Debugging by *lastChange*

Salman Mirghasemi Claude Petitpierre
Ecole Polytechnique Fédérale de Lausanne
{salman.mirghasemi,claude.petitpierre}@epfl.ch

John J. Barton
IBM Research - Almaden
johnjbarton@johnjbarton.com

Abstract

Backward search from bug symptoms to defects-which cause the bug-is a natural way to debugging. However, due to the lack of support for easily bakward movement by traditional debugging approaches (e.g., breakpoint-based or log-based debugging), applying this strategy demands a lot of effort from developers. A fundumental step in this process is tracking origins to wrong values and any aid in this regard can greatly reduce the complexity of debugging for developers.

In this paper we present a new functionality in debuggers, *lastChange*, which locates the last place a value is changed and lets developer to collect data and ask for more *lastChange* queries at the point. The only prerequisite for *lastChange* functionality is bug reproduciblity (i.e., there is a test case which reproduces the bug). We explain the algorithms behind this functionality. As a proof of concept we developed a javascript prototype which adds this feature to a well-known javascript debugger, Firebug.

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 LastChange, Locating Defects, Breakpoint, Watchpoint, Logging

1. Introduction

Debugging is an inevitable part of programming, still hard and time-consuming. To fix a bug, developers have to reproduce and monitor the buggy execution several times to understand the program's unexpected behavior. Trial and error, guess-work and analyzing huge collected data are the inseparable parts of this process. According to [5], developers spend about fifty percent of their time debugging. Therefore, every improvement to debugging techniques and tools can considerably save developers' time and improve programs' quiality.

Locating defects cause the bug is the main part of debugging. A common strategy for locating defects is starting from bug symptoms and backward movement by following the origings to wrong values. While this strategy seems straight-forward and effective, applying this strategy using traditional methods is complicated. There are two traditional approaches to debugging, breakpoint-based and log-based debugging. None of these approaches assist developer in finding origings to a wrong value. It is due to developer to search source files, find the list of possible origings to a wrong value and set breakpoints or insert log statements in a way that covers all possible origins. The next dilemma is data collection. In breakpoint debugging, developer has to memorize values or collect data manually at every breakpoint hit and in cases that there are many breakpoint hits, this becomes a tedious task. In log-based debugging, developer has to decide about data should be collected when inserts the log statement. In this approach developer ends up in dealing with long log files and analyzing huge collected data.

Our proposal is a new functionality in debuggers which locates the origin of a wrong value, call it *lastChange*. The only prerequisite for *lastChange* functionality is bug reproduciblity (i.e., there is a test case which reproduces the bug). Assume that the program execution is paused on a breakpoint and the developer is suspicious about the value of an object property or variable. The developer selects *lastChange* on the value and it is due to debugger to locate and shows the last change of object property or variable. Debugger reproduces the buggy execution and collects limited data and once the execution reaches the same place (i.e., the same breakpoint hit), call it *reproduction point*, it pauses the execution, analyzes the collected data and shows the location of the last change to the developer. Developer can also examine the program state at the located point of execution, and asks for a new *lastChange* at this point.

Our contritution in this paper is the algorightm *lastchange*, which locates the last place a value is changed, gathers other values from that execution point, and allows *lastChange* operations from that point. The algorithm builds on existing breakpoint debugger technology. We demonstrate the feasibiltily of the algorithm with an implementation extending the Firebug Javascript debugger.

2. 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) which shows the `myObject.myProperty` value. Whenever user clicks on the button `onClick` function is called. This function increases the value of `myObject.myProperty` by one (line 15) and calls `updateButton` function which updates button's text to the new value. Once the page is loaded for the first time the button shows `1` as the initial value of `myObject.myProperty`. The bug happens when user clicks on the button and instead of `2`, `0` appears on the button.

To start debugging, the developer sets a abreakpoint on line 22 which is the place button's text is updated. Once the button is clicked execution is paused at line 22. (Figure 3(a)) shows the debugger while the execution is paused. The developer can examine the program state at the right panel and as it shows the `myObject.myProperty` value at this point is zero. This value is wrong and the developer seeks the last place this value is changed. By right clicking on `myObject.myProperty` in the right panel, the

```
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. The buggy javascript code.

developer can run *lastChange* command (Figure 3(b)). Debugger considers the paused point at the breakpoint as the *reproduction point*, and re-executes the buggy execution, pauses at the reproduction point and shows line 29 in a new panel (Figure 3(c)). The program state debugger shows on the right panel belongs to the past. Therefore the right panel just shows the partially collected data. If the developer needs some values which are not available in the right panel, debuggers re-executes another time and collects and shows the requested data.

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 *lastChange* command on `myCondition.value` at this point. Debugger re-executes the execution and pauses at the reproduction point and shows line 25-the place `undefined` value is assigned to `myCondition.value` (Figure 3(c)). Now it is clear that the bug occures because `oldValue` is `undefined` once execution reaches line 25.

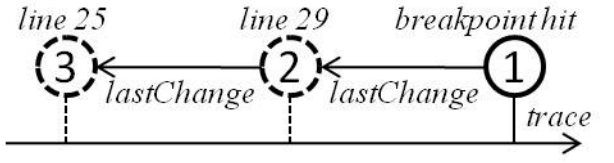


Figure 2. The examinated points.

As demonstrated in Figure 2, the developer examinated three points of execution. These 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 3(c) there is an opened list which shows all three examinated points. The first one is the breakpoint on line 22, the second one is the point which is the last change of `myObject.myProperty` before the 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 tagged with red-cycle 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 the 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 is the last one until we return to the original breakpoint.

3. *lastChange* implementation

lastChange can be called on two different types of values: An object property or a variable value. We explain each case separately.

3.1 *lastChange* on object propery

At the beginning, we assume that every object has a unique id. Moreover, we assume that there is a special watchpoint, *property watchpoint*, that can be set on a property name `foo` and whenever property `foo` in any object changes this watchpoint hits. We know that none of these assumptions is true in javascript. We discuss this issue later in traking object creation.

Once developer asks for the last change of `bar.foo` at a reproduction point, debugger sets a property watchpoint on `foo` and re-executes the program. Whenever the property watchpoint hits, debugger keeps the object id and stack frame locations. Whenever the execution reaches the same reproduction point it looks at the list of all changes and finds the last `foo` change with the same object id as `bar` id at the reproduction point(Figure 4).

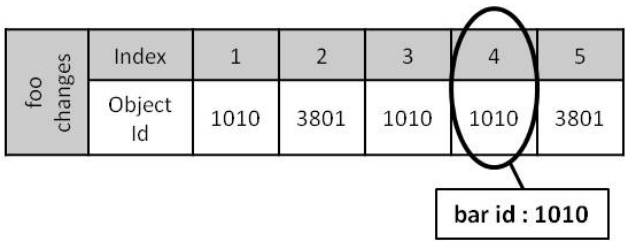
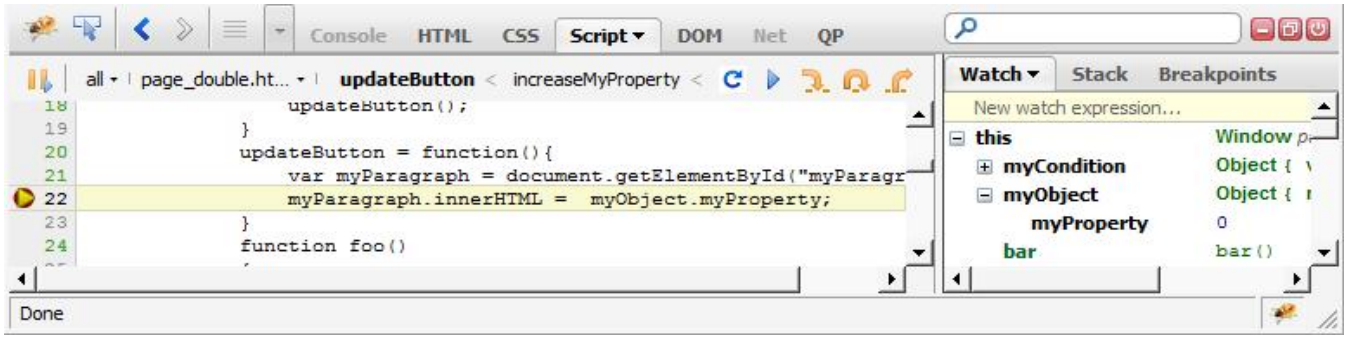


Figure 4. The list of `foo` changes and `bar` id which identifies the last change of `bar.foo` in the history of events.

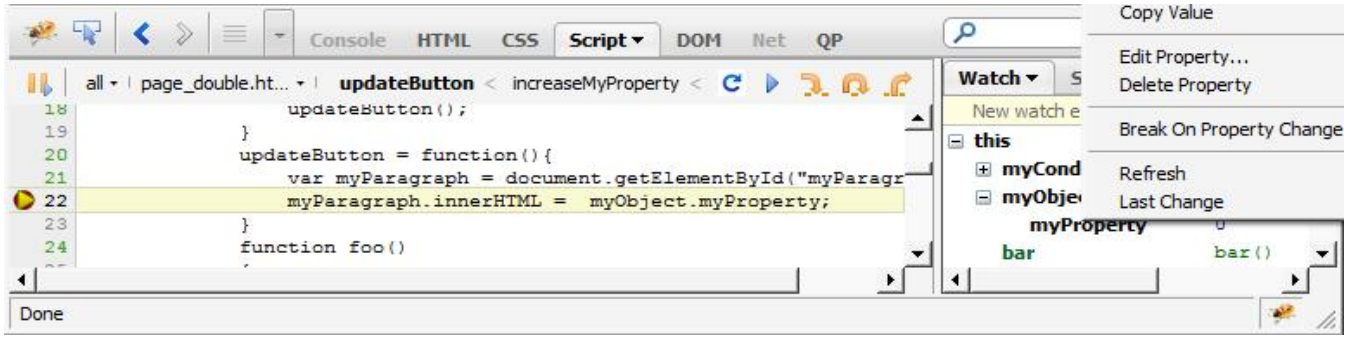
3.2 *lastChange* on variable

Once developer asks for the last change of variable `foo` at a reproduction point, debugger first finds the scope the variable comes from. Consider that in javascript, available variables are not resricted to local varialbes, but all variables in outer scopes are also accessible. The outer scopes are regualr javascript objects or closure scopes. The first group are usually created by calling `with()` function. The second group are created once a function is called. Therefore, there are theree cases for the escope of a variable, the current scope, an outer scope regular object or an outer scope clousure scope.

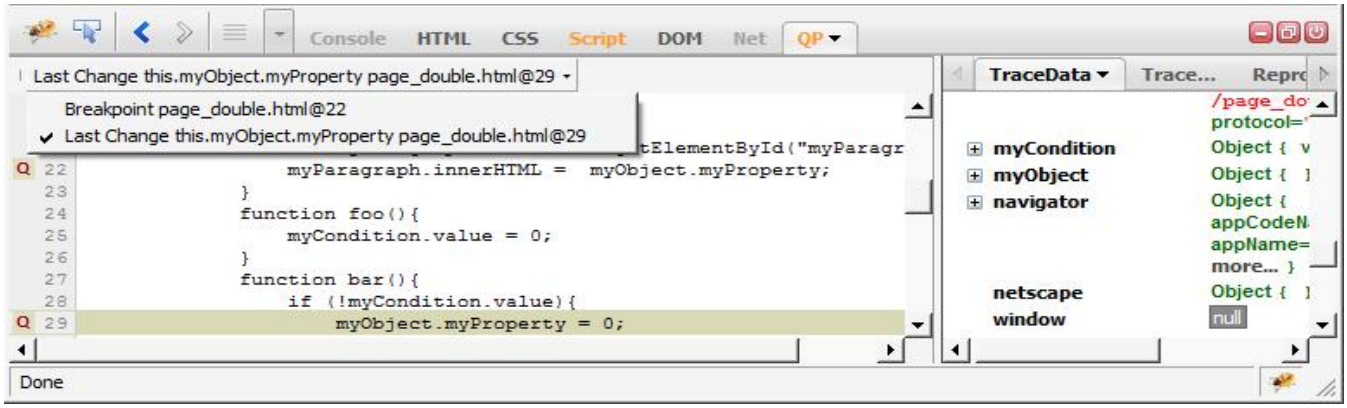
If the variable belongs to an outer scope regular object, debugger treats the case like *lastChange* on an object property. In the



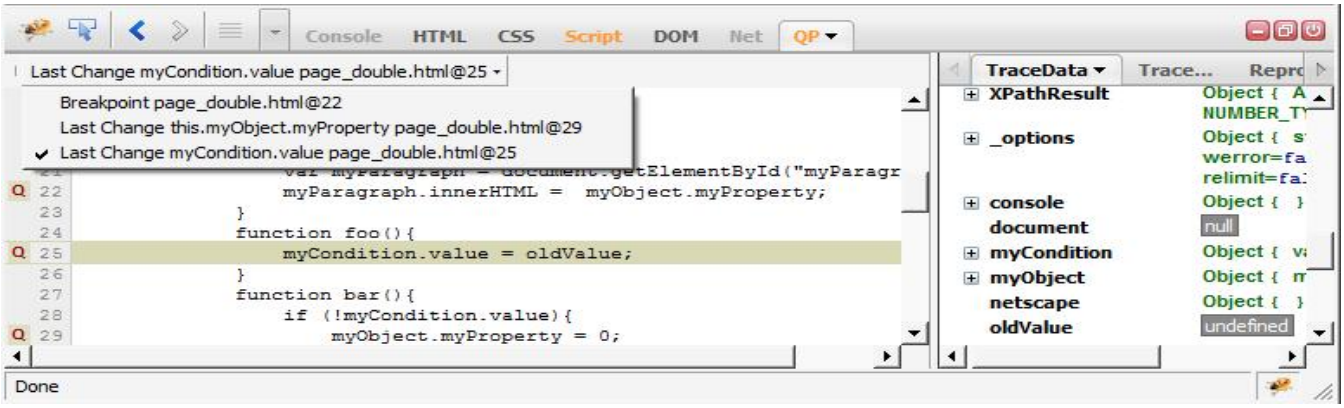
(a) The execution is paused at line 22 by an ordinary breakpoint.



(b) *lastChange* command.



(c) The result of *lastChange* call on `myObject.myProperty`.



(d) The result of *lastChange* call on `myCondition.value`.

Figure 3. The stages of locating the defect using *lastChange* feature.

other two cases, debugger sets breakpoints on the line the variable is defined and all lines the variable is assigned a new variable (including inner-functions). For example in Figure 5, if the execution is paused at line 24 and the last change of `var2` is requested by the developer, two breakpoints on lines 22 and 23 will be set, but if the execution is paused at line 19, four breakpoints on lines 11, 12, 16, and 18.

```
10  function parent(var1){
11    var var2;
12    var2 = 1;
13    function myfun(var3){
14      var var4;
15      var4 = {...}
16      var2 = var4;
17      function firstChild(var5){
18        var2 = 5;
19        var4 = 5;
20      }
21      function secondChild(var6){
22        var var2;
23        var2 = 5
24        var4 = 5;
25      }
26    }
23  }
```

Figure 5. Sample javascript code demonstrating closure variables.

Everytime one of the breakpoints hit a new event is inserted to the list of events. To separate irrelvant event from each other, we use a special variable `_scopeId`. If the scope has this variable, the value of this variable is used for the scope id otherwise a new id will be generated and `_scopeId` variable is created in the scope. This is specially needed in case of recursive calls. Figure Figure 6 shows changes to a the same variable `x` but in different scopes. alling *lastChange(x)* at point B should return point A.

```
Recursive function:
 f(){
   var x;
   x++;
   f();
   x;
   ...
 }

The trace of recursive calls:

  f was called.
    x changes; // A
      f was called;
        x changes;
        f was called
          x changes;
    //B
```

Figure 6. Recursive calls.

3.3 *lastChange* on *lastChange*

Once the developer asks for the first *lastChange* all program state is available to the debugger. But the next *lastChange* calls are asked over collected data. For example, debugger needs object id for the second *lastChange* and this data should be collected. To do so, debugger does a data dependency analysis between points which gives a list of items should be collected at every point.

4. Javascript implementation

The javascript prototype is implemented as an extension to Firebug[] javasrcipt debugger.

4.1 Execution reproduction

Altough execution reproduction is basicly should be provided by the developer, we tried to devise some automatic way which reproduces the execution. In the prototype, develooper can choose among three reproduction options. The first one is using a test case which provided by developer. The second one is by record and replay and the third one is local-reproduction.

Two parts should be carefully considered in execution reproduction. First, the initial state should be the same as previous execution. Second, the similar actions and events should be applied to the program during the execution. In the record and replay approach, debugger stores the initial page url and user actions and during replay phase, opens the same url and emulates user actions. In the local reproduction approach, debugger calls the funtion on the bottom of stack with the same parameters.

4.2 Tracking object creation

Now we discuss two mentioned assumptions. In the standard javascript every object has a `watch()` function which receives two parameters, a property name and another function as a hook. Whenever the property with the given name changes the hook function is called. The issue with using this function is that it can only be applied to available objects. However, at the beginning of re-execution there is no object available. To solve this issue we use a feature in Firefox javascript engine, setting a flag gives us the place the object created (e.g., myFile.js, line 24). Though this is a limited data it can help us to get a reference to the object once it is created. Therefore, at the reproduction point, debugger gets the creation location of `bar` and once the execution reaches to that location it gets a reference to the created object and sets a unique id for it.

4.3 Data collection

4.4 User interface

5. Converting *lastChange* result to a conditional breakpoint

We believe that in many cases it is possible to convert the *lastChange* result to a conditional breakpoint and therefore debugger is able to pause at the point instead of showing collected data to the developer. Here, we present an example which shows a case this transformation is not possible.

Figure 7 shows a function with processes an array. The array contains numbers except one itme which is `undefined` and it causes a bug in line 30. There is a call to `randomPermutation` function in line 11 which randomly permutates the array item. So at every execution the `undefined` item will be in a new place. Calling *lastChange* on `x` at the place bug happens, gives a point which shows line 14. Although this point exists at every re-execution but it can not be identified by a conditional breakpoint.

6. Reproducible Non-deterministic Execution

Thus far we have not discussed problems caused by mul-tiple threads or other sources of non-deterministic execu-tions. We want to explain why we believe *lastChange* is robust in the practically important case where a bug is reproducible even though the ex-

```
10 function randomProcess(array){
11   randomPermutation(array);
12   var x, y;
13   for (var i=0 ; i<array.length ; i++){
14      x = array[i]
...
30      y = x+1;
31   }
32 }
```

Figure 7. A counter-example for transforming *lastChange* to a conditional breakpoint.

ecution may not be deterministic. Because *lastChange* require re-execution, we rely on reproducible but not necessarily deterministic execution. A bug is reproducible for a developer when the developer can start from a determined initial state, operate on the program with a list of actions, and reproduce the symptoms of the bug. The details of the execution can change each time we re-execute the buggy program, but the buggy result is the same. The entire query chain reapplies during each execution so the data we show the developer will be internally consistent. The reproducibility of the bug means that the defect is very unlikely to depend on the order of events during the execution.

In this important case of reproducible bugs, *lastChange* are more effective than breakpoints. In the case of logically deterministic program execution, we can use the result from a *lastChange* operation to set a conditional breakpoint then re-execute the program to position the execution trace backwards from our first breakpoint.

In the case of a non-deterministic program, a *lastChange* is not equivalent to any series of conventional watchpoints or breakpoints. Each time we re-execute a non-deterministic program, the details of execution in-struction order may change. For example, if we record the source code lines every time a conventional watchpoint hits, the record may differ each time we re-execute. Sup-pose we consult one such record and set a breakpoint on the last entry, the apparent lastChange source line. When we re-execute, the breakpoint will hit, but the information we gain may be incorrect: this may not be the lastChange for this particular re-execution. The *lastChange* method records the values we need and the sequence of source lines from all of the watchpoints, then analyzes the record to select the correct lastChange point. The data shown to the developer will be internally consistent, but of course it may change from a previous re-execution, surprising the developer. This is just a signal that the execution is not deterministic. In future we hope to compare queries from successive executions as a tool for learning about non-deterministic executions.

7. Evaluation

8. Related Work

lastChange functionality supports obtaining information about the execution state logically earlier in the control flow. This support resembles a mixture of replay-based and logging-based debugging. Replay-based approaches capture limited data during execution and replay the bug-gy execution to reach past points. In contrast, logging-based approaches collect enough data during execution to relieve developer from re-execution. Replay-based approaches impose much less runtime overhead (about two orders of magnitudes) comparing to logging-based appproches. However, developer has to re-execute the buggy execution several times. *lastChange* functionality collects data on re-execution but this data is limited to the current queries of developer.

Among replay-based debuggers we compare to bdb [2] and reverse watchpoint [8]. 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 (i.e., the same sequence of instructions in re-execution) 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, is proposed by Maruyama et al., analyses the execution and moves the debugger to the last write access of a selected variable by re-executing the program from the [8]. Similar to bdb it relies on deterministic replay and uses a counter to correctly locate a point in the next execution. The main disadvantage of these appraoches is requiring the exactly the same executions. Even one instruction difference between in two executions leads to wrong results. This requirement is that much restricting that after ten years from bdb paper publication, there is no newer implementation of this idea. On the other hand, *lastChange* doesn't require any special feature in the re-rexecution and it is fit to everyday's developers' debugging practice.

Among logging-based approaches are "omniscient" debuggers ODB[6] and Unstuck[]. Omniscient debuggers have been proposed as a solution for the problems of breakpoint-debugging. 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 this approach there is no execution to resume: moving backwards in the log can be similar to moving forwards. Omniscient debuggers suffer from a different set of 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 omniscient debuggers. Third, querying collected data (e.g., to restore the program state at a certain point) may not be efficient enough for debugging of realistic programs.

A more scalable approach has been proposed by Pothier et al. [9]. Their back-in-time debugger, TOD, addresses the space problem by storing execution events in a distributed database. Comparing to Omniscient debuggers our approach is lightweight and more flexible. Developer can start debugging just after reproducing bug without a capturing step. Changing inputs or environment settings and re-executing to investigate the bug works as in conventional breakpoint debuggers.

Two new directions in logging debuggers explore more detailed use of the log and more effective logging approaches. WhyLine[4] provides visual interface to collected runtime information and let developer to move on execution log using queries expressed in terms of the programming objects. WhyLine stores the program user in-terface in addition to program trace and provides answers to why and why not questions to the user. Jive[3] depicts the history of execution by a sequence diagram and lets user to query on events database. Both tools suffer from similar issues with omniscient debuggers;

A recent work by Lienhard et al.[7] suggests virtual machine level support for keeping the object flow. It rep-laces every object reference with an alias object which keeps the history of changes to the object reference. In this way, when an object is collected by garbage collector, its track of changes (if it is not referenced by other alias-es) will be also collected. Though this approach incurs less runtime overhead (7 times to 115 times) in comparison to omniscient debuggers, it adds memory overhead. Querypoint debugging uses re-executions to gather infor-mation requested by the

developer: the memory overhead depends on the query not the entire program. Moreover, the Lienhard et al. debugger significantly changes the virtual machine, while our approach is a generalization to conditional breakpoints and available debugger infrastructure can be adapted to support it.

lastChange functionality does rely on a conventional breakpoint to begin queries, a requirement not shared by full logging solutions. Here we leverage past experience of developers, but there are also new tools [] to help with this problem in the case of graphical and event based systems.

9. Conclusions and Future Work

References

[1] 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.

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

[3] J.K. Czyz, and B. Jayaraman. Declarative and visual debugging in Eclipse. In *OOPSLA workshop on eclipse technology eXchange*, October, 2007.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.