Querypoint : Moving Backwards on Wrong Values in the Buggy Execution

Salman Mirghasemi
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
salman.mirghasemi@epfl.ch

John J. Barton
IBM Research - Almaden
bartonjj@us.ibm.com

Claude Petitpierre
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
claude.petitpierre@epfl.ch

ABSTRACT

Developers often seek the origins of wrong values they see in their debugger. Their search must be backwards in time: the code causing the wrong value executed before the wrong value appeared. Searching with breakpoint- or log- based debuggers demands persistence and significant experience with the application being debugged.

Querypoint, is a Firefox plugin which enhances the popular Firebug JavaScript debugger with a new, practical feature called *lastChange*, which automatically locates the last point that a variable or an object property has been changed. Starting from a program halted on a breakpoint, the *lastChange* solution applies queries to the live program during re-execution, recording the call stack and limited program state each time the property value changes. When the program halts again on the breakpoint, the recorded information can be shown to the developer.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Debugging aids; D.2.6 [**Programming Environments**]: Integrated environments

General Terms

Algorithms, Human Factors, Languages

Keywords

Debugging, Locating Defects, Querypoint, LastChange, Breakpoint, Watchpoint, Logging

1. INTRODUCTION

According to [10], developers spend about fifty percent of their time debugging. To fix a bug, developers typically reproduce and monitor the buggy execution several times to understand the program's unexpected behavior. Trial-and-error, guess-work, and analyzing complicated data make debugging difficult and time-consuming. Enhanced debugging operations save time, reduce development costs and improve software quality.

A common strategy for locating defects starts from bug symptoms and works backwards, moving from a point in the program execution where a value appears to be incorrect back to the point where that value was set. Two conventional approaches, breakpoint-based and log-based debugging, require tedious steps of selecting data to be collected, collecting the data, then analyzing the results.

In breakpoint debugging, developers select data to be collected by searching through source files and setting breakpoints. To determine where a value was set incorrectly, a developer must set breakpoints at all possible points where the value changes. At every breakpoint, the developer must determine if the location is in fact related to the questionable value change then study the complex debugger user interface and memorize values or manually collect data. As the number of breakpoint hits increases, the process of checking the program state, collecting data and resuming the execution becomes cumbersome.

In log-based debugging, developers select data to be collected by inserting statements for all points of possible change. While in breakpoint-based debugging, the whole program state is available to developer, in log-based debugging, developer has to decide what data should be collected when inserts the log statement. It is very common that the developer has to repeat this step several times due to insufficient collected data, or to wait a long time because too much data is recorded. Once adequate data is collected, it still requires analyzing and understanding. Developers usually end up in dealing with long log files and analyzing huge amounts of collected data. Neither approach effectively assists the developer in finding origins to a wrong value.

Our new functionality in debuggers, *lastChange*, locates the origin of a wrong value by queries on the running program. Imagine that a program execution is paused on a breakpoint and the developer is suspicious about the value of a variable or an object property. The developer selects *lastChange* on the value. The debugger replays the buggy execution and collects data when the data field changes. Once the execution reaches the same place (i.e., the same breakpoint hit), it pauses the execution, analyzes the collected data and shows the location of the last change to the developer. The developer can also examine the program state at the located point of execution, and continue debugging by more *lastChange* queries from that point.

Our contribution in this paper is the technique *lastChange*, which locates the last place a value has changed, gathers other values from that execution point, and allows *lastChange* operations from that point. The technique builds on existing breakpoint debugger technology and it does not require a special environment to create identical, instruction by instruction, re-executions. We demonstrate the feasibility of the approach with *Querypoint*, an implementation extending Firebug JavaScript debugger. *Querypoint* also provides mechanism for automated bug reproduction, and a novel user interface which summarizes investigated execution points and collected results. The *lastChange* algorithm provides information on important program values during the program execution without voluminous logs and without tedious insertion and removal of breakpoints. We believe other queries over the running program can be formulated to generalize this technique.

2. RELATED WORK

The *lastChange* approach resembles the operational model of replay-based debugging and the query approach of logging-based debugging. Replay-based approaches capture limited data during execution and replay the buggy execution to reach past points. In contrast, logging-based approaches collect enough data during execution to relieve developer from re-execution, then query the data to inform the developer. Replay-based approaches impose much less runtime overhead (about two orders of magnitudes) comparing to logging-based approaches. However, developer has to re-execute the buggy execution several times. *lastChange* collects data on re-execution by queries selected by developer interaction with the debugger. Therefore it has the selectivity of the replay-based approaches that improves performance, and the flexibilty of the queries so it does not require deterministic replay.

Among replay-based debuggers we compare to bdb [4] and reverse watchpoint [13]. A bidirectional C debugger, bdb employs a step counter to locate the requested point from the beginning of execution. It relies on deterministic execution replay and records the results of non-deterministic system calls and re-injects them into the program when it is replayed. It makes use of checkpoints to reduce the time needed for re-execution. Reverse watchpoint, similar to bdb, uses a counter to correctly locate the last write access of a selected variable in the next execution [13]. The main disadvantage of these approaches is that they require identical, instruction by instruction, re-executions. Even one instruction difference between two executions leads to wrong results. On the other hand, *lastChange* doesn't require any special feature in the re-execution and fits into existing debugging practice

Among logging-based approaches are *omniscient* debuggers ODB [11] and Unstuck [5]. Both approaches keep the log history in memory and hence can only record and store the complete history for a short period of time. These debuggers record all the events that occur during the buggy execution and later let the developer to navigate through the obtained execution log. In these approaches there is no execution to resume: moving backwards in the log can be similar to moving forwards. A more scalable approach to omniscient debugging has been proposed by Pothier et al. [15]. Their back-in-time debugger, TOD, addresses the space problem by storing execution events in a distributed database. Bhansali et al. [2] attempted to address the performance overhead and large trace size of logging-based debugging in their time-travelling debugger, Nirvana. Nirvana first collects a full compressed trace of program execution and then simulates re-execution. Their results are quite similar to omniscient debuggers. Nirvana incurs about 15 to 68 times runtime overhead in re-simulation.

Logging-based debuggers suffer from different issues. First, the recording step is time expensive and it should be repeated in case of changes in program. Second, the execution log cannot fully replace the live execution. There are other aspects of execution (e.g., program user interface, file system, database tables, etc.) which are also important in debugging and are not available to the developer in logging-based debuggers. Third, querying collected data (e.g., to restore the program state at a certain point) may not be efficient enough for debugging realistic programs. Comparing to logging-based debuggers our approach has little upfront cost and more flexibility. The developer can start debugging just after reproducing the bug without a capturing step. Changing inputs or environment settings and re-executing to investigate the bug works as in conventional breakpoint debuggers.

Program slicing [17] approaches debugging from a completely different perspective. Given a point of interest **S** in a program, a *slice* is an executable subset of the program affecting **S**. Many studies have shown ways to compute the slice more quickly or to constrain the slice to a smaller subset of the program [6, 16]. Dynamic slicing [9] applies slicing analysis at run time and thus most closely resembles *lastChange* from among the slicing approaches. Our *lastChange* approach can be related this way: first, allow the developer to select from **S** a key critical value **V**. Then compute a slice affecting **V**. Show the developer only the most relevant part of that slice (the *lastChange*). Finally iterate towards the fault by chaining new slice-selection criteria.

Both approaches rely on selecting **S**. Both approaches differ from breakpoint debugging in analyzing a program from a known good point (e.g. start of the program) to **S**. Slicing reduces the space the developer needs to search, but provides no guidance within the slice; *lastChange* explores the slice using developer-selected search criteria, but does not help the developer recognize code that cannot impact **S**. Slicing requires a sophisticated compiler-like technology while *lastChange* builds on familiar run-time debugger technology.

A recent work by Lienhard et al. [12] suggests virtual machine level support for keeping the history of events. It replaces every object reference with an alias object which keeps the history of changes to the object reference. Although this approach incurs less runtime overhead (7 times) in comparison to omniscient debuggers, it adds memory overhead.

Origin tracking of `undefined` and `null` values employing *value piggybacking* technique proposed by Bond et al. [3]. This approach has two main limitations comparing to *lastChange*. First, it is limited to `undefined` and `null` values. Second, it does not return the last change of a `null` variable but the first place that the `null` value is originated.

WhyLine [8] stores the program user interface in addition to the program trace and provides answers to why and why not questions to the user. WhyLine suffers from the requirement of gathering tracing information before its unique capabilities can be used. We imagine that the runtime queries we use in *lastChange* may be used to gather data incrementally for this kind of debugging approach.

3. INTRODUCTORY EXAMPLE

We illustrate the *lastChange* functionality by a simple example. The example demonstrates a buggy JavaScript code in a HTML page (Figure 1). The page contains a button (line 40) showing the value of `myObject.myProperty`. When the user clicks on the button, the `onClick` function (line 13) is called. This function increases the value of `myObject.myProperty` by one (line 15) and calls `updateButton` function which updates the button's text to the new value (line 22). Once the page is loaded for the first time the button shows `1` as the initial value of `myObject.myProperty`. In practice when the user clicks on the button, `0` appears instead of `2`: there is a bug.

Two other functions are called in `onClick()`, `foo()` and `bar()`. As developers we often encounter function calls which seem peripheral to our current concern; they may have been added by another developer, or we may have forgotten their exact properties or those properties may have changed, and so on. The difference between what we expect these functions to do, e.g. nothing interesting, and what they do in practice may cause bugs.

```
1 <html>
...
5   <script type="text/javascript">
6     myObject = {myProperty : 1};
7     myCondition = {value : 1};
...
13    function onClick(){
14      foo();
15      myObject.myProperty++;
16      bar();
17      ...
18      updateButton();
19    }
20    function updateButton(){
21      var myParagraph =
            document.getElementById("myButton");
22      myButton.innerHTML = myObject.myProperty;
23    }
24    function foo(){
25        myCondition.value = oldValue;
26    }
27    function bar(){
28      if (!myCondition.value)
29          myObject.myProperty = 0;
30    }
31  </script>
...
40  <button id="myButton" onclick="onClick()">
41      1
42  </button>
43 </html>
```

Figure 1: A Web page containing JavaScript code. Some lines not related to our paper have been elided.

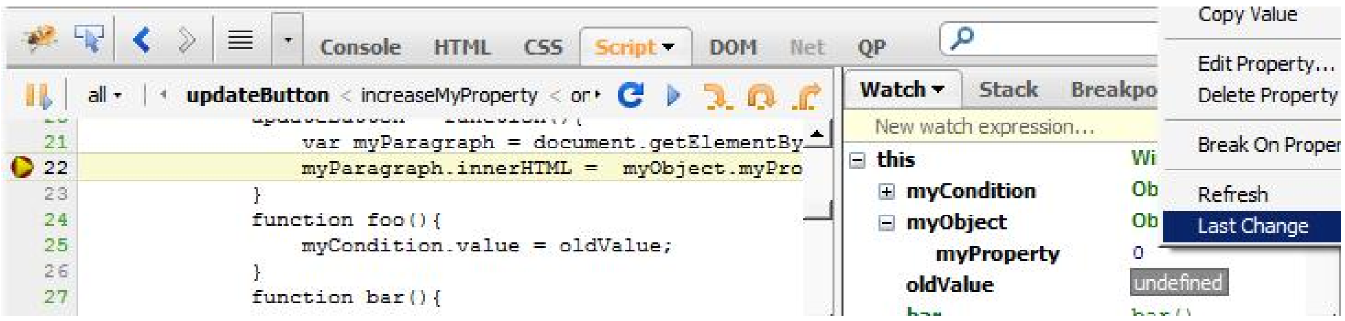
By browsing through the code or other means [1], the developer determines that the value displayed on the button is set at line 22. Since the displayed value is incorrect we know the bug occurred before we hit this line. To start debugging, the developer sets a breakpoint on line 22. Once the button is clicked, the execution is paused at line 22. Figure 2(a) shows the Firebug debugger while the execution is paused. Firebug has several panels (e.g., HTML, CSS, Script, DOM, etc.) that each demonstrate one aspect of the Web page. The Script panel contains the list of all loaded source files and regular debugging facilities such as setting breakpoints and stepping. To the right of the script panel, the Watch panel shows the program state where the developer can examine object and variable values. In our case, the `myObject.myProperty` value at the paused point is `0`. We expected this value to be `2`.

To apply backward search strategy for locating defects, the developer first needs to know the origin of the wrong value. To achieve this goal using breakpoints, the developer should search code to find all possible places that `myObject.myProperty` might get a new value and set breakpoint at these locations. However, an object and property can be accessed and changed through different names and methods. There is no simple way to identify these aliases or even their total number. The developer can make a good guess and set breakpoints on lines where the property seems to be changed. Then they re-execute the program and examine the state looking for values that may lead to the incorrect value observed at line 22. All this work must be repeated if a new alias is discovered or if some information related to the buggy result was missed while stopped on one of the breakpoints.

In contrast, we have added a high-level function in the debugger, *lastChange*, which provides the answer without tedious manual effort from the developer. By right clicking on `myObject.myProperty` in the Watch panel, the developer can run *lastChange* command (Figure 2(a)). The debugger re-executes the program and halts again at the breakpoint on line 22. However, it shows a new panel, called QP, centered on the source at line 29 (Figure 2(b)), the point of *lastChange*. To the right, the TraceData panel shows values of properties of the program state when it passed through line 29. These two panels resemble the Script and Watch panels, but they show data collected by the debugger at one execution point which is now past: these are *traces* or *logs* of information collected during the re-execution.

Looking at line 29, it seems that something is wrong with `myCondition.value` which causes line 29 execution. The developer examines `myCondition.value` and it is `undefined`. The next step is to know when this property got this value. To do so, the developer runs the *lastChange* command on `myCondition.value` at this point. The debugger re-executes the program and breaks again on line 22, analyzes its queries and shows the developer line 25-the place `oldValue` is assigned to `myCondition.value`. If the developer asks for *lastChange* on `oldValue`, the debugger can notify the developer that this variable is never assigned a value. Now it is clear that the bug occurs because `oldValue` is `undefined` once the execution reaches line 25 (Figure 2(c)).

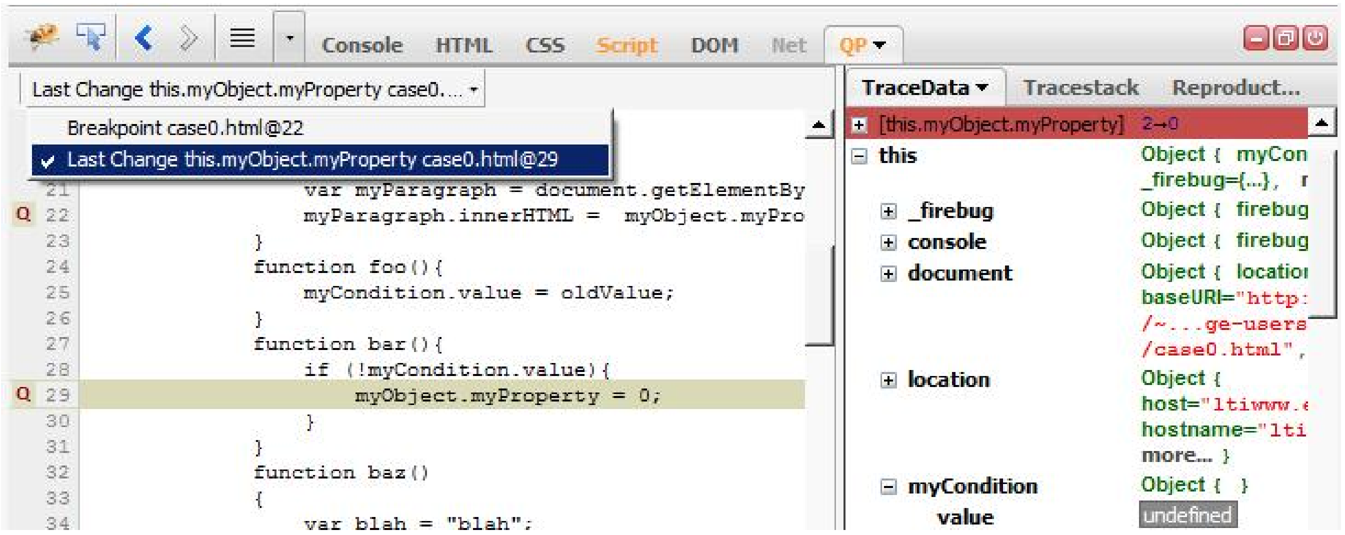
As demonstrated in Figure **??**, the developer has examined three points of execution. The first point was the breakpoint set by the developer. We call this special breakpoint the *reproduction point*. The second and third points preceded the reproduction point in execution sequence. All three points-the history of the search for the defect-are available through the debugger's interface. On the top of the left panel in Figure 2(c) there is an opened list which shows all three examined points. The first one is the breakpoint on line 22, the second one is the point which is when `myObject.myProperty` changed before reaching the breakpoint and finally the last one is the point of execution in which `myCondition.value` gets the `undefined` value. Moreover, the source lines related to these points are marked with red



```
21            var myParagraph = document.getElementBy
22            myParagraph.innerHTML =  myObject.myPro
23        }
24        function foo(){
25            myCondition.value = oldValue;
26        }
27        function bar(){
```

```
New watch expression...
⊟ this                 Wi
  ⊞ myCondition        Ob
  ⊟ myObject           Ob
      myProperty       0
    oldValue           undefined
```

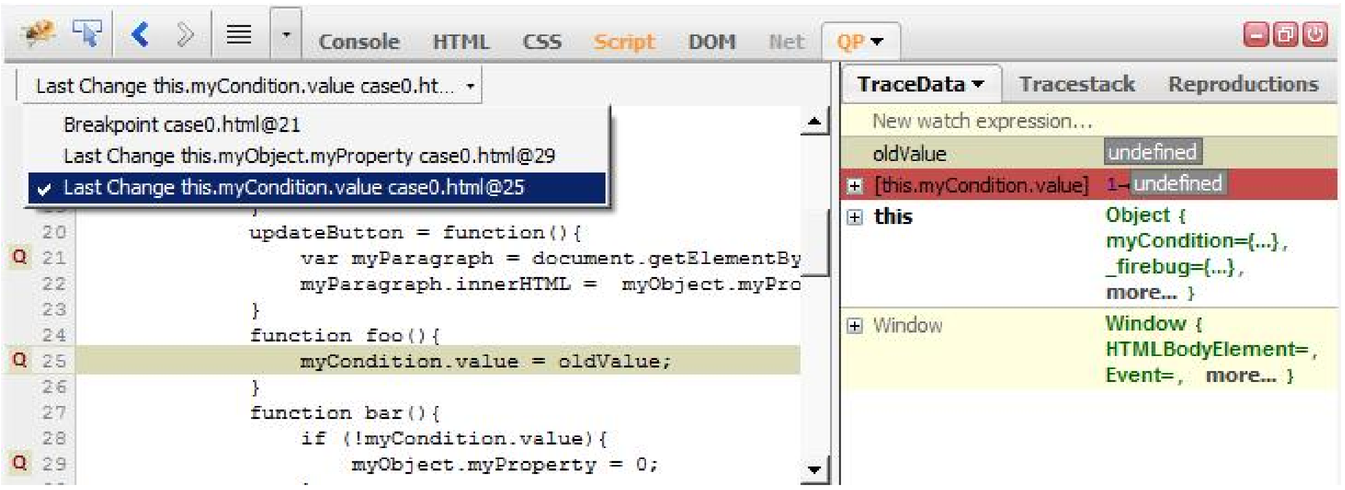
(a) A screen shot of the Firebug debugger while running the example code from Fig. 1. The Script panel is selected; it gives access to all loaded source files and allows breakpoints to be set on lines. In this figure, the execution is paused at line 22 by a regular breakpoint. The Watch panel on the right shows the program state at the paused point. Developer can query *lastChange* on `myObject.myProperty` by right-clicking on the value of `myProperty`.


Console
HTML
CSS
Script
DOM
Net
QP
TraceData
Tracestack
Reproduct...
Last Change this.myObject.myProperty case0....
Breakpoint case0.html@22
✓ Last Change this.myObject.myProperty case0.html@29

```
21            var myParagraph = document.getElementBy
22            myParagraph.innerHTML =  myObject.myPro
23        }
24        function foo(){
25            myCondition.value = oldValue;
26        }
27        function bar(){
28            if (!myCondition.value){
29                myObject.myProperty = 0;
30            }
31        }
32        function baz()
33        {
34            var blah = "blah";
```

```
⊞ [this.myObject.myProperty]  2→0
⊟ this              Object { myCon
                    _firebug={...},  r
  ⊞ _firebug        Object { firebug
  ⊞ console         Object { firebug
  ⊞ document        Object { locatioı
                    baseURI="http:
                    /~...ge-users
                    /case0.html",
  ⊞ location        Object {
                    host="ltiwww.
                    hostname="lti
                    more... }
  ⊟ myCondition     Object {  }
      value         undefined
```

(b) The result of *lastChange* query for `myObject.myProperty`. The left panel, QP, shows the source code at the point of *lastChange*; The right panel, TraceData, shows the collected data at the point.


Console
HTML
CSS
Script
DOM
Net
QP
TraceData
Tracestack
Reproductions
Last Change this.myCondition.value case0.ht...
Breakpoint case0.html@21
Last Change this.myObject.myProperty case0.html@29
✓ Last Change this.myCondition.value case0.html@25

```
20        updateButton = function(){
21            var myParagraph = document.getElementBy
22            myParagraph.innerHTML =  myObject.myPro
23        }
24        function foo(){
25            myCondition.value = oldValue;
26        }
27        function bar(){
28            if (!myCondition.value){
29                myObject.myProperty = 0;
```

```
New watch expression...
oldValue                   undefined
⊞ [this.myCondition.value]  1→undefined
⊞ this                     Object {
                           myCondition={...},
                           _firebug={...},
                           more... }
⊞ Window                   Window {
                           HTMLBodyElement=,
                           Event=,   more... }
```

(c) The result of *lastChange* query for `myCondition.value`. To evaluate an expression (e.g., oldValue) at this point, developer can enter the expression in the watch box and after re-execution the result is available. The opened list on the top of the left panel shows the visited execution points. Clicking on each point in the list shows the corresponding code and data.

Figure 2: The stages of locating the defect using *lastChange* feature.

Q icons.

Notice that in our example, *lastChange* combines some aspects of breakpoint and of log-based debugging. Like breakpoint debugging, the developer re-executes a live runtime without changing the source and without a special execution environment beyond the debugger. The state of the program memory and the call stack are available at each lastChange point. Like log-based debugging, the program state and the call stack are recorded during program execution. We can't halt the program at *lastChange* because we don't know which point is the last one until we return to the original breakpoint. In section 5 we discuss cases where it is possible to pause at lines of *lastChange*.

3.1 Re-Execution, Reproduction Point and Data Collection

4. LIMITATIONS AND FUTURE WORK

5. REPRODUCIBLE NON-DETERMINISTIC EXECUTION

5.1 Combination of *lastChange* and Breakpoint Debugging

6. CONCLUSION

7. ACKNOWLEDGMENTS

8. REFERENCES

[1] J.J. Barton, and J. Odvarko. Dynamic and graphical web page breakpoints. In *Conference on World Wide Web(WWW)*, April, 2010.

[2] S. Bhansali, W. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, J. Chau. Framework for instruction-level tracing and analysis of program executions. In *International Conference on Virtual Execution Environments(VEE)*, June, 2006.

[3] M.D. Bond, N. Nethercote, S.W. Kent, S.Z. Guyer, and K.S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. In *22nd annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications(OOPSLA)*, October, 2007.

[4] B. Boothe. Efficient algorithms for bidirectional debugging. In *Conference on Programming Language Design and Implementation(PLDI)*, June, 2000.

[5] C. Hofer, M. Denker, and S. Ducasse. Implementing a backward-in-time debugger. In Proceedings of*NODe'06*, volume P-88, pages 17-32. Lecture Notes in Informatics, 2006.

[6] S. Horwitz, B. Liblit, and M. Polishchuk. Better Debugging via Output Tracing and Callstack-Sensitive Slicing. *IEEE Transactions on Software Engineering*, 36(1):7-19, 2010.

[7] Java Platform Debugger Architecture. http://java.sun.com/javase/technologies/ core/toolsapis/jpda.

[8] A.J. Ko, and B.A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *30th international conference on Software engineering(ICSE)*, May, 2008.

[9] B. Korel and J. Laski. Dynamic Program Slicing. *Information Processing Letters*, 29(3):155-163, 1988.

[10] T.D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits In *28th international conference on Software engineering(ICSE)*, May, 2006.

[11] B. Lewis, and M. Ducasse. Using events to debug Java programs backwards in time. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications(OOPSLA)*, 2003.

[12] A. Lienhard, T. Gîrba, and O. Nierstrasz. Practical Object-Oriented Back-in-Time Debugging. In *22nd European conference on Object-Oriented Programming(ECOOP)*, July, 2008.

[13] K. Maruyama, and T. Kazutaka. Debugging with Reverse Watchpoint. In *Proceedings of the Third International Conference on Quality Software*, 2003.

[14] Mozila JavaScript Debugging Interface. http://www.mozilla.org/js/jsd.

[15] G. Pothier, É. Tanter, and J. Piquer. Scalable omniscient debugging. In *22nd annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications(OOPSLA)*, October, 2007.

[16] M. Sridharan, S.J. Fink , and R. Bodik. Thin slicing. In *Conference on Programming Language Design and Implementation(PLDI)*, June, 2007.

[17] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352-357, July 1984.

[18] A. Zeller. Why programs fail: A guide to systematic debugging. Morgan Kaufmann (2005)