of failures. The post-mortem approach was chosen explicitly for this purpose; an alternative design of "live" debugging could have been implemented, but would have required a different approach.

3.1 Software Architecture

The Java Whyline design has five major parts (each implemented in Java): the instrumentation framework, the trace data structure, the user interface, the question extractor, and question answerer. These are shown in Figure 2 and discussed throughout subsequent sections. Here we note their basic functions and relationships to each other.

The *instrumentation framework* consists of an API for reading, writing, and representing Java classfiles (similar to other bytecode APIs, such as BCEL,[2] but less error-prone in its instrumentation support) and support for reading,

[1]Our prototype is available at *http://www.cs.cmu.edu/~natprog/whyline-java.html*.
[2]http://jakarta.apache.org/bcel

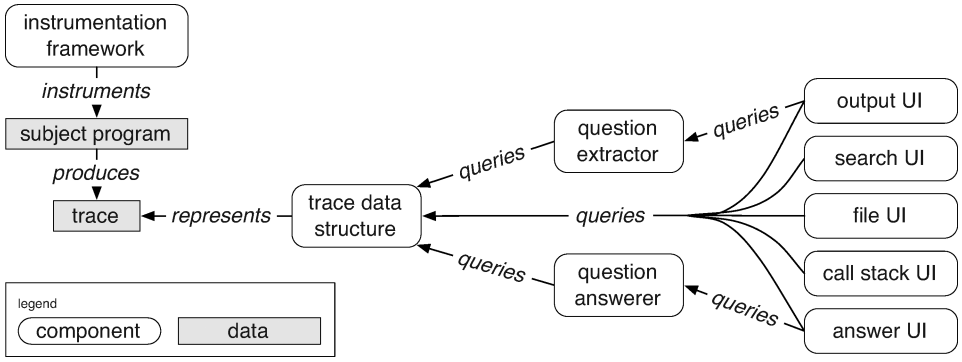

Fig. 2. A depiction of the major components of the Java Whyline architecture. The instrumentation framework inserts instrumentation calls into the subject program, which produces a trace data structure. The user interface components, question extractor, and question answerer all query the trace data structure in response to user requests, and the user interface presents the results.

analyzing, and instrumenting classfiles to capture a trace. The instrumentation framework modifies the *subject program* in Figure 2 such that a *trace* is produced when it is executed.

The *trace data structure* encapsulates all of a program's source, class files and execution history into a single data structure. It is designed to read the files that represent a Whyline trace from disk and represent it compactly in memory, shuttling information to and from the disk on demand. The trace is immutable, except for log information and annotations that the user might place on the immutable data. The trace data structure provides access to static and dynamic facts about the immutable trace, such as all of the executions of a particular method or all of the potential callers of a method, through several query and predicate functions. The results of most of these queries are cached to improve performance and serialized to disk so that subsequent uses of the trace can reuse these analyses. The trace data structure has detailed knowledge of how the instrumentation framework records events to disk, as it needs to know how to read them from disk.

The Java Whyline *user interface* components (the five components on the right of Figure 2) provide views of the trace data structure, facilitating users' questions by querying the trace and providing views of the information returned. Views of source files, execution events, call stacks, and so on, are generated on demand and discarded for optimal memory management. These user interfaces serialize their user interface state, such as window size and the visibility of various views, using a metadata framework supported by the trace data structure.

When a developer clicks on some output using the *OutputUI*, the *question extractor* queries the trace data structure for static and dynamic information and uses the results to display questions in menu form. When a question is selected, an *answer UI* is generated which queries the *question answerer* to present visualizations and answer text. The other user interface components interact directly with the trace data structure to present trace data based on the user's selection.

Table I. The File Hierarchy of a Recorded Whyline Trace

File	Purpose
`metadata`	number of events, objects, files, etc.
`Static...`	
`call graph`	constructed on the first load to speed up subsequent loads
`class identifiers`	unique identifiers for each loaded class, used to identify instructions in the event sequence
`class names`	used to find classes stored on disk that were loaded at runtime
`source...`	a source file hierarchy for the executed program
`dynamic...`	
`immutables`	a table of strings, colors, fonts, gradients, strokes, rectangles, and transforms, stored by object ID, used to recreate output history efficiently
`objects`	a history of object instantiations, stored by object ID
`thread histories`	a set of files, each containing a thread event histor

3.2 Recording Program Execution

A Whyline trace of an execution consists of a number of types of information: sequences of events that occurred in each thread (many of which regard program output); all class files executed and the source files that represent them; and other types of metadata recorded to interpret the data in the trace. This information is summarized in Table I. This section describes this information in detail.

3.2.1 *Recording Source Files.* Before launching the program, the Whyline scans the user-specified folders for user-defined source code, copying all of the current versions of the source. The directory structure of the source is maintained, whether in a platform-specific directory or a JAR file, so that the qualified name of the class defined by the source file can be recovered.

3.2.2 *Analyzing a Method for Instructions to Instrument.* In general, there are two major ways to capture an execution history of a Java program. One is to instrument a Java Virtual Machine to record a history of the program as it is executing.[3] This has the advantage of having potentially lower performance overhead, but the disadvantage of being platform- and VM-dependent. Instead, the Java Whyline uses bytecode instrumentation. As each Java class is loaded, the tool intercepts its byte array (using the `java.lang.instrument` package, standard across most JVM implementations after version 1.5), instruments each of the methods in the class, and returns the modified code as a byte array to the JVM. This approach allows the prototype to work in a largely platform-independent manner (although there may be inconsistent support for this particular instrumentation mechanism). The disadvantage is the complexity of inserting bytecode instructions into a Java program to capture information about its execution, as well as the additional overhead of executing

[3]An alternative to instrumentation would be to use the Java Platform Debugger Architecture (JPDA), which was not implemented for most platforms at the time of the implementation of the Java Whyline. This would allow more control over the subject program's execution, while still providing access to the same kinds of data.

the instrumentation code. A third option would have been to instrument the source, which is the most platform-independent manner of capturing a trace, but that approach has the most overhead and will not work with precompiled libraries.

The instrumentation process involves an analysis step and an instrumentation step. The analysis step identifies *control* and *data* instructions to instrument. Control instructions include invocations of methods, thrown exceptions, exception catches, and branch instructions (all part of the Java bytecode instruction set). These are straightforward to identify by simply parsing the bytecode and looking for particular opcodes. The data instructions in Java bytecode are more various, but generally involve instructions that affect the JVM operand stack (essentially arithmetic) and instructions that affect the JVM heap or local variable space (assignments to local variables, fields, globals, etc.). The Java Whyline instruments all of the latter category instructions. For the former category that only has operand stack effects, the Whyline instruments only those instructions that compute values for control instructions or data instructions with heap or frame side-effects. For example, in the Java assignment statement "x = a + b + c," the prototype would instrument the value produced by the final addition and the assignment instruction, but not the values pushed onto the operand stack by "a" and "b." This omission is purely for performance purposes. The value of "a" and "b" will be known from prior instrumented assignments, so recording their values at the time of use would be redundant. The one exception to this case is the use of global variables or public fields; such variables may be changed by uninstrumented code.

Of course, to know which instructions produce a value consumed by a control or assignment instruction, the prototype must first analyze the operand stack dependencies within a method. To do this, the prototype uses an algorithm that explores all execution paths through a method, and for each path, pairs instructions that push values onto the operand stack with instructions that later pop them off (this algorithm is similar to the verification steps performed by JVMs for security purposes). While exploring paths, the algorithm maintains a simulated operand stack, with each value-producing instruction on the stack representing the value produced. (The rules for whether an instruction produces and/or consumes a value are based on the Java bytecode specifications for each instruction.) This process determines a set of stack dependencies for each instruction in a method, allowing the system to perform a variety of analyses on the data dependencies within a method. For performance, the system caches these stack dependencies as a method attribute (defined in Section 4.7 of the JVM specification, second edition) to make class loading and analysis more efficient when a trace is loaded for the first time.

3.2.3 *Instrumenting a Method.* Next, the Whyline steps through each instruction, inserting a call to a global instrumentation method either before or after the instrumented instruction. Stack duplication instructions are also inserted if the instrumentation needs a copy of a value from the operand stack. For example, to record the result of an integer addition, `dup` instruction would be inserted to push a copy of the result onto the stack. An `invokestatic`

instruction would be inserted afterwards to call the `record_int()` method, which would pop this copied result and record it to the trace file. In other cases, the instrumentation call is inserted *before* the instruction; for example, to record a thrown exception, the event must be recorded before the `throw` instruction executes.

Each instrumentation call records specific information as a prefix to any other arguments included in the type of event. Each event has a header containing the following information. A 1 bit `switch` flag represents whether the event is the first occurring after a thread switch. If it is set, a 32-bit serial event ID is recorded. The IDs for all subsequent events follow this ID in sequence, until the next switch. Switches are identified when reading the trace by checking whether the next event ID follows the last in a thread. A 1 bit `io_callstack` flag is set to true if the code represents I/O or is necessary for maintaining a call stack, which helps the trace loader know which events to process immediately. The event type is represented with 6 bits (there are currently 55 types, as shown in Table II). Finally, 32 bits represent the instruction ID, consisting of two parts: a 14- bit class ID (maintained for all instrumented classes, across all programs), and an 18-bit integer represent the index of the instruction as it appears in the class file. (The largest JDK class file we have seen contains fewer than 200,000 instructions and is an outlier).

Event types include assignments, invocations and returns, thread synchronization events, exception throws and catches, instantiations of objects and arrays, and some special events to represent I/O events that are generated natively (such as mouse and keyboard events). All 55 are shown in full in Table II (with some events grouped, namely those that cover the eight primitive Java types but with the same semantics). Multiple studies have suggested that developers find concrete values essential for interpreting program state [Ko et al. 2004, 2006b]. Therefore, unlike prior work [Baowen et al. 2005; Wang and Roychoudhury 2005], many of these events also include a value after their header. For example, the Whyline records values passed as arguments to invocations and as well as values assigned to variables.

When recording an *object*, the Whyline obtains a unique 64-bit ID for it, creating a new ID if the object has not yet been encountered. These are stored in a thread-safe weak reference hash table, so that objects can be garbage-collected. For each new object encountered, the tool also writes the type of the object (as a class ID) with its object ID to a separate file. Thread IDs are managed in the same way at runtime.

3.2.4 *Special Instrumentation for I/O Events.* Most I/O events are extracted from the regular event sequence. For example, calls to `java.awt.Graphics` are captured as `invoke` events in the trace just like any other call, and these are used to identify graphical output events and their arguments. Some I/O events benefited from or required special support, namely the last six event types in Table II. For example, the prototype replaces all calls to `java.awt.Window.getGraphics()` with a custom call, which gathers information about the size and location of `Window` instances before returning the value originally requested. The prototype also inserts custom instrumentation into

Table II. The 55 Different kinds of Events Recorded by the Java Whyline

Event	Purpose	Event	Purpose
`putfield`	object field assignments	`throw`	captured just before throw
`putstatic`	global variable assignments	`catch`	captured at beginning of catch block
`setarray`	array index assignments	`monitor`	before and after synchronized blocks
`setlocal/iinc`	local variable assignments	`constant`	for 8 primitive types; placeholder for constant used in expression
`comprefs/compnull`	reference comparison branch	`value`	for 8 primitive types; records value of expression or call
`compints/compzero`	integer comparison branches	`this`	records occurrence of event, but not value of reference (to save space)
`tablebranch`	switch statement branches	`newobject`	captured after constructors complete
`invoke`	the four JVM invocation instructions	`newarray`	captured after array instantiated
`start`	marker for the beginning of a method's execution, in case its call was not instrumented	`argument`	for 8 primitive types; captures value argument passed to method, in case call not instrumented
`return`	marker before method return		
Events that do not map to bytecode instructions			
`repaint`	used to mark a graphical repaint cycle	`context`	used to track duplications of graphics context used for rendering
`mouseevent`	tracks mouse input arguments	`keyevent`	tracks keyboard input arguments
`window`	tracks window size and state changes	`imagesize`	tracks the size of images drawn to screen for use in placeholders

The `constant`, `value`, and `argument` categories contain eight events each, to cover each of the eight primitive types in Java. The group of six events at the bottom are custom instrumentation to capture certain I/O events.

the constructors of the `java.io.KeyEvent` and `java.io.MouseEvent` constructors to capture information about low-level I/O events and their parameters. The latter two were added for performance reasons; extracting them from the normal event sequence at load time would have been possible, but slower than capturing it as a customized event.

3.2.5 *Instrumenting Programs.* The Whyline intercepts loading classes and performs a number of preprocessing steps before instrumenting the class. For each class loaded, the Whyline:

—copies the uninstrumented version of each class in a trace folder, in case the class was loaded off a network;

—uses the uninstrumented versions of classes for those that the user has marked to skip. The Whyline also skips classes that are used in the

instrumentation code itself as well as Java methods that, once instrumented, exceed the 65,536 byte-length limit imposed by the JVM;

—keeps track of each class referenced by each loaded class, in order to keep track of code *not* executed (for answering "why didn't" questions). Just before the program halts, the tool writes each of these unexecuted class files to the same trace folder. Ideally, this would be done recursively, in order to get the complete call graph of all of the code that the program could have executed, but this would take considerable time and likely include all known classes. Tracking classes that are never loaded is important for "why didn't" questions, since code that does not execute is often not dynamically loaded by the JVM;

—caches instrumented versions of the class files and their modification date so that later executions of the target program or other programs that use the same classes can load faster by avoiding redundant instrumentation. This is particularly useful for API classes that are used by many programs.

There are four major files recorded to disk in a Java Whyline trace (summarized in Table I).

—A file declaring a list of fully qualified class names that were loaded or referenced by a program execution, along with global class identifiers for each. The names are listed in loaded order.

—An "immutables" file, which stores constant values used by the program execution. This includes all of the strings referenced, the names of threads, and custom support for common immutables, such as `java.awt.Color`.

—A source file hierarchy, as just described.

—A set of event sequences, one for each thread. Each thread history is formatted as stated above in a separate file.

When a program halts, the Whyline writes a few bits of metadata to disk as well, to note how many events were written, how many objects were instantiated, and so on, to help the loader later create reasonably sized data structures to store the information. If the program halts in such a manner that prevents this information from being captured, the trace can still be loaded, but less efficiently. This usually occurs because the program unexpectedly crashes or is forced to terminate because of an infinite loop, deadlock, or other fatal condition.

3.3 Loading a Recording

When a Whyline trace is loaded, it performs a number of duties to prepare for question-asking. First, the source files and class files are loaded, since these are used for nearly every aspect of question-asking and- answering. As classes are loaded, the loader also processes several types of static information extracted from the class files:

—Associating invocation instructions with methods potentially called,

—Associating field references with field declarations,

—Associating class references with class declarations,

—Gathering all known "primitive output" instructions, which in this prototype include all calls on `java.awt.Graphics, java.io.PrintStream`, thrown exceptions, and exception catches. These are later analyzed to generate questions.

After loading this static information, the Whyline generates a precise call graph, using all of the invocations found in class files. *Precise* in this situation means that rather than using the type declared in the invocation instructions, the tool uses the analysis in Figure 3, which scans definition-use edges to find transitively reachable `new` expressions from the code's receiver. The result is a set of `new` instructions, which represent potential sources of new instances for the given instruction. This set is used to conservatively find all of the potential types of the actual instance used in the call, and resolves the method on these types. This omits many types of infeasible calls, increasing the precision of "why didn't" answers. This algorithm is called on demand whenever the Whyline needs to identify a set of potential callers to a method. (Whenever an algorithm in this article refers to a "caller," the set of potential callers is identified using this algorithm.) It should be noted that many call graph construction algorithms have been proposed, which may help make the algorithm in Figure 3 more efficient [Grove and Chambers 2001].

Next, the Whyline reads the thread traces, loading events in the order of their event IDs, switching between thread trace files as necessary using the `switch` flag in each event. This allows the Whyline to have a complete ordering of the events in the execution. As events are read, events whose `io_callstack` flag are set are processed immediately (essentially output and events needed to maintain a call stack); others are processed on demand. As call stacks are maintained, they are cached at equal intervals to provide constant time access to the call stack state at any event.

To improve the performance of question extraction and answering, the Whyline constructs tables of invocations, assignments to fields and other types of variables, and the values produced by expressions, all by event ID. All of these histories are extracted from the serial thread histories the first time a Whyline recording is loaded. Most histories are stored as integer sequences of event IDs. For example, if there were 30 calls to some method `foo()`, the event IDs for each call would be stored in a sorted list. These lists can then be efficiently searched using a binary search.

3.4 Recreating an I/O History

After loading the static and dynamic information, the final duty of the loader is to create a primitive I/O history. This step is fundamental to the Whyline's question support, since every question extracted depends on the Whyline's ability to relate the pixels on the screen to the program logic responsible for them. The prototype assumes that a program uses standard Java I/O interfaces and their subclasses to produce output: `java.awt.Graphics2D` for graphical output, `java.awt.Window` to represent windows and `KeyEvent` and `MouseEvent` for input events in these windows, `java.io.Writer, OutputStream, PrintStream, Reader`, and `InputStream` for console and file I/O, and `java.lang.Throwable` for exception output. The Java Whyline does not record the native I/O, such as

```
getSources(Instruction inst)

  if values for inst are cached in global table, return them
  value_producers = {}
  add value_producers to global table, with key inst      // mark inst as visited
  if inst pushes a constant, the result of an expression,
      or a new object or array
    add inst to value_producers

  else if inst manipulates operand stack          // find what inst manipulates
    add getSources(inst's value producer) to value_producers

  else if inst gets a variable                 // find variable's potential value
    if inst gets an method argument
      for each call to inst's method
        for each producer of the arg that inst gets from call
          add getSources(producer) to value_producers
    else
      for each assignment to the variable used by inst
        add getSources(assignment) to value_producers

  else if inst gets an array element           // find array's potential values
    arrayAllocations = getSources(inst's array argument)
    for each allocation of arrayAllocations
      for each assignment of getElementSources(allocation)
        for each producer of assignment's element value
          add getSources(producer) to value_producers

  else if inst is an invocation                 // find potential values returned
    for each method that the inst could invoke
      if method is native, add inst to value_producers
      else for each return_instruction of method
        for each producer of return_instruction's return value
          add getSources(producer) to value_producers

  if instruction after inst is a type cast         // filter out illegal values
    exclude instructions from value_producers that produce types
      that do not conform to the type cast.

  return value_producers
```

```
getArraySources(Instruction newarray)

  if visited with newarray, return, otherwise, mark visited
  consumer = instruction that uses the array instance produced by newarray
  if consumer sets an array element
    include consumer as source of array element value
  else if consumer sets a local, field, or global
    for each use of the variable defined, getArraySources(use)
  else if consumer invokes a method
    for each method consumer could call
      for each use of newarray in method, getArraySources(use)
  else if consumer returns a value
    for each caller of consumer's method
      getArraySources(caller)
```

Fig. 3. Algorithm `getSources`, which gathers instructions that could produce a value for a given instruction's argument, and `getArraySources` which gathers instructions that could produce values for an array.

those used in some Java look-and-feels or in native UI toolkits such as the Simple Windowing Toolkit (SWT) used in Eclipse. This would involve instrumenting and recording code compiled for platforms other than the JVM, which was out of the scope of this implementation.

3.4.1 *Recreating Graphical Output.* The Whyline extracts graphical output events from the standard event sequence in a Whyline recording (described in previous sections). For example, a call to `Graphics2D.drawRect()` and the events producing its arguments are combined into an I/O event representing the rectangle drawn. This extraction process produces a sequence of I/O events, which the Whyline then uses to construct a user interface for navigating the I/O history, like the one seen in Figure 1. To recreate this history, the Whyline iterates through each I/O event, segmenting the event sequence into repaint cycles using the `repaint` event (in Table II). Within each repaint, the Whyline tracks the creation and duplication of `Graphics2D` instances, determining when and where each render event occurred on screen. A `Graphics2D` instance stores information about the current color, stroke, font, and origin, among other information, all used to determine the appearance of the next render call. Java programs duplicate these `Graphics2D` instances when painting in order to modify the render context and draw some output, without having to explicitly revert the rendering context to its previous state. Each render event is related to the `Graphics2D` instance that was responsible for rendering it. The Whyline then uses the history of modifications to these `Graphics2D` instances to determine the font, color, stroke, and so on, of each of the graphical primitives rendered.

While parsing repaint cycles, the Whyline also tracks the *visual occlusion* of render events. For example, if a hundred small rectangles were drawn into a buffer and then a large rectangle drawn over all of them, the hundred small rectangles would be marked "occluded" after the time of the larger rectangle's rendering. Once this process is complete, there is enough information to render the program output for any given time in the program's execution history. For a given time t, the Whyline finds all repaint cycles that began before $t$and renders all of the render events in each repaint cycle until reaching a render event that occurred after t. The Whyline uses the visual occlusion information to skip render events that are not visible at t, allowing users to "scrub" the I/O history at interactive speeds.

In order to allow users to point and click on render events, the Whyline uses a basic picking algorithm. Given a point p, the Whyline first finds which window contains p. Then, it searches through all of the visible render events in the window, gathering a bottom-to-top list of render events that contain p. This list of events is used to generate a list of questions about the top-most render event (i.e., a text label and a background rectangle), as well as the objects that the list of render events represent (i.e., the button represented by the text and rectangle). This process is described in the next section. For each item in the list, the Whyline also has to determine the original renderer of the output. This is because render events can be drawn into arbitrary image buffers and these buffers can then be rendered into other buffers (this includes double-buffering, which is used to ensure that whole screens are drawn at once to a physical display rather than pieces at a time, avoiding a flickered appearance). For example, to paint a gradient for a button in a UI, which can be an expensive operation, some systems will render part of the button's gradient into a buffer, then quickly paint the buffer at multiple locations to "tile" the output. Such

techniques are now ubiquitous in modern operating systems, meaning that any Whyline that supports graphical output needs mature support for tracking to which buffer a graphical primitive is drawn. What makes such support challenging is that a single event can be rendered into a buffer, but the buffer can be drawn multiple times and at multiple locations in other buffers. This means that a single render event can have effects on multiple places in the screen. Therefore, when mapping a user's mouse cursor to the graphical primitives underneath the cursor, the system must distinguish between the *render time* of the event and one or more *appearance times* in which the rendered output appeared in the physical display. A single user click in the graphical output thus refers to a list of render event pairs, which is then processed to create a list of questions.

3.4.2 *Recreating Console Output.* Console output and exception output are relatively straightforward to create, as it is just a list of strings and exceptions to display as a vertical list. Textual output events are extracted from the standard event sequence by watching for calls on `java.io.PrintStream`. Each textual output event, rather than referring to just the string printed by the print stream (as in the string in `System.out.println(‘‘message =’’ +message)`), refers to each individual argument used to concatenate the final argument. This way the Whyline can support questions about both the string `‘‘message =’’` and the string variable `message` independently. Once these are extracted, the sequence of textual output events is processed in order of execution, and text printed to the console is laid out onto a single line, advancing a single line on the occurrence of each '\n' character. Clicking on a string in the console output generally results in a single question `‘‘why did this get printed?’’`.

3.5 Extracting Questions

In any program execution, many things happen, and many things do not. The Whyline uses both static and dynamic analyses to extract questions about these behaviors that the developer may or may not have expected.

"Why did" questions refer to a specific event from a trace; the questions available for asking depend on the input time selected by the user (Figure 1(b)), since this time also determines what events are visible on screen (this differs from the Alice version [Ko and Myers 2004], which required the user to pause the program at the time desired). When the user clicks on an output event, the Whyline shows questions related to the output event selected. For example, in Figure 1(c), "why did" questions relate to the properties of the line the user has selected.

In addition to questions about output primitives, it is also helpful to have questions about higher level concepts that these primitives represent (e.g., in addition to questions about rectangles, also the supporting questions about a button that is drawn using the rectangles). The Java Whyline supports questions about two types of higher level entities. The first kind of entities are fields of Java objects that indirectly influence an output primitive's arguments; these

are *data* that affect primitive output.[4] For example, imagine a drop-down menu with a list of items; it is important to not only be able to ask about the individual items, but also about the list itself. The prototype follows all upstream dynamic control and data dependencies of the primitive output's arguments to identify fields that affected the primitive output's arguments, stopping when finding no more upstream dependencies. This amounts to a backwards dynamic slice, tuned to gather field references. The size of this list can be relatively large, since there are typically many upstream data dependencies, but the entities are organized by Java class, making the list easier to navigate.

The second kind of higher level question regards entities responsible for indirectly rendering low-level output; -these are *callers* that affect primitive output. For example, when clicking on the label of the button, the user may also want to ask about the button itself and its properties, such as its visibility, enabled state, and so on. These objects are found on the call stack of the invocation that rendered the primitive output, and all such objects are included in the list (with the exception of those filtered out as described in the next section).

3.5.1 *Filtering by Familiarity.* One way to reduce the size of the question menus is to filter the menus by familiarity. It is important to include only those objects that are relevant to output and that the user is likely to have created or used, since questions about unfamiliar classes or data structures will likely not seem relevant to the user. For example, in the Swing UI toolkit, `Button` UI class does not know how to draw itself. This is delegated to a `Button` look-and-feel class which renders the button. A developer may write code to instantiate a Button, but have no idea about the existence of its look-and-feel delegate. The same is true of Swing's `ButtonModel`, which is a helper class for storing a button's pressed state. To avoid presenting questions about these types of delegate and helper classes, the Whyline defines a notion of familiarity. A class is *familiar* if user-owned code either defines or references the specific class. In the prototype, user-owned code consists of those classes that were derived from source on the last compile (thus excluding APIs and libraries for which the developer has no source). One could imagine more sophisticated definitions for familiarity and ownership based on authorship, checkins, or other measures.

This notion of familiarity is used to filter the two types of higher level questions about data and callers. For callers, the Whyline inspects the call stack of the invocation that produced the selected output primitive and for each call stack entry that represents a call on an object and only includes questions about that object if the object is of a familiar class. This results in questions about `Buttons`, but not `ButtonUIs`, unless the user had directly referenced `ButtonUI` in their code. To filter questions about data, the Whyline only includes questions about familiar data structure classes, thus excluding helper classes such as `ButtonModel`.

[4]These types of questions were not originally included in the Java Whyline design; the need for such questions became apparent after piloting the design of the study in the evaluation described at the end of this article.

```
markAffectors(Instruction inst)

 if instruction been visited, return, otherwise, mark inst as visited

 if instruction acquires a field value      // mark assignments to fields
  mark field as affecting instruction
  for each definition of field, markAffectors(definition)

 else if instruction is an invocation       // mark data used by return statements
  for each method potentially called by instruction
   mark method as affecting instruction
   for each return instruction in method, markAffectors(return)

 for each control dependency of instruction // mark code causing instruction to execute
  markAffectors(control)

 for each instruction data that instruction is data dependent on
  markAffectors(data)
```

```
markInvokers(Instruction inst)

 if instruction has not been visited    // mark callers to method of instruction
  mark instruction as visited
  mark instruction's method as invoking inst
  for each caller of instruction's method, markInvokers(caller)
```

Fig. 4. Algorithms `markAffectors` and `markInvokers` which mark methods and fields that affect or invoke output (the two algorithms do not invoke each other).

3.5.2 *Filtering by Output Modality.* Another way to filter question menus is to exclude code structures that do not affect the *modality of output* in question. For example, if a user is asking about graphical output, it can exclude questions about fields, methods, and classes that only indirectly affect textual output. To accomplish this, the Whyline finds fields and invocations that could have affected each known output instruction, using the first algorithm in Figure 4. For example, the color used to draw a rectangle might be affected by some field in an object or by the return value of a call to some method. To find these fields and invocations, the algorithm follows upstream static data dependencies, marking fields and methods as "output-affecting" along the way, keeping track of the modality of the output. This way, methods and fields are marked as affecting graphical output, textual output, or other types.

Next, if the output instruction directly *invokes* output (such as drawing a rectangle, unlike setting the color, which merely affects appearance), all potential indirect callers to the output instructions method are marked as *output-invoking*. This is done by following potential callers of a method, starting with the output instruction's method (bottom of Figure 4). Each algorithm is run on each primitive output instruction, and halts either when reaching an instruction already visited for a particular modality or code with no dependencies.

One detail not mentioned in the algorithms in Figure 4 is how the algorithm traverses potential callers of methods. Aside from the precise call graph mentioned earlier in this article, the algorithm also tracks the class of the method that the propagation begins in, and remembers this class during the traversal of potential callers. For example, if the propagation started in

a method of the `javax.swing.jButton` class, then arrived later at some call on a `java.awt.Component` (a superclass of `JButton`, but eventually arrived at a method of `javax.swing.JComboBox` (which is not a subclass of `JButton`, the algorithm would know not to follow any calls on the `JComboBox`, because the original source of output was a `JButton`. This allows the algorithm to exclude infeasible calls as part of the call graph traversal by propagating the class that originated the output.

Intuitively, it would seem these algorithms mark everything; after all, what code is not responsible for affecting or invoking some output? Practically, however, the parts of applications that affect different kinds of output are often isolated from each other. Because the algorithms in Figure 4 are run on each individual output instruction, the Whyline knows what kind of output a particular field or method can affect. The Whyline can use this knowledge to generate and filter questions based on the kind of output the user expresses interest in (whether textual, graphical, or otherwise).

3.5.3 *Creating Questions.* Once the Whyline identifies each entity represented by the selected output primitive, the Whyline generates questions for each entity. The first questions identified regard the properties of the selected primitive output (Figure 5.1) and the rest of the questions regard entities related to the primitive output. These include "why did" questions about each of the familiar, output-affecting fields' current values, such as "why did this Button's visible = true?" (Figure 5.2) and also "why didn't" questions about why these fields were not assigned after the selected time (Figure 5.3). Each of these questions points to the most recent assignment to the field on that instance. The Whyline also generates questions about objects that indirectly invoked the selected output primitive, including questions about the creation of the object (Figure 5.4), about the objects output-affecting fields, and about output-invoking methods that may not have executed (Figure 5.5).

The actual phrasing and presentation of questions depends on the type of output. Exceptions thrown by the program, caught or uncaught, are phrased as "why did" questions, and map to a throw event. Output in the console history supports questions about why a particular string was printed (mapping to the event that produced it). The questions supported for graphical output are somewhat more diverse, because the output itself is more complex in nature. For primitive-level output, such as a line, circle, or rectangle, users may ask "why did" questions about any of the properties used to render the output. These correspond to arguments passed to the render method, such as position and size, as well as state in the `Graphics2D` object such as color and font.

"Why didn't" questions refer to one or more instructions in the code. The Java Whyline currently supports "why didn't" questions about:

—*Output-affecting fields*, such as "why didn't this Button's hidden field change?"
—*Output-affecting methods*, such as "why didn't this Button repaint()?" and
—*Output-affecting classes*, such as "why didn't a Button appear?"

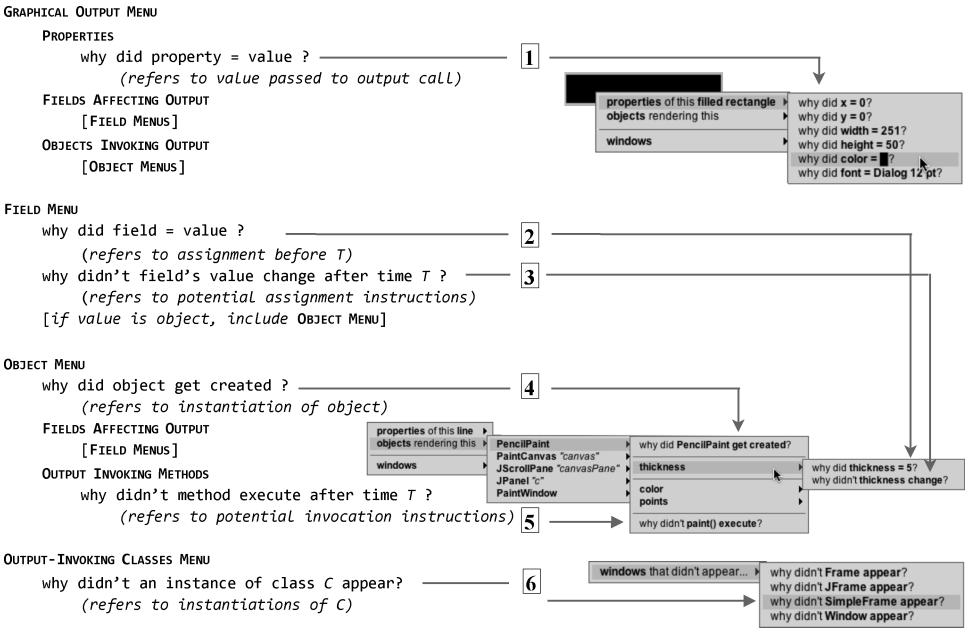

Fig. 5. All supported questions for a graphical output event in the Java Whyline prototype, showing six types of questions currently supported by the prototype (numbered 1–6) and three types of menus. For each, the content on the left lists the meaning of the question (items in [] represent nested menus of the specified type) and the content on the right gives an example screen shot.

For "why didn't" questions about fields, there are two types. For discrete-valued variables such as booleans or enumerated types as well as constant-valued objects, the system can identify specific values for "why didn't" questions. For example, one might ask "Why didn't the filled rectangle's color = red?" if the program referred to the constant `Color.red`; these values are found by following upstream data dependencies until reaching constant values. For continuous-valued variables (integers, floats, etc.), this is usually not feasible; for these variables, the system instead supports questions of the form "why didn't the variable get assigned?" For both kinds of questions, there may be numerous places that could have caused a variable to be assigned, these questions refer to the set of potential assignment statements. These instructions are grouped into a single question to avoid user speculation about which particular source should have executed; instead, all of them are considered together. For "why didn't" questions about methods, the system analyzes all of the potential callers of the subject method.

The last type of "why didn't" question supports questions about output that has no representative output to click on. For example, a user might have expected a dialog box to appear after a certain input event, or a console string after a certain action; but there are no primitives to choose that would enable questions about such output. To support these, the Whyline includes a question for each familiar class that has output-invoking methods inherited or

declared. An example of the resulting menu is seen in Figure 5.6, showing several different types of windows that could have appeared (the prototype does not yet add questions about nonwindow classes, but this is a straightforward addition).

There are a few types of questions that the Whyline does not support. First, it does not support questions about concrete values of continuous variables ("why didn't x = 6.08?"). This is partly because answering such questions can be computationally expensive and that such answers can pose too many possible reasons to be useful. Moreover, developers do not often know precisely what value to expect. Rather, they might guess that x > 0 and around 5; such specifications would require new algorithms to support. As an alternative to these kinds of questions, the Whyline allows users to ask "why didn't x change?" questions or simply to ask the positively phrased version of the same question to find the source x's value.

One other type of unsupported question is about the effects of input such as "why didn't this drag event do anything?" The problem with such "forward-looking" questions is that even in very simple situations, input events have many effects on programs, and with no expectation provided to the system there is no way to filter all of the things that do occur. Instead, users can ask the "why didn't" questions discussed throughout this article, which inquire about some expected output *after* an input event. This is an unfortunate but necessary restriction. Future work might investigate techniques for adding forward-reasoning questions (one place to start might be the Garnet toolkit's limited support for forward reasoning "why didn't" questions about input events [Myers 1990]).

The Alice Whyline [Ko and Myers 2004] and Crystal [Myers et al. 2006] both contained global why menus. The Java Whyline does not, largely because there are far too many events that do and do not occur in a Java program globally. If there were a menu, there would be too few constraints on what appears in the menu. Having the user choose a particular output at a particular time is central to providing a reasonably-sized question menu.

3.6 Answering Questions

In this section we discuss how the Whyline computes answers to "why did" and "why didn't" questions. Details on the presentation and interactive features of these answers in the Whyline user interface appear elsewhere [Ko and Myers 2009].

3.6.1 *Answering "Why Did" Questions.* Although there are a variety of types of "why did" questions, each maps to an event from the program's execution history. The approach to answering them is to work backwards from the event to find its chain of causality. Dynamic-slicing techniques [Baowen et al. 2005], which use a concept of control [Cooper et al. 2001] and data dependencies, are a natural approach to constructing these chains. However, the typical approach of dynamic-slicing algorithms is to generate a set of instructions and present them to the user as a set of highlighted lines of code. Given evidence that developers tend to work backwards temporally [Ko et al. 2006b;

Weiser 1982], this seems less helpful, therefore the Whyline produces a visualization of events, temporally sequenced. This visualization is a tree of the events that are traversed in a typical backwards dynamic-slicing algorithm. Although the algorithm is essentially the same [Baowen et al. 2005], the difference in data structures affect how the information is presented to the user: a chain of events shows what happened at runtime temporally, whereas a set of instructions simply states dependencies, many of which a user might already know.

Another difference from traditional slicing algorithms is that each event's control and data dependencies are computed *on demand* when a user selects an event. For example, rather than have the algorithm automatically traverse all of the data dependencies in a slice, the user explicitly chooses the data dependencies to navigate in the form of "follow-up questions" (Figure 1(d)), similar to the interaction in *thin slicing* [Sridharan et al. 2007]. This means that answers are produced almost immediately, making slicing time largely moot, unlike previous slicing systems which process answers in full as a batch process [Baowen et al. 2005].

Some "why did" questions pre- and postprocess slicing algorithm inputs and outputs to increase the utility of the answers. For example, when answering a question about an argument value passed to a method, the system first finds the "source" of the value by default. The source essentially follows data dependencies backwards until reaching a data dependency with multiple incoming dependencies (not counting control dependencies). To be concrete, imagine a color is instantiated in a call and then the color is passed, unmodified, through a dozen other calls until it is finally used. When asking about this color (the "follow-up questions" about data dependencies, like those in Figure 1(d)), the system follows these data-passing dependencies backwards until reaching the source, which might be an instantiation, an expression, or the return value of an unrecorded method. The assumption that this analysis makes is that the *calls* made to pass such data are not buggy, but that the data itself is buggy. If this assumption is not true, the analysis will skip over the buggy code; for example, perhaps the color was obtained from the wrong call. To account for this, the system also allows users to follow direct data dependencies and avoid skipping these potentially erroneous intermediate steps.

3.6.2 *Answering "Why Didn't" Questions.* "Why did" questions analyze an event by searching backwards in the history at a certain time. "Why didn't" questions analyze one or more potentially unexecuted instructions forward from the I/O event the user has selected using the time cursor. A "why didn't" query thus consists of one or more instructions, a time, and, in addition, an optional constraint on the expected conditions of the given instructions' execution (discussed shortly). For example, imagine a question about a button's `enabled` field; there may be three places this enabled field could be assigned. Each potential assignment is analyzed individually, generating individual answers. These answers are then combined into a final single answer.

To explain each individual instruction, the Whyline uses two analyses: (1) determining why an instruction was not executed, and (2) determining why

the wrong value was used. Each of these is constrained by two types of scope. *Temporal scope* affects what events it considers. For example, a developer may ask about something that did not occur after a specific event, but may have occurred in other situations. Therefore, "why didn't" analyses only search through events that occurred after the event selected by the time cursor (see Figure 1(b)) and before the end of the program. This omits other executions of events and reduces the amount of information to process. The tool could have supported scopes that end at a time different than the end of the program, but developers are notoriously bad at predicting precisely when something should have happened in the future; including the whole scope ensures that they make no false assumptions. Developers are fine at knowing the time after which something should happen, since most things happen as the result of a user action.

Identity scope is the second kind of scope, which considers what object(s) the developer has expressed interest in. For example, if they have asked why the "hidden" field of a button did not change, the analyses are restricted to events on that specific button instance. This *calling constraint* is propagated through the algorithms discussed next.

Why was this instruction not executed? To explain why an instruction was not executed, the first thing the Whyline does is check if it did execute. Our studies have found that developers are often prone to misperceiving output [Ko and Myers 2004; Ko et al. 2006b], and believe something has occurred when it has not (for example, believing that something did not change color, when it did, but then changed back). By supporting "why didn't" questions about things that did happen, the Whyline can reveal these assumptions.

If the instruction did not execute, the Whyline uses the algorithm `whynotreached` listed in Figure 6, to explain why the instruction did not execute after a particular time. The approach is to explore all reasons why an instruction was not executed after a specified time. Algorithm `whynotreached` gathers unexecuted instructions in an iterative control-flow graph traversal; the helper function `explain` considers a single instruction and the various reasons why it may not have been executed, identifying either (1) one or more instructions that needed to be executed in order for the given instruction to be executed or (2) the execution event in the trace that explains why the instruction did not execute. The data structure `UnexecutedInst` helps to represent the subgraph that `whynotreached` traverses, storing the incoming control-flow edges of the instruction, the reason for each instruction not being executed, and any execution events returned by `explain`. Therefore, the result of `whynotreached` is a directed graph of the program's control-flow graph including only those instructions that, if executed, could have caused the instruction of interest to be executed. Nodes that involve an invocation on a different object or a conditional branching in the wrong direction also have an execution event attached, explaining the source of the wrong object, or the value of the conditional's expression, respectively. When a question refers to multiple potentially unexecuted instructions, a single answer containing the union of these graphs is presented.

Algorithm `explain` in Figure 6 checks for possible reasons why an instruction was not executed. The first step is to check whether the method of the

```
structure UnexecutedInst
  Instruction instruction
  Set of Instruction reasons = {}
  String explanation
  int objectID (optional)
  int argument (optional)
  ExecutionEvent event (optional)

whynotreached(UnexecutedInst instruction, int time)

  checked = {}
  instructionsToAnalyze = { instruction }
  newInstructionsToAnalyze = {}

  while |instructionsToAnalyze| > 0

    for each instructionToAnalyze in instructionsToAnalyze
      if checked ∉ instructionToAnalyze
        include instructionToAnalyze in checked
        explain(instructionToAnalyze, time)
        include instructionToAnalyze.reasons in newInstructionsToAnalyze

    instructionsToAnalyze = {}
    include newInstructionsToAnalyze in instructionsToAnalyze
    instructionsToAnalyze = {}

explain(UnexecutedInst instruction, int time)

    method = instruction's method
    invocation = find most recent execution of method after time,
      optionally excluding invocations that did not use objectID for argument number
    if invocation was not found
      if method has no callers
        instruction.explanation = "unreachable"
      else
        instruction.explanation = "method was not called"
        for each caller of method's callers
          add new UnexecutedInst(caller)
            to instruction.reasons, translating argument and objectID to new calling context
    else
      if instruction is in exception handler and exception was caught
        instruction.explanation = "caught exception skipped over"
        instruction.event = exception thrown event
      else if method returned because of exception
        instruction.explanation = "exception thrown caused premature return"
        instruction.event = exception thrown event
      else if final instruction executed in method was at thread or program halt
        instruction.explanation = "method did not finish"
        instruction.event = final event of thread or program
      else if controlDependency executed
        instruction.explanation = "conditional branched in wrong direction"
        instruction.event = branch event
      else
        instruction.explanation = "conditional didn't execute"
        add new UnexecutedInst(instruction's control dependency) to instruction.reasons
```

Fig. 6. Algorithm `whynotreached`, which explains why a particular instruction in a Java program was not executed. The helper function `explain` analyzes the reason for an individual instruction not executing.

instruction of interest was executed at all. As shown in Figure 6, the explaination takes optional arguments `objectID` and `argument` if these are included, `explain` excludes method invocations that did not use the value in `objectID` for the given `argument` number. This allows questions about particular objects, such as "why didn't this button execute `doAction` ()?"

If an invocation satisfying these criteria was not found after the given time, then either: (1) the method has no known callers (one or more calls may exist, but they may not have been loaded at runtime); or (2), none of the known callers executed. In this latter case, the algorithm identifies the set of callers and calls `whynotreached` on each for further explanation. If the instruction's method did execute after the given time, then there are several possibilities for why the given instruction did not: (3) a caught exception jumped over the instruction of interest; (4) the method exited prematurely because of an uncaught exception; (5) the method did not finish executing because the thread or program halted; (6) the instruction's control dependency executed in the wrong direction (e.g., a conditional evaluated to true instead of false); (7) the instruction's control dependency did not execute.

One additional detail about case (2) above is worth noting. If an `ObjectID` and argument was passed to `explain`, then to pass this constraint backwards along the calls to a method, the algorithm must translate the constraint into the new calling context. For example, if the current expectation is that a virtual method `doAction()` was invoked on a particular `Button` (argument number 0 in Java bytecode terms), but the call was `button.doAction()`, then the argument number needs to change from 0 to the argument number of the local variable `button`. This expectation can then continue to filter invocations until the value no longer originates from a method argument.

There are a few exceptional cases to the algorithm for Java code. For example, some instructions have multiple control dependencies in Java, as in the case of code in an exceptional handler, where there may be multiple points of entry into the handler. In this case, all possible dependencies should be explained. Also, some methods have no explicit calls, such as `java.lang.Thread.run()`, and thus the answer generated by the algorithm should be careful in explaining that the method was not called (as opposed to explaining that it is not reachable).

Why was the wrong value used? Questions that ask about potential values of fields or primitive properties compare the expected dynamic dependency path to the actual dynamic dependency path at runtime. The former is obtained by tracking the path followed by `getSources` in Figure 3; the latter comes from the dynamic slice on the event that actually occurred, whether it was a field assignment or argument of an output instruction. (These are lists because the algorithm only analyzes unmodified values passed through intermediaries.) To illustrate, consider the following code, which controls a text field's background based on various states.

Imagine that the user expected the background to be red (line 8) and selected a question "why didn't this `TextField`'s color = red?" The expected dependency path from `H` would be `2, 1, 5, 4, 8`. Then imagine that, instead, the background was gray (line 10), with actual dependency path `2, 1, 5, 4, 10` or `black` with path `2,1,5,4,12`. In both cases, the point of deviation was 4: the program called `setBack()` with some color other than red. To explain why, the Whyline then checks whether the expected line (8) did execute. If it did and the other call to `setBack()` occurred after, then the color was overridden. If line 8 did *not* execute, then the tool uses the `whynitreached` algorithm in Figure 6 to determine why the instruction did not execute (in this example, it would be

```
 draw()                                          determineColor()
1  color = getBack()                           6  if(invalid)
2  setColor(color)                             7   if(enabled)
3  fillRect()                                  8    setBack(red)
                                               9   else
4 setBack(newColor) color = newColor           10   setBack(gray)
5 getBack() return color                       11 if(override)
                                               12  setBack(black)
```

```
whynotvalue(List of instructions expected, List of events actual)

 co-iterate through expected and actual, comparing
  instructions and finding point of deviation

 if deviation was not found, reason = value was used
 let exp be instruction after deviation in expected
 let act be event after deviation in actual

 if exp executed within temporal scope
   reason = value was used, but then overriden
 else
   whynotreached(exp)                // Algorithm described in text above
```

Fig. 7. Algorithm `whynotvalue`, which explains why a certain dynamic data dependency did not occur.

because enabled and/or invalid were false, or `determinecolor()` was not called). This algorithm is shown in Figure 7.

Limitations of "why didn't" answers. Because the "why didn't" answering algorithm uses a constrained traversal of a program's control-flow graph, the completeness of the control-flow graph greatly influences how much the user can trust the Whyline's answers to "why didn't" questions. For example, if the Whyline determines that there are no callers to a method and answers, so it may be that the actual calls occur through reflection or other mechanisms that the control-flow graph construction algorithm cannot detect. If the Whyline determines that there are callers to a method, it may be that none of the calls can feasibly be made at runtime (a precision issue). These precision issues can be dealt with by incorporating other research on creating precise control-flow graphs, but with a performance cost [Milanova et al. 2002].

4. EVALUATION

Although the primary contribution of this article is the implementation of the Java Whyline, it is worth summarizing the results of evaluations of its performance and its effectiveness. For example, several performance and scalability results are reported in Ko and Myers [2008] and Ko [2008], showing that the traces are reasonable in size and compress well; data on the instrumentation overhead shows that interactive programs have slowdown factors ranging from 2 to 8, and pipelined architecture programs like compilers have slowdown ranging from 10 to 20. These results are similar to those reported elsewhere for other tracing and analysis techniques [Zhang and Gupta 2005].

To date, we have completed two studies of the Java Whyline in use by developers. In the first study, nine people worked on the slider bug in Figure 1 after a 1 to 2 minute tutorial including information on how to ask questions and how to follow data dependencies. Their task performance was compared with that

of 18 self-described expert Java developers from the study in Ko et al. [2006b]. Overall, the participants with the Whyline completed the task in a median of 4 minutes, ranging from 1 to 12, significantly faster than the control group, which had a median of 10 minutes, ranging from 3 to 38 ($p < .05$, Wilcoxon rank sums test). The Whyline participants were more than twice as fast as the skilled developers without the Whyline. This is despite the fact that most of the Whyline users were self-described novices and that many of the developers in the control condition had already spent time understanding the design of the application. In fact, in the Whyline condition in this pilot study, the novices tended to outperform the skilled developers for some interesting reasons. The novices tended to say aloud, "Why is the line black?" and then use the Whyline to ask that question directly, quickly finding the cause. One novice said that "It was like a treasure hunt! It was fun! I didn't know debugging was like this." The skilled developers asked the same question, but then rather than proceeding to ask it with the Whyline, speculated about the possible reasons (e.g., "Why didn't this slider's event get handled?"), and then looked for a question that allowed them to check their speculation. When they failed to find such a question, only then did they ask about the color. One skilled developer explained that he did not "expect the Whyline to be able to make the connection between the slider and the color," and so he thought he had to make the connection himself. This led to a number of changes to the presentation of the data dependencies to make them appear as "follow-up questions."

In the second study, we compared skilled Java programmers using the Whyline to similarly skilled Java programmers using breakpoint debugging tools. The study had a between-subjects experimental design, with the independent variable of "debugging approach" and dependent variables of task completion time and task success. The goal was to determine whether the Whyline would significantly impact success at program understanding compared to modern debugging tools. The study participants consisted of 10 people in each group for a total of 20. Participants were all students in a masters program in software engineering, but had a median of 1.5 years of industry software development experience before coming back to school, ranging from 0 to 10 years. Participants worked on two tasks adapted from real bug reports of `ArgoUML`, a 150,000 line open source Java application for designing Java applications themselves using UML diagrams. The first bug involved removing a particular checkbox from the user interface. The second bug involved investigating a drop-down list of Java types that was supposed to contain all legal Java classes for a Java field, but was for some reason excluding classes in different packages that had equivalent names. All ten Whyline participants completed task 1, compared to only three control participants ($\chi 2 = 10.6$, $p < .05$). Whyline participants also completed task 1 twice as fast ($t = 4.5$, $p < 0.05$). The control participants who did finish the task explored hundreds of files, but got lucky in their searches, whereas the Whyline participants only explored a median of three. For task 2, of the ten participants, four Whyline participants were successful, compared to none in the control group ($\chi 2 = 5$, $p < .05$). This task was considerably more difficult; the successful Whyline participants spent all thirty minutes on the task, but much of it was in order to understand

some of the Java APIs used in constructing the list for the drop-down menu.

One interesting result of these studies was the suggestion that the Whyline might be a useful way to teach diagnostic strategies, such as that of working backwards from program output and exploring data dependencies. In fact, many of the participants in these studies of the Whyline, after getting Whyline answers about things that they thought did not happen, but actually did, commented to themselves about needing to be more cautious about assumptions. Participants would hover over questions about particular data, ask questions like, "Is this the data I really want to ask about?" These anecdotes suggest that it may be possible to train developers to be more objective and careful about their debugging efforts by using the tool. An interesting research question is whether such strategies would then persist, even if the Whyline was not available, and whether such strategies are the same strategies that skilled developers use.

While we have not discussed our experimental methods in detail here (such details appear in other publications, including Ko and Myers [2008, 2009] and Ko [2008]), it is worth mentioning that both of these human subject studies included small samples of developers and were quite limited in the range of bugs to which the Whyline was applied. Further studies of the Whyline on more diverse sets of faults would be necessary to make broad claims about the general effectiveness of the Whyline on real-world debugging tasks.

5. RELATED WORK

In more than half a century of research on debuggers, there have been countless ideas of how to make program understanding easier [Ungar et al. 1997]. Earlier ideas (such as core dumps) were constrained by performance needs, limiting the type and amount of information that a developer could obtain about a program's execution. As performance became less of a concern, researchers proposed new ways of collecting data and replaying or exploring it [Lewis 2003]. However, such techniques failed to consider how users might search through such data. The Whyline is different from other tools in its ability to elicit high relevance, high precision queries from users in an intuitive manner.

One notable approach is Cleve and Zeller's Delta Debugging [2005], which, given a specification of success and failure, and successful and failing program inputs, can empirically deduce a small chain of failure-inducing events. Similar tools take successful and failing runs of a program and perform other kinds of differential diagnosis [Liblit et al. 2005; Jones and Harrold 2005]. These approaches are quite powerful, but limited to circumstances where the success is simple to specify and possible to demonstrate (typically in situations where a new version of a program has regressed). These techniques could be integrated with the Whyline to provide higher precision answers.

Researchers have explored question-asking tools in other application domains. The ACT-R cognitive framework [Bothell 2004] and cognitive tutoring tools that used this framework [Aleven et al. 2006] both support "why not" questions about production rule systems. AI knowledge base systems support

"why not" questions about why certain data was not used in answering queries to a knowledge base [Chalupsky and Russ 2002]. Lieberman explored "why" questions about e-commerce transactions [Wagner and Lieberman 2003]. The Whyline concept has directly inspired projects looking at one-way constraints in user interface design [Clark et al. 2007], as well as spreadsheets [Abraham and Erwig 2005], where a developer can choose a wrong value, specify the correct value, and receive change suggestions that would cause the program to compute the desired value. These spreadsheet suggestions are feasible because of the limited domain of spreadsheet functions and the functional aspect of spreadsheet languages. It remains to be seen if such change suggestions are feasible (or even useful) for more complex imperative languages. Our own Crystal framework [Myers et al. 2006] explored question support in a word processor. Another budding area of research is in helping to understand failures in software that uses machine learning and AI techniques, since the indeterminacy and data set dependencies of such software can be difficult to explain [Tullio et al. 2007].

The Whyline is also related to work on static and dynamic program slicing [Weiser 1982; Baowen et al. 2005; Korel and Laski 1988], which the Whyline employs in many of its answering algorithms. The Whyline is less a competitor to these approaches and more of a consumer of them. It provides a more reliable way for users to select inputs to these techniques, providing queries of higher relevance than if users chose their queries unassisted. Furthermore, the Whyline is easily capable of taking advantage of advancements in slicing techniques [Sridharan et al. 2007; Zhang and Gupta 2005]. Slicing is also related to other work in feature-location tools, which have similar goals to the Whyline. For example, Eisenberg and De Volder discuss a test-case approach to helping users identify portions of a system relevant to a particular behavior [Eisenberg and De Volder 2005].

The Whyline would integrate well with tools that address other difficulties in bug fixing. For example, work on capturing failures in the field [Clause and Orso 2007] could be used to automatically produce a Whyline trace. Bug reports could then contain replicas of failures observed directly by users, which could then be shared, annotated, and analyzed, not only leading to faster debugging, but also providing a form of institutional knowledge that could then be mined for other issues. There may also be useful analyses from tools that focus on diagnosing particular kinds of failures, such as data structure integrity and threading issues [Lencevicius et al. 2003; Potanin et al. 2005]. The challenge will be to adapt the Whyline's approach to queries to support these techniques' input requirements.

6. LIMITATIONS

While the Whyline approach has a number of benefits, it also has several inherent limitations. We discuss them here.

6.1 Program Quality affects Question and Answer Quality

Because the Whyline extracts all of the knowledge about a program from the program itself, any limitation on the knowledge encoded in a program limits

a Whyline's utility. For example, the Whyline uses identifiers in the code to phrase questions, hence, if the quality of the identifiers is low, the quality of the question phrasing will be low. In an organizational context, this dependency may have interesting social side-effects. By making class and field names inherently public to the rest of the software development organization, the Whyline could incentivize more descriptive names for code constructs. It could also incentivize descriptive comments for fields and classes, which could be extracted from source code and shown to Whyline users to help them choose appropriate questions.

Another issue is the degree to which the concepts defined in a program faithfully represent the domain they represent (what might be called "type fidelity"). For instance, the Java Whyline relies heavily on Java's object-oriented and statically-typed nature. Object-orientation compels developers to declare classes and fields that separate distinct behaviors and state in these classes. The Java Whyline relies on this conceptual organization to provide conceptually organized menus of questions. There are a number of language paradigms that compel different forms of conceptual organization of domain concepts. For example, a procedural C program that nevertheless supports GUI components like buttons and menus, may not have a collection of clearly defined structures to render the components. Instead, there may be parameterized procedures for doing so, and a Whyline would have to do extra work to identify and organize these procedures before extracting questions from them.

6.2 Call Graph Precision Affects "Why Didn't" Answer Precision

The Whyline relies on static type information in order to extract, present, and answer "why didn't" questions. Therefore, dynamically-typed programming languages pose unique challenges for Whyline answers. For example, one challenge is building a precise call graph. Dynamically-typed languages such as Javascript are most problematic: even with runtime data, we cannot find all possible calls to a method without being conservative and losing precision. This makes it more difficult for the Whyline to answer "why didn't" questions precisely. Even statically-typed late-binding languages like Objective-C pose problems: when analyzing why an instruction did not execute, it is necessary to know all of the feasible callers to a particular method. Another problem is if the call graph is incomplete: if the Java class containing the invocation that needed to be called was never loaded, the call will not be known and will not be part of the Whyline's answer. This can be mitigated by actively loading referenced classes, but traversing too many levels in such a call graph becomes impractical.

One side-effect of imprecise call graphs is the difficulty of identifying data structures that indirectly render or affect primitive-level output. For example, it would be difficult for a Whyline to determine precisely what variables could affect output produced by a JavaScript function, unless there was runtime information to detect such data dependencies. Even then, this would not reveal other possible data dependencies about which a user might want to ask negatively phrased questions. The consequence of this imprecision is that it may be

difficult to identify good names to use in "why didn't" questions, since so many different functions may be relevant.

6.3 Limitations of Tracing

Execution traces pose several limitations. It is not practical to record executions that span more than a few minutes because the amount of data captured is too much to load and process in a reasonable amount of time. Programs and test cases that process and produce substantial amounts of data also pose a similar problem, since so much intermediate state is captured in the process (of course, what is feasible depends highly on the context: if a bug is particularly difficult to find, it may be worth the time and space necessary to capture and analyze a trace). Tracing also limits Whyline support to program behavior that is reproducible while probing, meaning that nondeterministic multithreaded bugs may not be reproducible while tracing. Programs that rely on real-time behavior may also behave differently when instrumented, making it difficult to reproduce a problem that relies on real-time performance.

Tracing also makes the approach feel "heavier" than tools like breakpoint debuggers, which require virtually no set-up time compared to the time spent waiting for a Whyline trace to load. All of these issues are worsened by the fact that the memory demands on a developer's machine grows with the size of the trace. At a certain size, performance becomes an issue as the Whyline begins to rely on virtual memory. Better disk bandwidth would alleviate this. Also, there may be ways to utilize multicore or distributed CPUs to provide dedicated support for trace capture and processing. Another possibility is that there may be ways to only trace at certain times, like today's performance profilers; the challenge would be that the causes of events might not be captured, even if the effects were.

6.4 Limitations of Question Extraction

By relying on a program as the source of questions, the Whyline will rarely perfectly match the questions that the user has in mind. People will phrase questions differently than the Whylines and there will be other contexts that the user may wish to specify in a query that the Whyline may not support (such as questions relative to multiple objects, as in "why didn't the Menu appear next to the Button?"). Furthermore, the Whyline will not always describe output at the level of granularity that the user thinks of it. If the Whyline cannot extract a "Button" concept from the program's code, the user will have to ask about whatever concept the Whyline is able to find as a proxy for "Button." The consequence of these limitations is that users will have to learn about how the Whyline extracts questions in order to know what questions to expect and where to find them.

There are other aspects of program output that are at a higher level of abstraction than an output primitive, but may not have any corresponding program entity to represent it. Imagine a menu with a list of items with varying degrees of padding between items in the list and the margins of the menu boundaries. To ask about the whitespace in the menu if the whitespace was

defined internally through constants, we would have to just ask about the position of one of the item's text labels and hope that it was *related* to the whitespace the user wants to ask about (another option is to add domain-specific support for such concepts, as in Crystal [Myers et al. 2006]).

6.5 Limitations of Answers

The most important limitation of Whyline answers is that they only reason about causality: they do not offer change suggestions, they do not isolate bugs, and they will not guide the user in interpreting the consequences of the answer to the debugging problem. All of these things are still the developer's responsibility. Therefore, although the Whyline will help users get closer to a fix than they would have otherwise, users must still objectively and systematically explore the answers provided by the Whyline in order to find and fix a bug. The reason for this limitation has less to do with the Whyline approach and more to do with the notion of a "bug." In truth, bugs are just undesirable program behaviors; even a program crash can be an expected and desired behavior under the right circumstances, for example when it prevents a more serious failure from occurring. As a consequence of this notion of a bug, any undesirable behavior has many possible solutions. The Whyline has no knowledge about the desirability of these various solutions, and so it is still up to the developer to decide what program behavior needs to change. The only kinds of techniques that can get around this problem are those that rely on specifications of a program's intended behavior, such as model-checking systems, because they can compare *intended* execution with *actual* execution.

The kinds of answers that the Whyline gives have their own limitations. The causal answers provided in response to "why did" questions and some "why didn't" questions can be quite large, since they are based on dynamic slicing. What determines how "large" these answers are depends on several factors. The more complex the program design and execution, the more complex the Whyline answer. The more users explores causality, the more information they will have to consider in assessing the cause of a problem. As they explore the answers, users will have to work around incompleteness in the Whyline's recording. Native calls and other procedures may not be amenable to instrumentation or even available for static analysis. This will result in a loss of precision for reasoning about the inputs and outputs of these calls, so Whyline prototypes will have to support ways for users to know when such information is missing and help them work around it. In general, the size of the answers was not a serious issue in user studies of the Whyline [Ko and Myers 2009], but these tests were limited to only two tasks.

There are also situations in which Whyline answers can lead to dead ends. There are at least two kinds, both having to do with developers' navigation of the control and data dependencies. First, if the Whyline has not instrumented some function and therefore cannot reason about it precisely, a developer must understand the function from its code rather than its execution. The other kind of dead end is where the Whyline answer *does* contain the relevant events, but the developer does not perceive them to be relevant. In these cases they

may overlook a relevant chain of causality. This did occur in some cases in a lab study of the Whyline [Ko and Myers 2009], although these developers eventually returned to the unexplored dependencies after exhausting other possibilities.

7. FUTURE WORK

Despite limitations inherent in its approach, the Whyline opens several new research opportunities and questions.

7.1 Real-Time Debugging

A natural extension of the postmortem version of the Java Whyline is to support debugging of a live Java program, without having to quit the program and load a recreation of its output. Aside from the challenge of tracking dynamic dependencies in real time, one significant challenge with this is in allowing the developer to point to output in the running application and have the Whyline actually relate it to live objects in the Java heap. To do so would require the Whyline to do the same I/O tracking that is done when a Whyline recording is being loaded, but instead doing it at runtime, thus incurring potentially significant overhead. One way around this problem is to have special toolkit support. For example, we could imagine a JVM debug mode which maintains a history of output, providing a hook for whatever debugging tool wanted to relate output to an execution history.

7.2 Other Output Modalities

Current Whyline prototypes only support questions about graphical and textual output, but there are many other popular forms of program output, including sound, network traffic, disk activity, and others. The central challenges in supporting these other modalities is in (1) finding effective ways to inquire about features of the the output and (2) choosing the appropriate output primitives. For example, what characteristics of sound are important to interrogate: just the presence or absence of sound, or detailed properties of its pitch and modulation? Writing to disk can often entail large amounts of data and we often only notice a failure in output after a file is completely written. How can tools effectively present this output in a manner that makes it easy to find the subject of the question amid so much information? It is possible that there could be a visible proxy for such modalities.

7.3 Collaboration Support

Collaboration support is a central design challenge for future Whyline prototypes. Debugging is an inherently collaborative activity in most software development contexts [Ko et al. 2007], requiring the knowledge of multiple people over the course of many days or more. This places a number of constraints on the Whyline design.

First, Whyline traces should be small and easy to share. This allows traces to be shared along with bug reports and other software development artifacts. Related to this support is the need to annotate a Whyline trace, as developers

discuss a program failure and move towards a solution. These annotations might relate to particular events in a trace or particular parts of the software's code. It is also possible that a Whyline trace could replace the modern notion of a "bug report," capturing information about who the trace is assigned to, along with a discussion of other aspects of the trace. It may also be possible to define a bug report as a collection of traces, all potentially demonstrating the same failure.

Another issue, unexplored in the Java Whyline, is the problem of *versions* of code. The current prototype just records the Java classes, but without any notion of which version of each class is stored in a version control system. Version information will be important in relating the Whyline trace to a particular bug report, which is typically related to a particular release. By explicitly relating a failure to a particular set of versioned source files, there may be other opportunities to detect the same failure in other versions of these source files as well. The Whyline could also integrate well with unit testing systems that explore changes to source that lead to unit test failures [Xie and Notkin 2007].

The notions of familiarity and ownership used in the Java Whyline might need to be more elaborate in collaborative settings. For example, the current definition defines familiarity by access to an editable source. Software development teams typically have much more complicated notions of ownership and certainly of familiarity [LaToza 2006].

7.4 Integration with Other Techniques

The Whyline concept utilizes modifications of a number of well-known software engineering techniques including static and dynamic slicing and other methods for constructing and analyzing call graphs and data-dependence graphs. There are, however, a number of other techniques that could be useful to integrate into the Whyline approach.

One challenge with navigating a Whyline's answer is that the user has little guidance beyond his own experience to know what control and data dependencies to follow. Researchers have looked at ways to provide such guidance. For example, static checkers [Bush et al. 2000; Cole et al. 2006] could provide cues about which dependency chains have the most potential problems, leading the user to fault-prone code. Another more interactive approach would be to allow the user to explicitly validate values as in the WYSIWYT testing and the fault localization approach [Ruthruff et al. 2005]. These annotations of correct and incorrect values could be propagated through the dynamic slice, highlighting contributors to incorrect values.

Another promising approach is to use multiple traces or multiple slices to identify differences in test cases, isolating the failure (the approach used by Zeller [2002], for example). Another approach would be to have the user specify multiple relevant questions related to the failure, rather than a single question; this is similar to the notion of a "chop" described in Gupta et al. [2005], but potentially easier to express.

The Whyline might also integrate well with other kinds of static verification tools [Cole et al. 2006] that apply a range of static checks as heuristics to

find common problems with code. Rather than applying these checks in batch mode, the Whyline might be a more helpful context in which to make these checks, using the slice as a filter on which code to analyze. Not only would such information be more helpful contextually, but it may reduce the number of false positives that the systems report because there would be much more dynamic data for the system to use in its static checking. Conversely, the Whyline user interface might be used to help explain the answers that static verification tools provide.

8. CONCLUSIONS

Debugging remains one of the most challenging aspects of software engineering, partly because today's tools require users to speculate about the causes of program behavior. We have presented an entirely new way to query program output, allowing the user to obtain evidence about the program's execution before forming an explanation of the cause. In the future, we hope this will inspire explorations of the limits of our approach for both Java and also for other languages and computing architectures.

ACKNOWLEDGMENTS

We thank Bonnie John, Gail Murphy, and Jonathan Aldrich for their comments on the first author's dissertation, which served as the basis of much of this article.

REFERENCES

ABRAHAM, R. AND ERWIG, M. 2005. Goal-directed debugging of spreadsheets. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE, Los Alamitos, CA, 37–44.

ALEVEN, V., MCLAREN, B. M., SEWALL, J., AND KOEDINGER, K. 2006. The cognitive tutor authoring tools (CTAT): Preliminary evaluation of efficiency gains. In *Proceedings of the International Conference on Intelligent Tutoring Systems*. 61–70.

BAOWEN, X., JU, Q., XIAOFANG, Z., ZHONGQIANG, W., AND LIN, C. 2005. A brief survey of program slicing. *SIGSOFT Softw. Engin. Notes 30*, 2, 1–36.

BOTHELL, D. 2004. *ACT-R Environment Manual*, Ver. 5.0. http://act-r.psy.cmu.edu/software/EnvironmentManual.pdf.

BUSH, W. R., PINCUS, J. D., AND SIELAFF, D. J. 2000. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper. 30*, 7, 775–802.

CHALUPSKY, H. AND RUSS, T. A. 2002. Why not: Debugging failed queries in large knowledge bases. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. 870–877.

CLARK, P., CHAW, S. Y., BARKER, K., CHAUDHRI, V., HARRISON, P., FAN, J., JOHN, B., PORTER, B., SPAULDING, A., THOMPSON, J., AND YEH, P. Z. 2007. Capturing and answering questions posed to a knowledge-based system. In *Proceedings of the International Conference on Knowledge Capture (K-CAP)*. 63–70.

CLAUSE, J. AND ORSO, A. 2007. A technique for enabling and supporting debugging of field failures. In *Proceedings of the International Conference on Software Engineering*. 261–270.

CLEVE, H. AND ZELLER, A. 2005. Locating causes of program failures. In *Proceedings of the International Conference on Software Engineering*. 342–351.

COLE, B., HAKIM, D., HOVENMEYER, D., LAZARUS, R., PUGH, W., AND STEPHENS, K. 2006. Improving your software using static analysis to find bugs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM, New York, 673–674.

COOPER, K. D., HARVEY, T. J., AND KENNEDY, K. 2001. A simple, fast dominance algorithm. http://www.hipersoft.rice.edu/grads/publications/dom14.pdf.

EISENBERG, A. AND DE VOLDER, K. 2005. Dynamic feature traces: Finding features in unfamiliar code. In *Proceedings of the International Conference on Software Maintenance*. 337–346.

GROVE, D. AND CHAMBERS, C. 2001. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst. 23*, 6, 685–746.

GUPTA, N., HE, H., ZHANG, X., AND GUPTA, R. 2005. Locating faulty code using failure-inducing chops. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. ACM, New York, 263–272.

JONES, J. A. AND HARROLD, M. J. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the International Conference on Automated Software Engineering*. 273–282.

KO, A. J. AND MYERS, B. A. 2009. Finding causes of program output with the Java Whyline. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems (CHI)*. ACM, New York, 1569–1578.

KO, A. J. 2008. Asking and answering questions about the causes of software behaviors. Dissertation, CMU-CS-08-122, Human-Computer Interaction Institute, Carnegie Mellon University.

KO, A. J. AND MYERS, B. A. 2008. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 301–310.

KO, A. J. DELINE, R., AND VENOLIA, G. 2007. Information needs in collocated software development teams. In *Proceedings of the International Conference on Software Engineering*. 344–353.

KO, A. J., MYERS, B. A., AND CHAU, D. H. 2006a. A linguistic analysis of how people describe software problems. In *Proceedings of the IEEE Visual Languages and Human-Centric Computing*. IEEE, Los Alamitos, CA, 127–134.

KO, A. J., MYERS, B. A., COBLENZ, M., AND AUNG, H. H. 2006b. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Engin. 32*, 12, 971–987.

KO, A. J. AND MYERS, B. A. 2004. Designing the Whyline: A debugging interface for asking questions about program failures. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*. ACM, New York,151–158.

KO, A. J., MYERS, B. A., AND AUNG, H. H. 2004. Six learning barriers in end-user programming systems. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE, Los Alamitos, CA, 199–206.

KOREL, B. AND LASKI, J. 1988. Dynamic program slicing. *Inform. Process. Lett. 29*, 3, 155–163.

LATOZA, T. D., GARLAN, D., HERBLSEB, J. D., AND MYERS, B. A. 2007. Program comprehension as fact finding. In *Proceedings of the Foundations of Software Engineering*. 361–370.

LENCEVICIUS, R., HOLZLE, U., AND SINGH, A. K. 2003. Dynamic query-based debugging of object-oriented programs. *J. Autom. Softw. Engin. 10*, 1, 367–370.

LEWIS, B. 2003. Debugging backwards in time. In *Proceedings of the International Workshop on Automated Debugging*. 225–235.

LIBLIT, B., NAIK, M., ZHENG, A., AIKEN, A. AND JORDAN, M. 2005. Scalable statistical bug isolation. In *Proceedings of the Programming Design and Implementation*. 15–26.

MILANOVA, A., ROUNTEV, A., AND RYDER, B. G. 2002. Precise call graph construction in the presence of function pointers. In *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation*. 155.

MYERS, B. A., WEITZMAN, D., KO, A. J., AND CHAU, D. H. 2006. Answering why and why not questions in user interfaces. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*. ACM, New York, 397–406.

POTANIN, A., NOBLE, J., AND BIDDLE, R. 2004. Snapshot query-based debugging. In *Proceedings of the Australian Software Engineering Conference*. 251.

RUTHRUFF, J. R., PRABHAKARARAO, S., REICHWEIN, J., COOK, C., CRESWICK, E., AND BURNETT, M. M. 2005. Interactive, visual fault localization support for end-user programmers. *J. Visual Lang. Comput. 16*, 1-2, 3–40.

SRIDHARAN, M., FINK, S. J., AND BODIK, R. 2007. Thin slicing. In *Proceedings of the Programming Language Design and Implementation*. 112–122.

TASSEY, G. 2002. The economic impacts of inadequate infrastructure for software testing. RTI Project 7007.011, 2002, National Institute of Standards and Technology.

TULLIO, J., DEY, A. K., CHALECKI, J., AND FOGARTY, J. 2007. How it works: A field study of non-technical users interacting with an intelligent system. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*. ACM, New York, 31–40.

UNGAR, D., LIEBERMAN, H., AND FRY, C. 1997. Debugging and the experience of immediacy. *Comm. ACM 40*, 4, 39–43.

WAGNER, E. AND LIEBERMAN, H. 2003. An end-user tool for e-commerce debugging. In *Proceedings of the International Conference on Intelligent User Interfaces*. 331–331.

WANG, T. AND ROYCHOUDHURY, A. 2004. Using compressed bytecode traces for slicing Java programs, In *Proceedings of the International Conference on Software Engineering*. 512–521.

WEISER, M. 1982. Programmers use slices when debugging. *Comm. ACM 25*, 7, 446–452.

XIE, T., TANEJA, K., KALE, S., AND MARINOV, D. 2007. Towards a framework for differential unit testing of object-oriented programs. In *Proceedings of the International Workshop on Automation of Software Test*. 202.

ZELLER, A. 2002. Isolating cause-effect chains from computer programs, In *Proceedings of the International Symposium on the Foundations of Software Engineering*. 1–10.

ZHANG, X. AND GUPTA, R. 2005. Whole execution traces and their applications. *ACM Trans. Architect. Code Optim. 2*, 3, 301–334.

Received September 2008; revised March 2009; accepted April 2009