Querypoint Debugging

Salman Mirghasemi Claude Petitpierre John J. Barton
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne, Switzerland
{salman.mirghasemi,claude.petitpierre}@epfl.ch
IBM Research – Almaden
San Jose, CA
johnjbarton@johnjbarton.com

Abstract

To fix a bug, software developers have to examine the buggy execution to locate defects. They employ different approaches (e.g., setting breakpoints, inserting printing statements in the code) to navigate over buggy execution and inspect program state at suspicious points. In this paper we describe *Querypoints*, a new kind of compound conditional breakpoint relating two execution points. Developers specify Querypoints relative to a successfully paused conventional breakpoint or conventional watchpoint or to another Querypoint. The relative conditions contain runtime data values, like the last time a value was changed, or program statements, like the last conditional branch. The Querypoint combines the program state information from two execution points in the same execution; Querypoints can be chained to work backwards from effects to causes in a program.

We present basic Querypoint concepts, two sample Querypoints, *lastChange* and *lastCondition*, and a description of our implementation of these Querypoints. To demonstrate that Querypoints are feasible we describe a prototype that implements an alternative approach to finding a bug in a graphical program analyzed previously with Whyline, one of the new logging-based debuggers.

Categories and Subject Descriptors D.2.5 [**Testing and Debugging**]: Debugging aids; D.2.6 [**Programming Environments**]: Integrated environments; D.3.4 [**Processors**]: Debuggers; H.2.3 [**Languages**]: Query languages

General Terms Algorithms, Design, Human Factors, Languages

Keywords Querypoint, TraceQuery, Conditional breakpoint, watchpoints, Locating defects

1. Introduction

Software debugging is an inevitable part of software development. Dealing with bugs is the everyday work of software developers. Debugging is 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. According to [9], developers spend about fifty percent of their time debugging. This shows the importance of improving tools and techniques used by developers for debugging.

The examination of buggy execution is necessary for locating defects that cause a bug. Developers employ different approaches to navigate over buggy execution. Breakpoints are one of the basic tools in this regard. Breakpoints (on source lines) and watchpoints (on value changes) let developers pause program execution at determined points and inspect the program state. However, breakpoint-debugging suffers from serious issues. First, the developer has to resume program execution after every pause. This makes debugging an unpleasant task when a breakpoint is hit several times before reaching the desired point. Second, the developer gets easily lost when the number of breakpoints and their occurrences increase. Third, breakpoints are naturally built for forward navigation and developers have to reason from effects they see back to causes.

Breakpoint-based debuggers support a few basic features to mitigate the two first problems. First, they let the developer pass a number of breakpoints to hit with no stop (sometimes called *hit count* or *pass count*). Second, they let the developer define additional conditions for breakpoints and therefore make a smaller set of target breakpoint hits. Unfortunately, these features are neither effective enough nor practical enough to fulfill developers' needs in debugging [15]. In particular, moving backwards to find causes remains tedious.

Omniscient debuggers have been proposed as a solution for the problems of breakpoint-debugging [10]. 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 [15]. 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.

In this paper we introduce a new kind of compound conditional breakpoint based on iterative program re-execution we will call a *Querypoint*. A *Querypoint* is a compound conditional breakpoint relating two execution points. Developers specify Querypoints relative to a successfully paused conventional breakpoint or conventional watchpoint or to another Querypoint. The relative conditions contain runtime data values, like the last time a value was changed, or program statements, like the last conditional branch. The Querypoint combines the program state information from two execution points in the same execution; Querypoints can be chained to work backwards from effects to causes in a program.

The name *Querypoint* combines the query concept from logging approaches with the point concept from breakpoints. Ultimately our goal is to show that these Querypoints combine the flexibility of conventional breakpoint debugging with the in-depth analysis possible with omniscient logging approaches.

Our contribution here includes the basic *Querypoint* concept, a unification of ideas from breakpoint and logging-based debugging. Like breakpoints we stop the live program at a point where we know the state is not correct; like logging-based debugging, we repeat queries on the state-changes leading up to this point. Live queries sample any variations in execution similar to the way the variations occur for users, we need not wait for a time consuming full-logging run, and we can build directly on existing breakpoint debuggers. We describe two sample Querypoints, *lastChange* and *lastCondition*, and a description of our implementation of these Querypoints. To demonstrace that Querypoints are feasible we describe a prototype that finds a bug in a graphical program analyzed previously with Whyline[9], one of the new logging-based debuggers.

2. Querypoint Introduction

Typically a developer uses a breakpoint to examine the program state and observes some values which seem incorrect according to their understanding of the program. Unless the suspect values are fully defined by code at the breakpoint, the developer seeks to understand the operations which caused the suspect values. In breakpoint debugging, the next step involves setting breakpoints or watchpoints and re-executing the program, hoping to stop the execution where those problematic operations occurred.

In *Querypoint debugging*, we query for the values during re-execution. Querypoint debugging begins from a halted program execution, for example, from a conventional breakpoint. Each re-execution causes the debugger to interrupt the program at some points in the execution and gather information, ultimately halting again at the same logical place, e.g., at the conventional breakpoint. The interrupt points are chosen by the debugger based on the constraints in the Querypoint, as we describe in section 5. The information we want from the re-execution, say "what caused this `foo` to be `null`", determines were we interrupt execution, e.g. where `foo` is changed. We don't halt at these interrupt points because we can't tell which point immediately preceded our breakpoint until we hit it again. When we again arrive at the conventional breakpoint, we select the correct values from among the values collected at those interrupt points to show the developer. The developer sees the causes for suspect values without repeated manual insertion of watchpoint or breakpoints.

We illustrate the idea with an introductory example. The example demonstrates a buggy java program (Figure 1) and the Querypoint debugging steps taken by developer before locating the defect (figure 2). This example resembles a real case; however it is simplified for presentation purposes.

The program processes a list in two consecutive loops and calculates and sets new values for each item in the list. Every item in the list has a boolean field with name `bar`.

```
   // first loop
1     for (Object record : list){
2        record.bar = true;
3        try{
4             stmt1;
5             stmt2;
7             record.bar = expr1;
8             stmt3;
9         }catch(Exception exp){
10           //ignore
11       }
12   }
13   Object foo;
  // second loop
14   for (Object record : list) {
15       if (record.bar || cond) {
16           record.bar = false;
17           foo = expr2;
18       } else{
19           foo = expr3;
20       }
21       assert (foo != null);
22   }
```

Figure 1. Java Pseudo-code, Introductory Example.

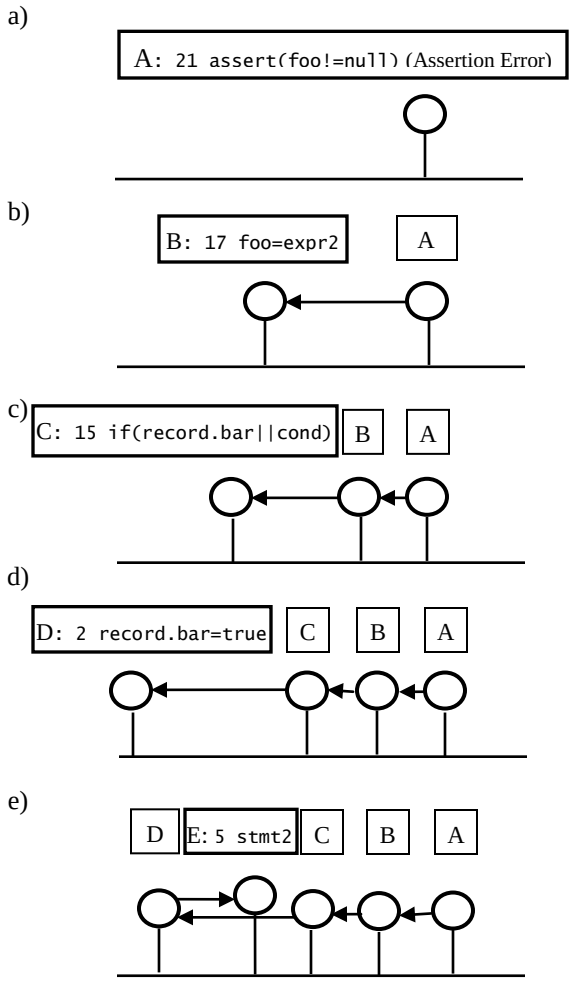

Figure 2. Diagrams of the Querypoint debugging steps for the Introductory example. Each of a-e correspond to a stage in the debugging process. The horizontal line represents the program execution steps; vertical lines schematically illustrate points in the execution selected by Query-points.

This value is set in the first loop and used in if-conditions in the second loop. The bug appears when the program throws an AssertionError exception in the second loop (line 21).

By setting a conventional conditional breakpoint on line 21 with condition (`foo == null`), the developer can halt program execution when line 21 is executed and `foo` is `null`. This point in the sequence of program execution steps is labeled *A* in figure 2.a.

The `null` value is unexpected so the developer seeks its origin. Looking at code, we see that this variable can get a new value either in line 17 or 19. To figure out which path was taken using conventional breakpoints, the developer can put two breakpoints on lines 17 and 19 or a breakpoint on line 15 and then steps forward to reach the target line.

As sometimes happens, many loop iterations succeed before we hit the assertion. To reduce the number of breakpoint hits, the developer has to use one of available features: *pass count* or *additional conditions*. The *pass count* solution is often tedious: the developer has to re-execute multiple times to deduce an appropriate *pass count* for this particular bug. The *conditions* solution needs added code and the accompanying potential for new bugs.

In *Querypoint* debugging, the developer defines a *Querypoint* to identify a point in the execution `foo` changed just preceding execution point *A*. Note that this is not the same as a watchpoint for the value of `foo`: that would be the first point in the execution where `foo` changed. Instead this Querypoint refers to an existing state, the execution point *A*, and *finds only that change of* `foo` *that immediately precedes point A*. In our example, Line 17 is where `foo` got its `null` value and we show that schematically as point *B* in Fig. 2b.

Suppose that the developer expected that execution path goes to the `else` branch (line 18) instead of `if` branch (line 15) and expects `foo` to get its value from `expr3` instead of `expr2`. To determine why the `if-conditions` became true, the developer has to check `record.bar` and `cond` values in line 15 just before point *B*. Consider that the developer cannot know the `record.bar` value at this line due the new value given to it in line 16. The developer defines a second *Querypoint* corresponding to the last effective condition before point *B* (i.e., the last condition that branches to point *B*) and asks for `record.bar` and `cond` values at this point. The developer re-executes the buggy execution and gets the `record.bar` and `cond` values at point *C* (Figure 2.c). We call this a `lastCondition` Querypoint. Here, point *C* is defined dependent to point *B* which itself is de-pendent to *A*.

The value of `record.bar` is true at point *C* and the developer has to find the origin to this wrong value. Consider that due to two separate loops, the last value assignment to the specific `record.bar` object at point *C* is not necessarily the last execution of `record.bar` assignment lines (i.e., lines 2 and 7) before point *C*. These assignment lines might be executed many times for other items in the list before reaching point *C*. Again, the developer defines a new *Querypoint* as the last change of `record.bar` at point *C*. This third Querypoint builds on the other two. The developer re-executes the buggy execution and debugger shows point *D* (Figure 2.d) which is surprisingly corresponding to line 2 where `record.bar` is initiated.

Thus an exception must have occurred before line 7 in the `try-catch` block and prevents line 7 execution. To identify the statement which causes the exception, the developer defines a fourth *Querypoint* corresponding to the exception that occurs after point D and in the try-catch block. The developer re-executes the buggy execution and debugger shows point *E* (Figure 2.e) which is corresponding to line 5 and the defect point.

This example illustrates the critical aspects of *Querypoint* debugging:

1. The workflow resembles breakpoint debugging, with cycles of data examination and re-execution, with breaks at new points.
2. The internal operation resembles trace-based omniscient debugging, with the debugger applying queries to the program data without always halting.
3. The Querypoints related to each other, forming an explicit chain from effects back to causes, helping to centralize the information of a debugging session.

Every Querypoint re-execution returns us to the same point in the program execution, but each time we have gathered information from earlier points, working our way back toward the defect. As we discuss in Sec. 6, developers can choose to combine breakpoints with Querypoints, moving the halted execution effectively backwards.

3. TraceQueries

To implement a Querypoint we define a *TraceQuery*, A TraceQuery is the operational equivalent of a Querypoint, a conversion of the relative and symbolic definition in the Querypoint to a series of debugger breakpoints, watchpoints, and runtime constraint tests. Each Querypoint is translated into a traceQuery; each traceQuery relates low level things the debugger can implement: breakpoints, watchpoints, and constraints.

Conventional breakpoints can be considered as queries with constraints that select a set of points on execution trace. Simple breakpoints and watchpoints are defined by structural constraints (e.g., line number, method name, field name, exception class, etc.) for an event type (e.g., line execution, method call, field value change, etc.). Conditional breakpoints let developers to add dynamic constraints based on the program state and in this way leverage runtime data for filtering unwanted points. TraceQuery is a generalization to the conditional breakpoint or watchpoint concept that includes dynamic constraints between two points of execution.

To explain the traceQuery idea, we need to define a few concepts. An *Execution trace* consists of the ordered list of executed instructions during program execution. In this paper we focus on common bugs which do not depend upon process or thread interleaving. For every instruction in the execution trace, we define a *point* corresponding to the program state before the instruction execution. An *event* is a special change to program state and specifies an interval on the execution trace. A value assignment which is only one instruction is the simplest form of an event. We can consider other types of events such as method call or object creation. Events are usually represented by the point immediately preceding them.

A traceQuery combines an event type, a set of constraints to be tested at each such event, and the points selected by the query. We name every point selected by a traceQuery in execution trace an *instance* of the traceQuery and all instances of a traceQuery form the *result set* of traceQuery. An *index* is an integer that uniquely identifies one instance in traceQuery's result set. A non-negative index is corresponding to the instance position from the beginning of the result set. A negative index is the instance position from the end of the result set.

For example, if the event in the traceQuery is a function call, then index zero is the first time the function is called and minus one is the last time it is called. The Querypoint and its traceQuery are defined in terms of program state values and stack frames. To refer to objects and variables in heap during an execution we use *global object reference (gor)* with this syntax: `pointid(frame number): object reference`. For example, `P(1):x.y` refers to field `y` of variable or field `x` in the second frame (the newest stack frame is numbered zero) at point *P*. The oldest stack frame is number -1. If the *pointid* is not specified, it means the current point should be considered. If frame number is not specified, it will be assumed zero. If object reference starts with a dot, it refers to an object accessible through the event. For example, if the event type is *fieldchanged*, the field's owner object is specified by `.owner`.

We define two kinds of Querypoint in this paper:

- `lastChange(global object reference)` *: D*efines the point corresponding to the last value change of global object reference on the execution trace. In the last section's example, point *B* is defined by `lastChange(A(0):foo)` and point *D* is defined by `lastChange(C(0):record.bar)`.
- `lastCondition(pointid)` : Defines the point corresponding to the last condition that branches to this point. In the last section's example, point *C* is defined by `lastCondition(B)`.

We define three types of inter-point constraints used for translating defined Querypoints to traceQueries:

- `before(pointid)` : means a point is selected by the traceQuery if it happens before the point. We call this *before* constraint.
- `sameness(gor1, gor2)` : assures that two object references refer to one object. We call this *sameness* constraint.
- `mayAffect(pointid)` : means a point is selected by the traceQuery if the sequence of method calls in its callstack matches to the beginning of the sequence of method calls in the point's callstack. We call it *mayAffect* constraint.

In the next section we explain how debugger translates a Querypoint to a traceQuery employing these constraints.

Point	Querypoint	TraceQuery	Index
B	lastChange(A(0):foo)	interrupt on lines 13, 17 and 19, sameness((-1):this, P(-1):this), mayAffect(A), before(A)	-1
C	lastCondition(B)	interrupt on line 15, sameness((-1):this, P(-1):this), mayAffect(B), before(B)	-1
D	lastChange (C(0):record.bar)	fieldchanged interrupt on field *bar* in the *record*'s class, before(C),sameness(.owner, C:record)	-1

Table 1. Translation of Querypoints *B*, *C* and *D* to TraceQueries

4. Querypoint to TraceQuery

To locate a Querypoint, debugger translates it to a traceQuery with an index. We explain how two defined. Querypoints in the previous section translated to traceQuery. We separate *lastChange* Querypoint to two cases depending on whether the global object reference refers to a field or variable. For all cases the associated index is -1 which means the last instance in the trace query result set.

4.1.1 lastChange(field)

Assume that the global object reference defined in this form: `P(n):objectref` and refers to a field. *P* is a previously defined point and *n* is the frame number. The debugger translates this Querypoint to a traceQuery with *field changed* event type and two constraints, `before(P)` and `sameness(Q:.owner,P:fieldOwner)`.

The first constraint assures that this traceQuery only selects points before *P* so index -1 exactly refers to the last change. The second constraint assures that the field's owners are the same object. Debugger finds the right class for owner object from runtime data at point *P*. Point *D* in the introductory example (`lastChange(C(0):record.bar)`) is such Querypoint which is translated in Table 1.

4.1.2 lastChange(variable)

Assume that the global object reference defined in this form: `P(n):objectref` and refers to a variable. The debugger translates this Querypoint to a traceQuery which includes the variable definition statement, all the variable assignment statements in the variable block and three constraints, `before(P)`, `sameness((-1):this, P(-1):this)` and `mayAffect(P)`.

The first constraint assures that this traceQuery only selects points before *P* so index -1 exactly refers to the last change. The second constraint assures that selected point occurs in the same thread as *P* occurs. The last constraint excludes all similar variable assignments happen in lower frames. The need for the *mayAffect* constraint can be illustrated by simple recursive call. If we have *x* in method *m()* and this method calls itself recursively, then *x* changes many times. To correctly choose the last event we exclude the recursive frames.

Point *B* in the introductory example is such Querypoint and it is translated in Table 1. Depending on the underlying debugger technology, the interrupts may be considered watchpoints (on local variables) or breakpoints (on lines where value assignments are made).

4.1.3 lastCondition

Assume that we have this Querypoint: `lastCondition(P)` where *P* is a previously defined point. Having call stack at point *P* and program source code (or bytecode) it is possible to find the statement corresponding to the last condition. If we consider the code in all methods in call stack as one big block, then the last execution is one of the forks surrounding *P*. In most cases the last executed condition is the most internal fork containing P. The exception is a do-while loop:

```
if(cond1){
   do{
      XX_P_XX;
   }while(cond2);
 }
```

where it is not clear from the callstack alone whether P happens in the first or next iteration. The last condition may be the do-while condition or the outer fork surrounding it. The last condition may be do-while condition or the outer fork surrounding that. If the surrending fork is again a do-while loop, the outer fork should also be considered. This process finally specifies all statements that may be the last condition branched to point *P*. Debugger translates the Querypoint to a traceQuery which includes all the fork statements resulted from mentioned process and three constraints, `before(P)`, `mayAffect(P)` and `sameness((-1):this,P(-1):this)`.

The first constraint assures that this traceQuery only selects points before *P* so index -1 exactly refers to the last condition. The second constraint assures that selected point occurs in the same thread as *P* occurs. The last constraint excludes all similar branches happen in lower frames. Point *C* is such Querypoint and its translation can be found in Table 1.

5. Implementation

We implemented a prototype of Querypoint debugger for *Java*. This prototype works based on iterative program re-execution. Whenever the developer introduces a new

Querypoint, debugger adds it to Debug Model, translating each Querypoint to a traceQuery as explained in the previous section and updates a *dependency graph*. The dependency graph represents dependencies between Querypoints created by inter-point constraints. A new Querypoint can only refer to previously defined Querypoints and they are evaluated in order, so the graph is acyclic. In the introductory example, point *B* is dependent to point *A* due to three constraints conditions: `before`, `sameness` and `mayAffect`. The dependency graph is used to check interpoint constraints. Point B happens before point *A* and therefore none of the `sameness` and `mayaffect` constraints can be checked before debugger locates point *A*. To manage this situation debugger keeps a list of all points have the chance to be point *B*. When debugger locates point *A*, it checks the constraint for all points in the list and removes those that don't satisfy these constraints.

While building the dependency graph, the debugger also make a list of data should be collected at every point. For example to check `sameness` constraint, the id of object should be stored for both object references. For example due to sameness dependency between points D and C, the object id of `.owner` for every potential instance for point D and the object id of `record` for every potential instance for point C should be stored.

```
While (there is any unlocated Querypoint)
   Re-execute the buggy execution.
   While (there is any event)
      If (event is classload)
         Set required breakpoints for the class.
      If (event is new Querypoint by the user)
         Add it to debug model.
         Translate Querypoint to traceQuery.
         Update dependency graph.
      If (event is breakpoint hit)
         Find correponding traceQuery.
         Check runtime conditions.
         If there is any remained condition
            Add it to potential list
         Else
            Add it to resultSet
            If any queryopint matches
               Find all dependent Querypoints.
               Check dependent conditions for them.
               If any new queryoint matches,
                                 Redo this step.
      Resume the execution;
```

Figure 3. Outline for Locating Querypoints.

After adding a Querypoint to Debug Model, the debugger re-executes the program and monitors the execution. The overall process is outlined in Fig. 3. Our prototype uses *Java Debug Interface (JDI)* to launch to the debuggee program and allows the user to add queries to break program execution and to re-execute. Debugger finds the result of a traceQuery in three stages. First, it sets breakpoints or watchtpoints at statements which are defined by the trace query. To do this, it listens to class load events and sets necessary breakpoints for every loaded class. We assume that debugger directly has access to bytecode of all classes loaded to JVM.

Second, it monitors the execution and whenever a breakpoint is hit, it finds the corresponding traceQuery and checks conditions, skipping any conditions dependent on points later in the execution. If all these conditions are satisfied, it keeps the call stack structure and collects required data like values of fields or variables. If any condition was skipped, this point is added to the list of potential instances for the traceQuery (so that these instances can be tested later). Otherwise this point is added to the resultSet of traceQuery.

Whenever an instance is added to a traceQuery resultset, debugger checks whether this instance matches to a Querypoint by checking the index. If the point matches to a Querypoint debugger assigns this point to the Querypoint. Then debugger finds all dependent Querypoints to this Querypoint and refines the list of potential points by checking constraints. The debugger recursively repeat this step until no new Querypoint is added.

6. Reproducible Non-deterministic Execution

Thus far we have not discussed problems caused by multiple threads or other sources of non-deterministic executions. We want to explain why we believe Querypoint debugging is robust in the practically important case where a bug is reproducible even though the execution may not be deterministic.

Because Querypoints 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, Querypoints are more effective than breakpoints. In the case of *logically deterministic* program execution, we can use the result from a Querypoint operation to set a conditional breakpoint then re-execute the program to position the execution trace *backwards* from our first breakpoint. (This may be a useful adjunct for Querypoint debuggers to implement, but this backwards motion in the execution

logic is not required for Querypoint debugging.) Thus in this case Querypoint debugging can do the same kinds of things as conventional breakpoints just more automatically.

In the case of a *non-deterministic* program, a Querypoint is not equivalent to any series of conventional watchpoints or breakpoints. 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. Suppose 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 Querypoint method co-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. Debugging a Painting Application

To demonstrate Querypoints are feasible we use a simple painting environment introduced in [9]. This program lets the user to draw graphics on the right white pane by pressing and releasing the mouse button. User can selects one of available drawing modes (e.g., pencil, eraser, line). The color is also specified by three sliders which are corresponding to main three colors. Figure 4 shows the interface of this environment.

The bug happens when user changes the sliders positions to draw a blue line (i.e., when both the red and green sliders are at the left side while the blue slider is at the right side). In this configuration, new lines' color is black instead of blue. To reproduce the bug it is enough to restart the program and put sliders in the same positions and draw a line.

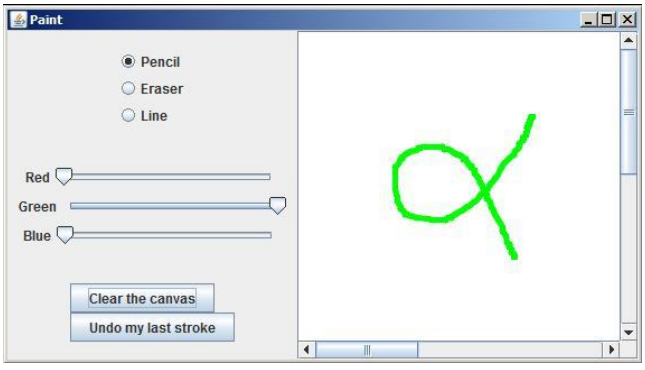

Figure 4. Painting Application.

We assume the developer set a breakpoint that halts execution after the bug appears. We'll call this point in the execution trace *P1*. Figure 6 shows *P1* definition as well as the call stack and source code related to this point.
Using the debugger, the developer sees that *g.foregroundcolor* is black, an unexpected value and possibly related to the defect. To explore this possiblity, the developers asks for `lastChange(P1:foregroundcolor)`.
Debugger reads the Java class of object g and puts a watchpoint over the foregroundColor field. Every time this watchpoint hits, the debugger stores the call stack and an integer uniquely identifying the object g within this execution. When execution again reaches *P1*, the debugger works backwards through the traceQuery's potential list stopping at the instance with a stored object id equals to the object id of the reference `g`. We'll name this instance *P2*. Figure 6 shows the *P2* definition, both the Querypoint and its translation into a watchpoint and a constraint as described in section 3. Fig. 6 also shows collected data at both *P1* and *P2* at the last execution.

P1 and *P2* happen in `EventDispatch` thread. Whenever an event is fired which requires updating graphical interface this thread calls `repaint` method on the parent component and it recursively calls this method on children should be updated. The `repaint` method is called non-deterministically. Therefore, to locate *P2* by traditional watchpoints, the developer has to pause at every hit, writes down call stack, object ids and then resume and later compare all these collected data to find out which ones are related.
The source code for point *P2* shows that the `foregroundColor` change depends on the value of the member field `color`. Consequently the developer sets a new Querypoint for the last change of field `color` of the `PencilPaint` object. The next point, *P3*, is defined by `lastChange(P2(1):color)`.

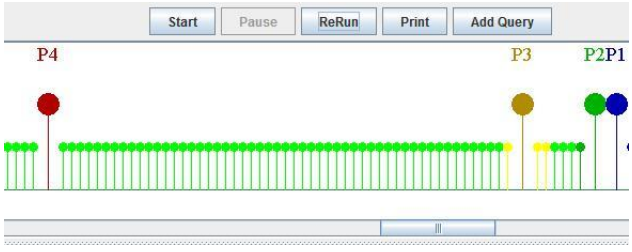

Figure 5. Screenshot of the execution-trace viewer in a prototype Querypoint debugger applied to the application shown in Figure 4.

Point P1

Definition

```
Line Breakpoint: SunGraphics2D, line 2098
Condition: this.getColor() == Color.BLACK
Hit index: 0
```

Call stack

```
SunGraphics2D.java:drawLine():2098
PencilPaint.java:paint():56
PaintCanvas.java:paintComponent():42
JComponent.java:paint():1027
...
EventDispatchThread.java:run():122
```

Source Code

```
5  public class PencilPaint extends PaintObject {
 …
47     public void paint(Graphics2D g) {
…
56       g.drawLine((int) one.getX(), (int) one.getY(),
57                  (int) two.getX(), (int) two.getY());
…
60    }
61 }
```

Point P2

Querypoint

```
Querypoint: lastChange(P1:foregroundColor)
```

TraceQuery

```
Event: FieldChange SunGraphics2D.foregroundColor
Conditions: sameness(.owner, P1:this), before(P1)
Index:-1
```

Call stack

```
SunGraphics2D.java:setColor():1653
PencilPaint.java:paint():50
PaintCanvas.java:paintComponent():42
JComponent.java:paint():1027
...
EventDispatchThread.java:run():122
```

Source Code

```
5  public class PencilPaint extends PaintObject {
 …
47    public void paint(Graphics2D g) {
…
50      g.setColor(color);
…
56      g.drawLine((int) one.getX(), (int) one.getY(),
57                 (int)  two.getX(), (int)
two.getY());
…
60    }
61 }
```

Collected Data

```
P1:this <- ObjectId:1539,Class:SunGraphics2D
P2:.owner  <- ObjectId:1539,Class:SunGraphics2D
```

Point P3

Querypoint

```
Querypoint: lastChange(p2(1):color)
```

TraceQuery

```
Event: FieldChange PencilPaint.color
Conditions: sameness(.owner, P2(1):this), before(P2)
Index:-1
```

Call stack

```
PaintObject.java:setColor():10
PaintObjectConstructor.java:mousePressed():60
Component.java:processMouseMotionEvent():6261
JComponent.java:processMouseMotionEvent():3283
...
EventDispatchThread.java:run():122
```

Source Code

```
9  public class PaintObjectConstructor implements
             MouseListener, MouseMotionListener {
…
13 private PaintObject temporaryObject;
…
52 public void mousePressed(MouseEvent e) {
…
60      temporaryObject.setColor(color);
…
66      }
…
100}
```

Collected Data

```
P1:this <- ObjectId:1550,Class:SunGraphics2D
P2:.owner  <- ObjectId:1550,Class:SunGraphics2D
P2(1):this <- ObjectId:1536,Class:PencilPaint
P3:.owner  <- ObjectId:1536,Class:PencilPaint
```

Point P4

Querypoint

```
Querypoint: lastChange(p3(1):color)
```

TraceQuery

```
Event: FieldChange PaintObjectConstructor.color
Conditions: sameness(.owner, P3(1):this), before(P2)
Index:-1
```

Call stack

```
PaintObjectConstructor.java:setColor():26
PaintWindow.java:stateChanged():26
JSlider.java:fireStateChanged():420
JSlider.java:stateChanged():337
…
EventDispatchThread.java:run():122
```

Source Code

```
10 public class PaintWindow extends JFrame
     implements PaintObjectConstructorListener {
…
23  private PaintObjectConstructor objectConstructor;
…
25  public void stateChanged(ChangeEvent changeEvent) {
26        objectConstructor.setColor(new Color(
             rSlider.getValue(),
27           gSlider.getValue(), gSlider.getValue()));
28       repaint();
29  }
 …
165}
```

Figure 6. Visited points (*P1* to *P4*), their definition, call stack , source code and collected data by debugger.

The debugger translates the Querypoint into a traceQuery and re-executes. The process is similar to the steps for *P2*. The only difference is that *P3* is dependent to *P2*, and *P2* is dependent to *P1*. Therefore, debugger has to wait until the execution reaches P1 and then it can recognize *P2* and respectively *P3*. Figure 6, shows *P3* and col lected data at all three points. Managing this case by regu lar watchpoints is even harder, because developer has to keep track of hits for two different watchpoints.

After locating *P3*, developer seeks for the last change of the field `color` of `PaintObjectConstructor`. This point, *P4*, is corresponding to defect (Figure 6). As you see in the code the value of green slider is used as the value of blue slider and it's the reason for wrong color.

Figure 5 is a screenshot of the execution trace viewer in the prototype debugger, taken after applying the four Querypoints described above. The smaller circles are those points which are inspected but they have not satisfied all the constraints. The bigger circles demonstrate points *P1* to *P4*. Due to non-determinism, none of the points *P2*, *P3* and *P4* are recognized before reaching *P1*. Therefore developer cannot inspect program state by pausing at those points. Instead, developer uses this interface and asks debugger to collect needed data in the next re-execution. Moreover the developer can define new Querypoints from a point or print collected data at a point, by selecting associated circle. Circles provide handles to the developer to work with points which are not physically available but developer knows them.

8. Related Work

We have split the related work in three subsections. We first compare our approach to other similar approaches which attempt to provide the capability of backward movement on buggy execution. Then we look at runtime trace monitoring, and automated debugging.

Querypoint debugging 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 buggy 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. Querypoint debugging collects data on re-execution but this data is limited to the current queries of developer.

Among replay-based debuggers we compare to bdb [4] and reverse watchpoint [14]. 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, 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 beginning [14]. Similar to bdb it relies on deterministic replay and uses a counter to correctly locate a point in the next execution.

Querypoint also counts during replay but rather than halting the execution to allow the developer to investigate program state, it records query results to investigate program state. For common deterministic bugs, these two approaches should be similar; after a re-execution, Querypoint can support backwards step or backwards watchpoint at the cost of one additonal replay. For non-deterministic cases, Querypoint reports correct values from one path of execution (also like the other two), but if the developer asks for more information, causing another re-execution, Querypoint will report correct values from this new path. Since Querypoint does not require deterministic replay it is much simplier to implement and in future we may be able to support comparisons of query result from different reexecutions as a tool for solving non-deterministic bugs.

Among logging-based approaches are "omniscient" debuggers ODB[10] and Unstuck[8]. Both approaches keep the log history in memory and hence can only record and store the complete history for a short period of time. A more scalable approach 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. 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[9] 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 interface in addition to program trace and provides answers to *why* and *why not* questions to the user. Jive[6] 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; both provide models for extending Querypoint debugging to obtain a better user interface while retaining the flexible conventional replay model of debugging.

A recent work by Lienhard et al.[12] suggests virtual machine level support for keeping the object flow. It replaces 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 aliases) 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 information 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.

Querypoint debugging 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 [3] to help with this problem in the case of graphical and event based systems.

Lencevicius et al. proposed Query-based debugging which consists in identifying events that match a query expressed in a high-level language [11]. In their work, a query defines a set of constraint for the program state and debugger finds those execution points which satisfy these conditions. In contrast, our approach is focused on navigation from already defined points with a high-level language.

PQL[13], PTQL[7], JavaMOP[5], QVM[2] and Tracematches[1] provide means to find a sequence of events in executions that matches to a determined pattern. Though these approaches are similar to our approach in locating a point with specific characteristics, they are not developed for debugging but finding similar patterns of events (e.g, to prevent similar bugs) or verifying some properties about the execution.

9. Conclusions and Future Work

We have described Querypoints, an extension of conditional breakpoints supporting queries that extract information from points in the execution logically earlier in the program execution A *Querypoint* is the high-level query building on previous Querypoints and on information obtained in previous queries. The goal is that developers specify new points by Querypoints instead of setting low-level breakpoints and watchpoints.

Developers use lastCondition Querypoints to examine program state before the last branch, analogous to a backwards single step; they use lastChange Querypoints to examine program state at a state change, analogous to a backwards watchpoint. It is the debugger's responsibility to correctly and efficiently locate these backwards points and developer has not to deal with filtering unwanted breakpoint hits and making complex breakpoints. Moreover, this high-level provides an abstract central reference view over the buggy execution during debugging. Information about the execution accumulates in the Querypoints as we work backwards towards the cause of the bug.

By using queries with constraints Querypoints combine some of the advantages of breakpoint debugging with some from query-based debuggers. Querypoints can be added onto existing breakpoint debuggers and the re-execution behavior should be familiar to developers. We don't require deterministic replay so the infrastructure for replaying external inputs is not required. We only record selected information on each re-execution so memory overhead should resemble breakpoint debugger overheads. We have demonstrated using the queries for two forms of backwards movement, lastChange and lastComparison. We believe more kinds of queries can be supported for more sophisticated types of logical motion or runtime dynamic analysis.

Querypoints also potentially share some of the drawbacks of query-based debuggers: forming queries can be technically demanding limiting the appeal of the solution for already overburdened developers and complex queries could impact debugger performance in unpredictable ways. The WhyLine debugger [9] points to one path for avoiding these problems: the debugger can present the query possibilities to the developer in terms of concrete program constructs rather than abstractions. In this way we may combine the flexibility of a breakpoint solution with some of the power of the 'omniscient' debugger approaches.

Practical implementations of Querypoint debugging will need to explore the space-time tradeoffs of Querypoints. At one extreme we store very little data from each interrupt and the developer must issue new queries and re-execute to learn about the state of the program at earlier execution points. At the other extreme we checkpoint the execution so all the information about the state is available to the developer after each re-execution. More likely will be simple compromises where we record the local variables, arguments to functions, and objects referenced in a function or where simple analysis of the source guides data collection.

References

[1] Allan, C., Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. 2005. Adding trace matching with free variables to AspectJ. *SIGPLAN Not.* 40, 10 (Oct. 2005), 345-364. DOI= http://doi.acm.org/10.1145/1103845.1094839

[2] Arnold, M., Vechev, M., and Yahav, E. 2008. QVM: an efficient runtime for detecting defects in deployed systems. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications* (Nashville, TN, USA, October 19 - 23, 2008). OOPSLA '08. ACM, New York, NY, 143-162. DOI= http://doi.acm.org/10.1145/1449764.1449776

[3] Barton, J.J. and Odvarko, J. 2010. Dynamic and Graphical Web Page Breakpoints. In *Proceedings of the 19th international Conference on World Wide Web*. WWW '10.

[4] Boothe, B. 2000. Efficient algorithms for bidirectional debugging. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada, June 18 - 21, 2000). PLDI '00. ACM, New York, NY, 299-310. DOI= http://doi.acm.org/10.1145/349299.349339

[5] Chen, F. and Roşu, G. 2007. Mop: an efficient and generic runtime verification framework. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications* (Montreal, Quebec, Canada, October 21 - 25, 2007). OOPSLA '07. ACM, New York, NY, 569-588. DOI= http://doi.acm.org/10.1145/1297027.1297069

[6] Czyz, J. K. and Jayaraman, B. 2007. Declarative and visual debugging in Eclipse. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology Exchange* (Montreal, Quebec, Canada, October 21 - 21, 2007). eclipse '07. ACM, New York, NY, 31-35. DOI= http://doi.acm.org/10.1145/1328279.1328286

[7] Goldsmith, S. F., O'Callahan, R., and Aiken, A. 2005. Relational queries over program traces. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA, October 16 - 20, 2005). OOPSLA '05. ACM, New York, NY, 385-402. DOI= http://doi.acm.org/10.1145/1094811.1094841

[8] Hofer, C., Denker, M., Ducasse, S.: Design and implementation of a backward-in-time debugger. In *Proceedings of NODE'06*. Volume P-88 of Lecture Notes in Informatics,Gesellschaft f¨ur Informatik (GI) (September 2006) 17–32

[9] Ko, A. J. and Myers, B. A. 2008. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international Conference on Software Engineering* (Leipzig, Germany, May 10 - 18, 2008). ICSE '08. ACM, New York, NY, 301-310. DOI= http://doi.acm.org/10.1145/1368088.1368130

[10] Lewis, B. and Ducasse, M. 2003. 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* (Anaheim, CA, USA, October 26 - 30, 2003). OOPSLA '03. ACM, New York, NY, 96-97. DOI= http://doi.acm.org/10.1145/949344.949367

[11] Lencevicius, R., Hölzle, U., and Singh, A. K. 2003. Dynamic Query-Based Debugging of Object-Oriented Programs. Automated Software Engg. 10, 1 (Jan. 2003), 39-74. DOI= http://dx.doi.org/10.1023/A:1021816917888

[12] Lienhard, A., Grba, T., and Nierstrasz, O. Practical Object-Oriented Back-in-Time Debugging ECOOP '08: *Proceedings of the 22nd European conference on Object-Oriented Programming*, Springer-Verlag, 2008, 592-615

[13] Martin, M., Livshits, B., and Lam, M. S. 2005. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA, October 16 - 20, 2005). OOPSLA '05. ACM, New York, NY, 365-383. DOI= http://doi.acm.org/10.1145/1094811.1094840

[14] Maruyama, K., Terada, M.: Debugging with reverse watchpoint. In *Proceedings of the Third International Conference on Quality Software* (QSIC'03), Washington, DC, USA, IEEE Computer Society (2003) 116

[15] Pothier, G., Tanter, É., and Piquer, J. 2007. Scalable omniscient debugging. SIGPLAN Not. 42, 10 (Oct. 2007), 535-552. DOI= http://doi.acm.org/10.1145/1297105.1297067