Debugging by *lastChange*

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

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

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

ABSTRACT

We introduce a new, practical feature for debuggers 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. As a proof of this concept, we developed *Querypoint*, a prototype which enhances the popular Firebug JavaScript debugger with the *lastChange* feature and studied users applying the prototype to some test cases. The approach used in implementing *lastChange* combines the flexibility of breakpoint debugging with the expressive power of log-based query debugging. Contrary to other replay-based approaches, which require exactly the same re-executions (deterministic executions), our new approach only requires *bug reproducibility*, meaning a test case is available which reproduces the bug and a way to halt execution reliably after the reproduction.

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 efficiently 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 debug-

ESEC/FSE '11 SZEGED, HUNGARY

ging 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.

The rest of the paper is organized as follows. First, we demonstrate the *lastChange* usage on a simple example with the comparison to breakpoint debugging. Section 3 presents *lastChange* algorithm. In section 4, we explain the details of the JavaScript prototype implementation. We discuss the effect of non-determinism on the *lastChange* results in section 5. The user study results are presented in section 6.

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) 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.

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 3(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`.

```
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.

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 3(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 3(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`.

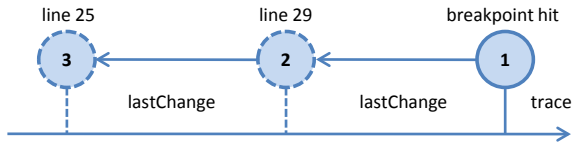


Figure 2: The examined points before locating the defect. The arrow represents the logical forward progress of the program. Three actual executions are superimposed on this arrow. All three stop at the reproduction point indicated by circle 1. After the first execution, the developer asks for lastChange as described in section 2, yielding information indicated by circle 2. After the second execution, another lastChange query causes a third execution, yielding information indicated by circle 3.

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 3(c)).

As demonstrated in Figure 2, 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 3(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 **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. LASTCHANGE ALGORITHM

The *lastChange* algorithm is based on program re-execution of a program halted on a breakpoint. The algorithm starts when developer examines the program state at a breakpoint hit and asks for the *lastChange* of a value. The breakpoint hit becomes the *reproduction point*. Debugger sets hooks (a callback function dependent upon the underlying runtime) on all instructions that might be the result of *lastChange* query. Then the debugger re-executes the program and every time a hook hits it checks for a *change event*. In the case of a change, it stores part of the program state values. Once the execution reaches the reproduction point, it analyzes the collected data and shows the result. The program state at the execution point of the last change event is the *lastChange*.

As we described in the preceding section, a *lastChange* query can be performed on the result of another *lastChange* query. If we name the reproduction point R, we can write the first *lastChange* in the introductory example in this form: *lastChange(R, myObject.myProperty)*. It means that this query is defined at R. If we name the result of this query L, we can write the second *lastChange* in this form: *lastChange(L, myCondition.value)*. In this way, a sequence of *lastChange* queries with any length can be defined.

lastChange can be called on object property, on a variable value, or on the results of a *lastChange*. Moroever, common data structures such as arrays and hashmaps are also supported as special cases of *lastChange* on object property. We explain each case in the following subsections.

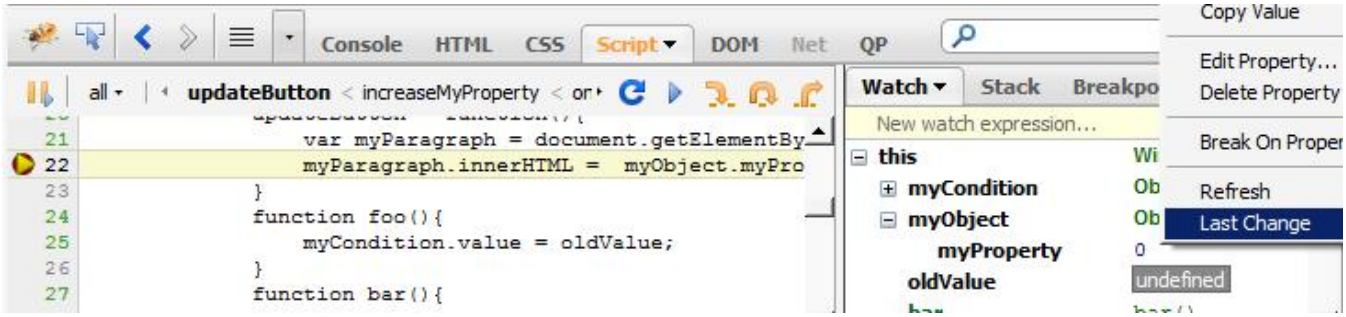
3.1 *lastChange* on Object Property

To simplify the algorithm explanation and defer technical details, we define two basic operations and later we explain the details of these two operations. The first operation is `objectId()`: given a JavaScript object it returns an integer as its identifier. This identifier is unique to the object during one execution. By using an object id instead of an object reference we allow the garbage collector to reclaim the space for dead objects just as it would in the absence of the debugger. The second operation is `setPropertyChangeHook()`: given a function and a string, the function is called whenever a property changes and its name matches the string. For example, if the string is `foo`, changes to `bar.foo` or `baz.foo` would call the function. The callback function receives a reference to the owner of `foo`.

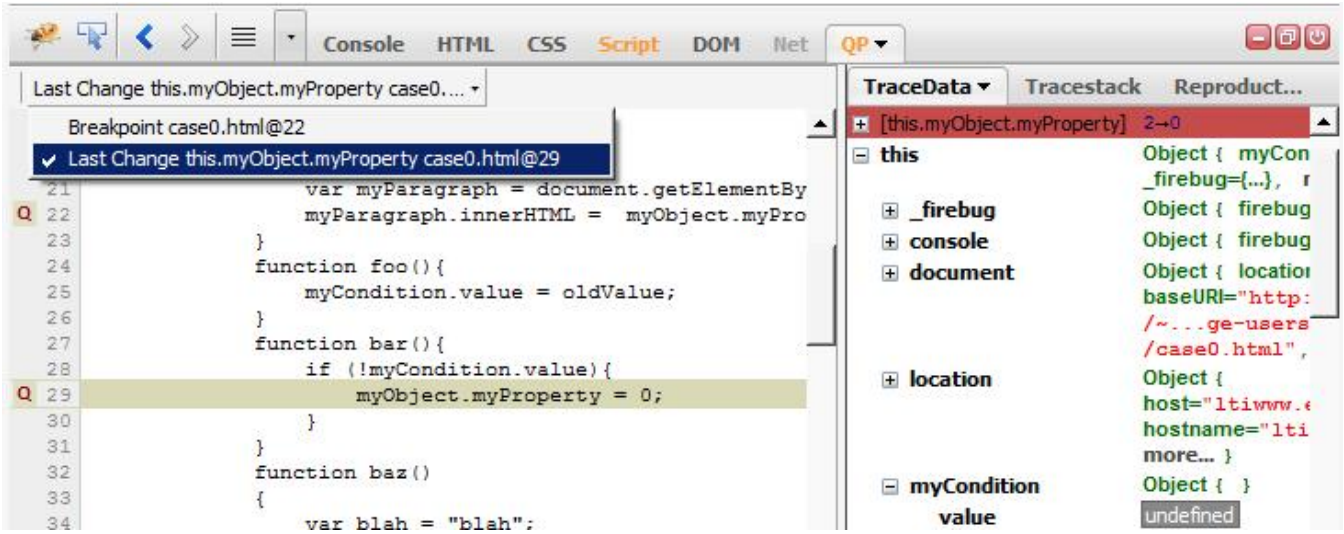
To see how these functions work, suppose the developer asks for the last change of `bar.foo` at the reproduction point in a program. The debugger calls `setPropertyChangeHook()` with `foo` as the property name and re-executes the program. Whenever `foo` changes and the callback function is to be called, debugger first calls `objectId()` on the `foo` owner object. Then it stores this owner id, the stack frame locations, and other state values in scope at the call point. Then the callback returns to continue the execution. Thus the query is not a breakpoint in the sense of pausing for user interaction, but breakpoint technology can be used to implement the query. Whenever the execution reaches the reproduction point the debugger looks at the history of `foo` changes and finds the last `foo` change with the same object id as `bar` id at the reproduction point. Figure 4 shows the list of property `foo` change events in a hypothetical execution. `bar` id at the reproduction point is 1010, so the last change of `bar.foo` is the fourth column.

3.2 *lastChange* on Variable

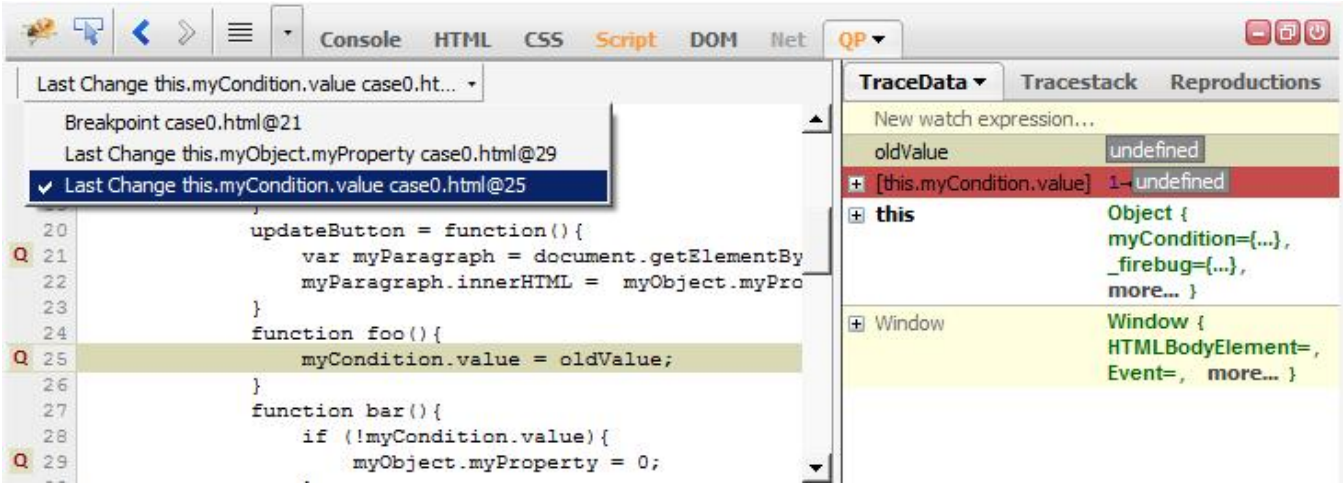
In JavaScript, every frame has a scope chain and every available variable in the frame comes from one of the scopes in the frame's scope chain. Once the developer asks for the last change of a variable with name `foo` at the reproduction



(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`.



(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.



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

bar id at the reproduction point : **1010**

foo changes	Index	1	2	3	4	5
	Owner id	1010	3801	1010	**1010**	3801

Call Stack
panel.js : 505
dispatcher.js : 44
...
index.html : 103

Figure 4: A hypothetical list of change events for a property `foo`. Each change event adds a column with the id of the object changed, the call stack, and some program state such as local variable values. At the reproduction point we determine which id corresponds to object `bar` and read out column 4, the last change of `bar.foo`. Column 3 is also a change of the object we want to study, id 1010, but it is not the last change; Column 5 is also a change of a property `foo` but it is not for the object we are interested in.

point, the debugger first determines the variable's scope as follows: it iterates over the scopes in the scope chain and the first scope which has a variable with the same name is the variable's scope. There are five different scope types: global, local, closure, `with` and `catch`. We explain these cases in two groups.

3.2.1 *global and* `with` *scopes*

Global scope is the most outer scope in the scope chain and it is also referred to as the global object (the `window` object in Web pages). This scope is a regular JavaScript object and therefore every global variable is a property of global object. Similarly, `with` scopes are also regular JavaScript objects. A `with` scope is created by a `with()` block with an object as the parameter. Every property of this parameter object is available inside the block as a variable. *lastChange* treats the case where variable's scope is global or `with`, like it does on an object property.

3.2.2 *local, closure and* `catch` *scopes*

Local scope refers to the most nested scope in the scope chain which contains the local variables. Closure scope refers to the scope which is created for a nested function and contains variables defined in the outer block. Catch scope is the scope created in the catch block of try-catch statements and contains the exception variable. These scopes are not necessarily regular JavaScript objects. Therefore, to track changes to a variable in these scopes we employ a different approach.

Having the scope chain and the source code, we can map every scope to a code block, enclosed the executing code. In JavaScript, a code block can be identified by the file url and the block's first instruction program counter. Given this information, the debugger is able to recognize the code block in loaded scripts or once it is loaded. Similar to *lastChange* on object property, we define two basic operations: `scopeId()` – which returns an integer given a scope – and `setVariableChangeHook()` – which calls its first parameter, a callback function, when a variable in its second parameter, a code block, matches the name given in the third parameter.

Code	Sample Trace	
function f ( ) {	f ( )	
var x, y;	**A** x ++	scope 1
x = 0;	f ( )	
. . .	x ++	scope 2
x ++;	f ( )	
. . .		
if (! stop) f ( );		
. . .	**B** x ++	scope 3
y = x;	**C** y = x	scope 1
}		

Figure 5: A recursive call trace illustrating the scope id. The lines in the bottom half of the diagram simulate a trace of the change events for the variable `x` as the function `f()` calls itself. Each call creates a scope; eventually when the variable `stop` is changed by an external process we return from the recursion. The lines marked A, B, and C are discussed in the text.

Figure 5 illustrates how the `scopeId()` operation separates instances of a variable in different scopes having the same name. If we ask for the last change on `x` at point labeled `C` in scope with id 1, we want the change at line `A` in scope 1, not the change at the line marked `B`, where a variable named `x` in scope 3 is changed.

If the developer asks for the last change of variable `foo` at the reproduction point in a program, debugger calls `setVariableChangeHook()` with the variable's defining block and name as parameters and re-executes the program. Whenever `foo` changes and the callback function is to be called, debugger first calls `scopeId()` on the variable's scope. Then it stores this scope id, the stack frame locations, and other state values in scope at the call point. Whenever the execution reaches the reproduction point the debugger looks at the history of `foo` changes and finds the last `foo` change with the same scope id as the variable's scope id at the reproduction point (Figure 6).

foo scope id at the reproduction point : **55**

foo changes	Index	1	2	3	4	5
	Scope id	50	**55**	36	50	100

Figure 6: A hypothetical list of variable `foo` change events. Each column of the list indicates a change event; for each change event the scope id returns by `scopeId()` is recorded along with call stack and program state information. The last column having the scope id `foo` at the reproduction point indicates the last change.

3.3 *lastChange* on *lastChange*

The *lastChange* algorithm records changes by id (either object or scope id), then reads out the last change when we arrive at the reproduction point and discover the id of the requested value. When we perform *lastChange* based on a previous *lastChange*, the query algorithm must retain additional information. Consider the following example:

```
point A : the reproduction point
point B : lastChange(A, bar.x)
point C : lastChange(B, baz.y)
```

where point A is a breakpoint, point B is the last change of the object property `bar.x` at point A, and point C is the last change of the object property `baz.y` at point B. The object referenced by `baz` changes upon re-execution. Therefore when the developer asks for the last change of `baz.y`, we need to track objects named `baz` at changes of `bar.x` and changes to objects named `y`. Then, at the reproduction point, we need to work out which `baz` the developer wanted, then select the last change of that `baz.y`. Figure 7 illustrates the extra row of data (tracking of `baz` objects at `x` change events) and how the id values allow the last change of `baz.y` to be worked out.

In the general case we perform dependency analysis as outlined in Figure 8 to create the list of additional data (object id or scope id) to be collected at a change event. The process can be repeated to cascade *lastChange* arbitrarily deep.

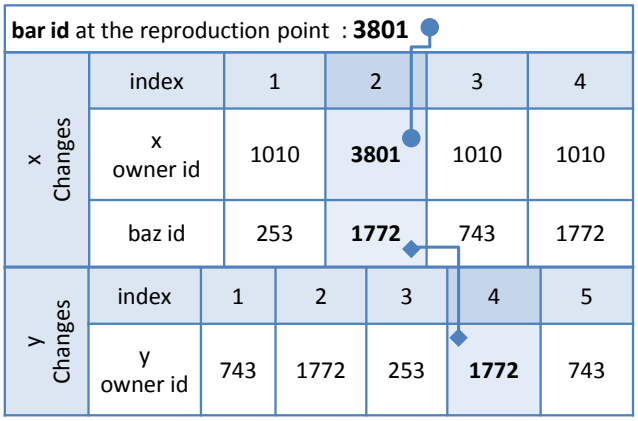


Figure 7: The list of change events stored for locating point B, the *lastChange* of `bar.x` at the reproduction point, and point C, the *lastChange* of `baz.y` at point B.

4. JAVASCRIPT IMPLEMENTATION

To verify the *lastChange* algorithm we implemented[1] it in an extension to the Firebug JavaScript debugger[2]. Firebug itself is an extension of the Firefox browser. The Firefox JavaScript engine provides a JavaScript debugging interface [14] and *Querypoint* is developed over this interface. Our prototype implements the four primitive operations in JavaScript using techniques which are cumbersome and comparatively slow to execute. However the JavaScript prototype is easy to explore, change, and share with others for feedback. A professionally useful debugger would implement these primitive operations within the JavaScript engine.

[1]http://code.google.com/p/querypoint-debugging
[2]http://getfirebug.com

---

for q in *lastChange* queries **do**
  for p is defined at the q result **do**
    if p is a *lastChange* on object property **then**
      the property **owner id** must be stored at q change events.
    else if p is a *lastChange* on variable **then**
      the variable **scope id** must be stored at q change events.
    end if
  end for
end for

Figure 8: *lastChange* queries dependency analysis.

4.1 `objectId()` Operation

`objectId(obj)` first checks the argument `obj` for a property `_objectId`. It returns the value if this property is already defined, otherwise it generates a new id and sets this property. The value of the id is simply an integer incremented for each new `_objectId` needed.

4.2 `setPropertyChangeHook()` Operation

The Firefox JavaScript engine supports watching property changes in an object. Every object has a function `watch(propName, callback)` which receives two parameters, a property name and a function. Whenever the property with the given name changes, the `callback` function is called. The hook set by this function remains enabled even if the property is deleted and defined again.

For our purposes, the `watch()` function only covers the case of global object properties. At the beginning of execution, no object excepting the global object and its predefined properties is available. For our *lastChange* prototype, we created a version of `setPropertyChangeHook()`. The basic strategy is to get a reference to the object just after its creation, then use `watch()` function to monitor property changes in the object. Setting a flag[3] into the Firefox JavaScript engine, we can get the file URL and the line number for each object creation (e.g., myFile.js, line 24). We set a breakpoint on this line and parse the source code to determine which object was created.

The only data we have is the object creation location including the file url and the line number and the goal is to get a reference to this object. Although in most regular cases we have only one statement and one object creation, there are cases where more objects are created in the line. There is no simple way to recognize the interesting object among these new objects. So instead of one object, we monitor all new objects created in the line.

An object might be created by one of these statements: object literal(`{...}`), *new* operator (`new constructor()`) or

[3]DISABLE_OBJECT_TRACE defined in jsdIDebuggerService.idl

function definition statement (`function()`). By parsing the source code we can recognize the statements that create an object. The next step is getting a reference to the new object.

The new object can be assigned to an object property or a variable by an assignment (`=`). In these cases we keep the assignee statement at the left side. The idea is that we create a list of assignee statements that the new objects are assigned to. We set a hook on the creation line. Once the hook hits, we evaluate the assignee statements. Then we do stepping(step-over) and after each step we evaluate the statements. Every statement which has a new value, we consider the new value (if it is an object) as a new object. For example in Figure 9(a), if the creation line number is 20, we have only one statement which creates a new object and it is assigned to `x.y`.

The new object can also be set as the property of a parent object (or array) inside an object (or array) literal. This case is also treated similar to previous case. The only difference is that the full path of property from the root parent in the local scope must be considered as the assignee statement. For example in Figure 9(b), if the creation line is 20, we keep `parent.child`. In this case stepping must be continued until the end of the object literal (line 22) for getting a reference to the created new object.

In cases where the new object is passed as an argument to a function (Figure 9(c)), we use step-in instead of step-over. To get a reference to the new object it is enough to evaluate the corresponding argument inside the function. The other cases where the program does not keep a reference to the created object (e.g., when a function object is just called after its creation), are not in our interest.

Although this approach is successful in many ordinary cases, we can imagine cases where a more comprehensive analysis needed for the correct behaviour. Consider the case where the new object is assigned to an expression like `a[++i]`. Obviously, evaluating this statement doesn't return a reference to the new object. Our prototype implementation does not handle these kinds of unusual cases yet.

Throughout this section we implicitly assumed that the object will be created at the same location in the next execution. If the assumption is not true, once the execution reaches the reproduction point, it reveals that the object has been created in a different location. This time, prototype re-executes the program considering both locations as possible object creation locations.

```
(a) Case 1:
 20   x.y = new MyClass();

(b) Case 2:
 19   parent = {
 20     child : {
 21       x : 5
 22   }}

(c) Case 3:
 20   myFunction({myProperty:5});
```

Figure 9: Examples of different cases in getting references to created JavaScript objects.

4.3 `scopeId()` Operation

The prototype sets a breakpoint at the beginning of all code blocks needing a scope id. The scope id is kept as a variable with name `_scopeId` in the scope. Whenever the hook is hit, meaning a new scope is created, `_scopeId` is set by calling JavaScript's dynamic compilation function `eval()`. For example, executing `eval("var _scopeId = 10")` creates a variable with name `_scopeId` and value `10` in the scope of the `eval()` call, which is our interesting scope. `scopeId()` operation returns the value of `_scopeId` in the scope.

4.4 `setVariableChangeHook()` Operation

Variables defined in local, closure, and catch scopes are only changed in their scope; that includes the defining scope and scopes nested whithin them. For example in Figure 10, variable `foo` in line 11 can only be changed in lines 10 to 23. Therefore, after locating the function in which the variable is defined, it is enough to parse the code inside the function block and set a hook on all lines where the variable is assigned a new value. We also set a hook at the first line of the function which is corresponding to the line where the variable is defined. In Figure 10, if the execution is paused at line 17 and the last change of `foo` is queried, two hooks on lines 16 and 17 will be set, but if the execution is paused at line 20 , four hooks on lines 11, 12, 14, 20 will be set. These two cases are different: `foo` in `childOne` is a local variable but in `childTwo` it is a closure variable.

```
10  function main(){
11    var foo;
12    foo = ...;
13    function parent(){
14      foo = ...;
15      function childOne(){
16        var foo;
17        foo = ...;
18      }
19      function childTwo(){
20        foo = ...;
21      }
22    }
23  }
```

Figure 10: Sample JavaScript code demonstrating local and closure variables.

4.5 Re-Execution, Reproduction Point and Data Collection

Querypoint needs a test case to reproduce the execution and conditions to correctly recognize the reproduction point. Although both elements can be directly provided by developer, *Querypoint* is also able to automatically create them from the first execution.

To replay execution, *Querypoint* keeps track of breakpoint hits and single steps. For example, if the developer queries *lastChange* at the third hit of breakpoint *b*, in re-execution, the third hit is recognized as the reproduction point.

The *Querypoint* prototype supports two mechanism for automatic re-execution: callstack-reproduction and record-replay. In callstack reproduction the function from the earliest frame of the call stack is called with the same parameters. The idea behind this mechanism is that many bugs in web pages can be reproduced by re-firing an event like clicking on a button. The record-replay execution uses two phases. In the record phase, it stores the initial page url and the events and parameters corresponding to user actions. In the replay

phase, it opens the same url and simulates events as if they were user actions. The callstack reproduction mechanism provides shorter re-execution cycles while the record-replay is more accurate about the initial state.

In addition to the data collected at every change event for identifying the *lastChange* result, *Querypoint* partially stores values in program state. There is a trade-off between the amount of data collected at every change event and the number of re-executions. If developer asks for some values which have not been stored, *Querypoint* re-executes and collects the requested data.

5. REPRODUCIBLE NON-DETERMINISTIC EXECUTION

We claim that the only prerequisite for *lastChange* is bug-reproducibility. A bug is *reproducible* for a developer when the developer can start from a given 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. All modern debuggers in wide use that we know about rely on reproducible but non-deterministic execution for simple practical reasons: developers must reproduce a bug to study it and modern execution environments are not deterministic. By relying on reproducibility but not requiring deterministic execution, *lastChange* works on the same range of cases.

5.1 *lastChange* Result Consistency

Each time we re-execute a non-deterministic program, the details of execution instruction 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. But *lastChange* does not compare values across executions. Rather it analyzes all of the change events in a single execution. Neither the data gathering nor the analysis require deterministic execution.

Since we don't need to compare values across executions to implement *lastChange*, we can get more information if we do compare: different points of last change on different executions will signal that the execution is not deterministic. Note the converse is not true, many non-deterministic programs will give identical results for *lastChange*. Consider the example in Figure 11. Assume that at the first execution, the developer sees that the `a` value is `true` and asks for the *lastChange* of `a` at the breakpoint. In the re-execution, `a` is `false`, and `b` is `true` at the reproduction point. It means that `lastChange` result shows line 10, which is a correct answer for this execution but not for the previous one. When user asks for the *lastChange* on variable `a`, debugger stores `a` value and compares it to the `a` value in the next executions. So if this value is different, debugger informs the developer that the *lastChange* query is made on a different value.

```
10 a = b = false;
20 if (random()) {a = true;} else {b = true;}
30 if (a || b) {bug(); /* breakpoint */}
```

Figure 11: *lastChange* on non-deterministic values.

5.2 Combination of *lastChange* and Breakpoint Debugging

Using *lastChange* a developer can work backwards on the flow of data, but sometimes bugs are more obvious when we watch control flow forwards. Consider for example,

```
44 vector.r = Math.floor(Math.random()*5)*6;
45 if (vector.r !== 0 && vector.r < 30);
46   return vector;
```

where *lastChange* shows us that `vector.r` is zero at line 44. In our user studies, developers wanted to single step forward from line 44. If they could do this, they may be surprised to see line 46 execute even if `vector.r` is zero, directing their attention to line 45 where they can discover the errant semicolon. However, a *lastChange* is just a query result, not a breakpoint. Under what conditions can we cause the debugger to stop at the *lastChange* point?

In the case that the execution is deterministic, the *lastChange* event index can be used as an index for a conditional breakpoint [4, 13]. Moreover, this works in a non-deterministic execution if non-determinism has no effect on the event index. The debugger can easily set this conditional breakpoint and reexecute the program. If the stream of change events differs in this re-execution the developer can be warned that the conditional breakpoint may not be the *lastChange* point. However, if the stream of events is the same, we do not know that the conditional breakpoint matches the last change, this fact can only be verified at the reproduction point. These theoretical concerns are not likely to cause significant practical problems: if the value at the conditional breakpoint is not suspicious, then the developer will know to simply return to *lastChange* or move on to another tactic to find the bug.

6. USER STUDY

We supplied four experienced Javascript developers with our prototype in an extended Firebug debugger[4]. Following a tutorial and a practice case, we observed as they applied both conventional breakpoint and *lastChange* on two small programs we provided (Our prototype did not have a user interface to support Sec. 5.2). The first program, Shapes, calculates the area and perimeter values for a list of shapes. The bug happens when one of the calculated numbers is zero. The second program, Moving Circle, randomly scales and moves a circle in the page. The bug happens once the circle becomes invisible after an exception occurs. This case represent a reproducible non-deterministic execution. The developers were asked to locate the defects that caused these bugs. All four developers successfully applied *lastChange* to the test programs and understood how it could help debugging.

To find the defect location with breakpoints, all four users took more steps and more time (Figure 12). Two users scrolled through the source, another searched the text, a third set a lot of breakpoints to understand the control flow. Based on our own experience we expect these strategies represent the kinds of approaches developers have available. These operations are time consuming and tedious. In contrast, all four users found the defect location with just two *lastChange* operations. We recognize that these programs

[4]http://ltiwww.epfl.ch/~mirghase/lastchange-userstudy

were designed to highlight lastChange and many kinds of debugging issues have been hidden by the design of our tests. Nevertheless our results show that, when a defect relates to incorrect values and a developer recognizes this, then the operational mechanics of lastChange lead to the defect much more quickly than breakpoints.

Our observation and discussions with users also brought out several important issues and improvements for our user interface. Perhaps the most important and challenging improvement would be better integration with breakpoint debugging. Our implementation put the results of *lastChange* in a similar but different view from breakpoint debugging. This focuses attention on value changes, but it makes studying control flow more difficult: we don't support single stepping from a *lastChange* result in our user interface. In our next iteration we plan to merge the query and breakpoint results and support pausing as described in Sec. 5.2.

Developers / Programs	DEV1	DEV2	DEV3	DEV4
Shapes	**3 (85 s)**	13 (182 s)	39 (318 s)	**3 (80 s)**
Moving Circle	9 (215 s)	**4 (40 s)**	**3 (195 s)**	12 (234 s)

Figure 12: The number of steps (and time in seconds) required before locating a defect, for each test subject and test program. Cells with a white background report values with conventional debugging; Cells with a colored background use *lastChange*.

7. DISCUSSION

We have presented the *lastChange* algorithm and described our prototype implementation. Our goal is practical improvements in debugging. To achieve our goal we need practical JavaScript engines to add new debug primitives so that developers in the field can use our new technique. This paper is one step to convince implementers to enable *lastChange*. Thus we summarize here our arguments that *lastChange* should be supported.

7.1 Developers need an operation like *lastChange*

Since ultimately programs are just transformation of state values, debugging is ultimately backtracking to find defects in program state change [17, 18]. When a developer halts a program on a breakpoint they compare the call stack and program state to their model of the program. If any value seems to be incorrect, they need to figure out what operation causes the incorrect value. Here *lastChange* takes over and addresses a key part of the debugging process.

7.2 Developers can learn to use *lastChange*

While our user study was small, our prototype demonstrates that *lastChange* can be easily activated by operations on the graphical representation of erroneous data in a debugger and the results can be interpreted by users. While *lastChange* is not a breakpoint, all of the assumptions developers already have for breakpoints hold for *lastChange*. In particular the re-execution is just the same operation developers use to debug with breakpoints.

7.3 Practical implementations are feasible

We have described our prototype all-JavaScript implementation in Sec. 4. It is adequate for exploring the ideas and may even be usable in production. However significant improvments can be made. A fully usable implementation would require access to the object id at the point of object creation and `setPropertyChangeHook()`. Many object-oriented runtimes provide object identifiers and provide access to object creation directly or by bytecode instrumentation [7].

7.4 In most cases *lastChange* will be much faster than current alternatives

Recall that we insert additional code through debugger callbacks, then re-execute the program. The additional code we insert is proportional (in our JavaScript algorithm) to 1) the number of places a property or variable with a given name is changed, 2) the number of places objects are created. The overhead for each execution at a change event depends upon the amount of data we store for each change event; the overhead for object creation could be small since we only need to determine if the object is one we need to watch.

For comparison consider today's practical alternative: developers setting breakpoints. For the vast majority of programs, a developer will take much more time to set one breakpoint than *lastChange* would add. Moreover, typically the developer may not guess the point of last change. They must then ponder another breakpoint and re-execute.

We could also compare to solutions based on logging or tracing. Manual logging has very high overhead: the developer must add code, debug that added code, then analyze the log (To be fair, the log can become a permanent debugging aid). Automatic logging as we discuss in section 8 causes about one or two orders of magnitude slow down as well as requiring a completely different set of development tools.

7.5 The worst cases are not more common or more painful than alternatives

Every time through the loop we incur the call back overhead; if the loop itself has relatively little code the overhead could be very large; if the loop computation is a significant fraction of the full program, the slowdown would be enormous. Other techniques also struggle with this case: breakpoints in highly repeating loops are not feasible and logging becomes unwieldy. Developers face this issue with any debugger today: occasionally a debugger causes too much overhead to be useful for debugging.

7.6 Generalizations

Our *lastChange* algorithm can be viewed as a particular interface to a general facility. The general facility replays execution, queries the runtime at points of interest during execution, and analyzes the result at the reproduction point. We have selected one kind of query and analysis that can be easily integrated in existing debuggers and explained to developers, providing automation of the problem of finding the point of last change. We believe other kinds of queries and analysis can be invented and integrated to automate other aspects of debugging.

8. 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 omniscient 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 is has little upfront cost and more flexibility. The 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.

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.

9. CONCLUSION

lastChange provides critical information for debugging programs: the location and state at the point where a questionable value was assigned. It builds upon existing technology and developer experience making it a practical solution for implementers. Our prototype demonstrates the feasibility of *lastChange* and its user interface hints at the potential this approach can have in organizing the debugging experience.

10. ACKNOWLEDGMENTS

The authors thank Leo Meyerovich for reading a draft of this paper and suggesting the example in section 5.1 and Jan Odvarko and Mike Collins for testing the prototype before our user study and providing helpful suggestions.

11. 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)