Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior

Andrew J. Ko and Brad A. Myers
School of Computer Science, Carnegie Mellon University

Presenter: Shaosong Li
Instructor: Christoph Csallner
Agenda

• Motivation
• Prototype
• Design & implementation
• Evaluation
• Limitation
• Reference
• Q & A
Motivation

• Program understanding and debugging
• Most challenging and time consuming aspects of software development
• Takes up to 70% of the time [17]
• Little has changed in how people work
Suppose you see an inappropriate behavior

- Translate their question
- A series of queries about the program’s code
- Guess what code is responsible
- Relationship between the symptoms of a problem and cause

- 90% Initial guesses were almost wrong! [7,8]
Current tools

• Breakpoint debuggers require people to choose a line of code
• Slicing tools require a choice of variable
• Querying tools require to write an executable expression about data

Limitation

• ‘garbage-in garbage-out’
• Not allow developers to ask why not questions
New Idea: Whyline

- Allow developers to choose a why did or didn’t question
- Generate answers to the question using a variety of program analyses

http://www.cs.cmu.edu/~NatProg/whyline.html
Earlier prototypes

• Alice Whyline: [8]
  Support similar interaction technique, but for an extremely simple language with little need for procedures and a rigid definition of output

• Crystal framework: [14]
  Support questions in end-user applications, applied same ideas, but limited the scope mostly to questions about commands
Extend and Implementation for Java

• Algorithms:
  ➢ deriving questions from code
  ➢ answering questions
  ➢ visualization of answers
• Conditions:
  ➢ standard Java I/O mechanisms
  ➢ program not run too long
Experiment: A simple painting application

• 500 lines of code
• A few bugs
• One Problem:

RGB color slider did not create the right color

<table>
<thead>
<tr>
<th>Text searches for “color”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manually followed data dependencies</td>
</tr>
<tr>
<td>Using breakpoints</td>
</tr>
</tbody>
</table>
1. Demonstrate the behavior user want to inquire about
2. Whyline loads the trace, then the user finds point in time they want to ask about by moving the time controller
3. Click on output and chooses a question
4. Whyline provides an answer, which user navigates
5. In order to understand the cause of the behavior
6. When user selects an event, the corresponding source file shown
7. Call stack and locals at the time of the selected execution event
Design and Implementation

• Intend to support interactive debugging
• Develop new incremental and cache reliant algorithm
• Take a post mortem approach to debugging, capturing a trace and then analyzing it after the program has stopped
Steps & processes

1. Recording an Execution Trace
2. Loading the Trace
3. Creating an I/O History
4. Deriving Questions
5. Answering ‘Why did...’ Questions
6. Answering ‘Why didn’t...’ Questions
Recording an Execution Trace

A Whyline trace of an execution contains:

- Sequences of events that occurred in each thread
- All class files executed
- Source files that represent them
- Other meta data recorded to interpret the data in the trace
Loading the Trace

a. Source files and class files are loaded
b. Generate a precise call graph, using all of the invocations found in class files.
c. Loader reads the thread traces, loading events in order of their event IDs
Creating an I/O History

• Prototype assumption: Program uses standard Java I/O interfaces

<table>
<thead>
<tr>
<th>Java I/O interface</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>java. awt. Graphics2D</td>
<td>Graphical output</td>
</tr>
<tr>
<td>java. awt. Window</td>
<td>Represent windows</td>
</tr>
<tr>
<td>java. io. Writer, OutputStream, PrintStream, Reader, InputStream</td>
<td>Console and file I/O</td>
</tr>
<tr>
<td>java. lang. Throwable</td>
<td>Exception output</td>
</tr>
</tbody>
</table>
Whyline handles output instructions gathered during class loading. After parsing output, Whyline finds fields and invocations could have affected output.

Insight:
Particular code is responsible for particular output.

Figure 2. Algorithms `markAffectors` and `markInvokers`, which mark methods and fields that affect or invoke output.
Deriving Questions

- Why did questions refer to a specific event from a trace
  - Depend on the input time selected by the user (event)
  - Actual depends on the type of output

- Why didn’t questions refer to one or more instructions in the code
  - About variables (discrete-value, other types)
  - About output has no representative output to click on
Figure 3. The question hierarchy 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 []’s represented menus of the specified type) and the content on the right gives an example screen shots.
1. Why did questions relate to the properties of the user selected
2. Why did questions about each of the familiar, output affecting field’s current values
3. Why these fields were not assigned after the selected time
4. Questions about objects that indirectly invoked the selected output primitive, including questions about the creation of the object
5. Questions about output-invoking methods that the user believes did not execute
6. Questions about output that has no representative output to click on
Answering ‘Why did...’ Questions

• Each question maps to an event that is analyzed using dynamic slicing techniques [2], including control [5] and data dependencies.

• Produces a causal chain of events, this chain is the tree of events that are traversed in a typical dynamic slicing algorithm [2].

• Each event’s control and data dependencies are computed on demand when a user selects an event. Which means answers are almost produced immediately.
Answering ‘Why didn’t…’ Questions

• Whyline handles each of the instructions referred to by the question individually

• Explain individual instruction --- two analyses
  ➢ Determining why an instruction was not executed
  ➢ Determining why a particular dynamic data dependency did not occur

• Constrained by two types of scope
  ➢ Temporal scope: consider events
  ➢ Identity scope: consider objects
Example 1 Why was this instruction not executed?

- First, check if it did execute
- If the instruction did not execute, the Whyline uses an algorithm (which be called whynotreached) to explain why
- Result of the algorithm is a directed graph, with nodes consisting of invocations and conditional instructions that were not reached
- Nodes also have a causal chain of event attached, explaining the source of the wrong object, or the values of the conditional’s expression
Example 2 Why was this value not used?

- Compare the expected dynamic dependency path to the actual dynamic dependency path at runtime
- Following code, which controls a text field’s background based on various state.

```plaintext
draw()
1    color = getBack()
2    setColor(color)
3    fillRect()
4    setBack(newColor) color = newColor
5    getBack() return color
determineColor()
6    if(invalid)
7    if(enabled)
8    setBack(red)
9    else
10   setBack(gray)
11   if(override)
12   setBack(black)
```
\texttt{whynotvalue}(\textit{list of instructions expected}, \textit{list of events actual})

co-iterate through \textit{expected} and \textit{actual}, comparing
instructions and finding point of \textit{deviation}

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

if \textit{exp} executed within temporal scope
  if all of \textit{exp}'s executions occurred before \textit{act}
    \textit{reason} = \texttt{value was used, but then overridden}
  else \textit{reason} = \texttt{value was used}
else \textit{reason} = \texttt{whynotreached(next_exp)}

\textbf{Figure 5.} Algorithm \texttt{whynotvalue}, which explains why a certain
dynamic data dependency did not occur.
Presenting answers

• As a navigational aid (help users understand the relationship between events that occurred at runtime and the code that caused them)
Keep the visualization compact
Focus the developer on the code
The Whyline hides information that the user would find unfamiliar or irrelevant.

Figure 7. An answer showing (1) a collapsed invocation, (2) a hidden call context, (3) several instructions not executed and (4) a conditional that evaluated in the wrong direction, preventing the desired instruction from executing.
Provide simple navigational model for exploring an answer
Evaluation -- Performance Feasibility

1. Slowdown (comparing normal running time to tracing time)
2. Trace Size
3. Compressed Trace Size
4. Trace Loading Time
Table 1. Statistics about tracing slow down, trace size with and without compression, and trace loading time, on five open source Java programs, averaged over ten runs. The profiling times were computed using the YourKit Java profiler with tracing mode on (rather than sampling). Lines of code for each program were computed omitting whitespace lines.

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Test case</th>
<th>Execution time (sec)</th>
<th>Slowdown (ratio)</th>
<th># of events</th>
<th>Trace Size (mb)</th>
<th>Loading (sec)</th>
<th>events/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>normal</td>
<td>YourKit</td>
<td>Whyline</td>
<td>YourKit</td>
<td>Whyline</td>
<td></td>
</tr>
<tr>
<td>BinClock</td>
<td>177</td>
<td>Run clock for five seconds</td>
<td>5.7</td>
<td>9.1</td>
<td>9.8</td>
<td>1.6</td>
<td>1.7</td>
<td>140,268</td>
</tr>
<tr>
<td>JEditor</td>
<td>12,258</td>
<td>Clean html of NY Times front page</td>
<td>0.9</td>
<td>3.9</td>
<td>13.8</td>
<td>4.3</td>
<td>15.3</td>
<td>16,504,866</td>
</tr>
<tr>
<td>JEditor</td>
<td>66,403</td>
<td>Load, open file, type “Goodbye”, quit</td>
<td>8.4</td>
<td>11.7</td>
<td>60.1</td>
<td>1.4</td>
<td>7.2</td>
<td>8,983,890</td>
</tr>
<tr>
<td>Javac</td>
<td>54,054</td>
<td>Compile 2,810 line Java source file.</td>
<td>2.0</td>
<td>3.7</td>
<td>17.0</td>
<td>1.9</td>
<td>8.5</td>
<td>35,193,667</td>
</tr>
<tr>
<td>ArgoUML</td>
<td>113,117</td>
<td>Load to splash screen and quit</td>
<td>5.60</td>
<td>15.00</td>
<td>28.6</td>
<td>2.7</td>
<td>5.1</td>
<td>18,303,691</td>
</tr>
</tbody>
</table>

**Subject programs**

1. Binary clock (bincloce)
2. Command line HTML formatter (JTidy)
3. Java compiler (javac)
4. Text editor (jEdit)
5. Diagramming tool (ArgoUML)
Evaluation -- Question Coverage

The degree to which users would be able to find a question that matches the question they want to ask

Approach
1. Randomly sample bug reports on the five applications in table 1
2. Check to see whether any question seemed like a reasonable translation
3. If so, how much translation was required
<table>
<thead>
<tr>
<th>Program</th>
<th>Bug Report Title - Description</th>
<th>Whyline Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>jTidy</td>
<td>Again DOM Parsing error - [error message listed in report]</td>
<td>Why did [error message] print?</td>
</tr>
<tr>
<td></td>
<td>JTidy allows duplicate ID attributes - If you give the same ID value, should cause error...</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>JTidy locks up in a never ending loop - it locks up with this content:...</td>
<td>Why didn't [success message] get printed?</td>
</tr>
<tr>
<td>jEdit</td>
<td>Soft wrap, cut and null-pointer exception - This results in a BeanShell error dialog...</td>
<td>Why did this text = [error dialog text]?</td>
</tr>
<tr>
<td></td>
<td>File Open/Save dialog's directory - File/Save dialog should start in the directory last selected</td>
<td>Why did this text = [current folder name]?</td>
</tr>
<tr>
<td></td>
<td>Invalid screen line count - java.lang.RuntimeException: Invalid screen line count: 0...</td>
<td>Why did [exception thrown] occur?</td>
</tr>
<tr>
<td>ArgoUML</td>
<td>Autosize triggers at wrong times - stretch any class to greater than it required size...</td>
<td>Why did this [class's] rectangle width = [wrong size]?</td>
</tr>
<tr>
<td></td>
<td>Invisible FigNodes are being saved - software just displays error and doesn't open project</td>
<td>Why did this text = [error dialog text] ?</td>
</tr>
<tr>
<td></td>
<td>Can not parse import statement after javadoc comment - unexpected token &quot;import&quot;...</td>
<td>Why did this text = [error dialog text]?</td>
</tr>
</tbody>
</table>

**Whyline questions**

2 were about console output
5 were about primitive graphical output
1 was about an exception
1 could not find a suitable question
Evaluation – User Study
Experiment: A simple painting application

• 500 lines of code
• A few bugs
• One Problem:
  RGB color slider did not create the right color

• Participants had a variety of backgrounds
• Compared with those people’s task performance with that of 18 self-described expert Java developers from a prior study [10]
Result

<table>
<thead>
<tr>
<th>Time</th>
<th>Whyline Participants</th>
<th>Control Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Median(min)</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>Range(min)</td>
<td>1--12</td>
<td>3--38</td>
</tr>
</tbody>
</table>

Why is the line blue?

Speculate possible reasons
Limitations

1. Trace based approach
2. Knowledge about a program
3. Programming language
4. Only find code related to a behavior
QUESTIONS
&
ANSWERS