Query-Point Debugging

Salman Mirghasemi

School of Computer and Communication Sciences,
Ecole Polytechnique Fédérale de Lausanne,
Switzerland
salman.mirghasemi@epfl.ch

Abstract

Software Debugging is still one of the most challenging and time consuming aspects of software development. Monitoring the software behavior and finding the causes of this behavior are located at the center of debugging process. Although many tools and techniques have been proposed to support developers in this job, none of them could replace or improve the traditional debugging methods. This paper presents Query-Point debugging as a new debugging approach and explains how it can facilitate debugging for developers.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Algorithms, Design, Human Factors, Languages

Keywords dynamic breakpoint assignment, execution monitoring, locating defects, program traces, query

1. Problem Statement

Software Debugging has little changed during the past decades and remained one of the most challenging and time consuming aspects of software engineering. Today, debugging is much more an art and the predominant techniques for finding bugs are data gathering (e.g., print statements) and hand simulation. Software developers spend huge amounts of time, up to half of their time, debugging. Finding and fixing bugs faster and more effectively directly increases productivity and can improve program quality by eliminating more defects with available resources [2].

Fixing a bug consists of three main stages, *reproducing the bug*, *locating the defects* and finally *fixing the defects*. The second stage, locating defects, located at the heart of debugging process and is also the focus of this paper. Locating defects is mainly carried out by the examination of buggy execution, therefore the bug should be reproducible. In this paper, when we talk about debugging and bugs, we mean locating defects and reproducible bugs respectively.

There are two traditional approaches to debugging: *log-based* debugging and *breakpoint-based* debugging. The first approach consists in inserting logging statements within the source code, in order to produce an ad-hoc trace during program execution. This technique exposes the actual history of execution but (a) it requires cumbersome and widespread modifications to the source code, and (b) it does not scale because manual analysis of huge traces is hard. The second approach consists in running the program under a dedicated debugger which allows the programmer to pause the execution at determined points, inspect memory contents, and then continue execution step-by-step. Although not subject to the two issues of log-based debugging, breakpoint-based debugging is limited: when execution is paused, the information about the previous state and activity of the program is limited to introspection of the current call stack [1].

Omniscient debuggers, which are built based on *capture-replay* techniques, have been proposed to overcome mentioned issues. While the advantages of omniscient debugging over traditional approaches are incredibly clear, it has had a very limited impact in production environments, and is still mostly seen as an unrealistic approach [1]. We can briefly mention three main issues about omniscient debuggers. First, capturing the execution trace has a huge overhead over normal execution. Second, the captured execution is different with live execution and many useful data such as GUI and memory heap states are not available to the developer. Third, querying the huge amount of collected data is expensive.

This work provides two main contributions to debugging: (a) *Query-Point debugging*, as a new debugging approach, which provides a systematic method to debugging; and (b) Techniques which let applying Query-Point debugging on live executions instead of captured ones.

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.
ACM 978-1-60558-768-4/09/10.

2. Query-Point Debugging

Before getting into the approach explanation, we need to define a few concepts. We can suppose a hypothetical line, *execution trace line*, corresponding to the program execution trace. An *execution point* is a point on this line. A *query* selects a set of points on the execution trace line.

Checking the correctness of program state, *Check State Problem* (CSP), at a given point is the problem arises many times during debugging and is one main source of debugging complexity. A *buggy* point is an execution point with a data-flow or control-flow problem in the program execution or program state. Consider that the buggy point is determined according to the developer's expectation which can be defined by a set of assertions. Therefore, at least one of the defined assertions at a buggy point should be violated. This definition might be inconsistent (due to defects in the developer's mental model) or it might change during debugging.

Two basic approaches are used for the navigation on buggy execution trace: forward and backward check-search. Each one of these approaches has its own features and usually a mixture of both is used. Answering to CSP is easier in forward check-search while backward check-search is more natural for finding the cause of an error.

Query-Point debugging works based on a few simple facts:

1. Locating defects problem can be reduced to locating the first buggy point (*defect point*) on the buggy execution trace line and providing an explanation of how the problem at the defect point eventually causes the bug.
2. From the defect point to the point the bug appears, all points are buggy points. Consider that program state consists of all data affect the program execution. For example when a program stores data in a database table, the table should also be considered as part of program state.
3. To answer to CSP at a point, a developer needs to know at which stage of computation scenario the point is located. Then the developer can use collected data to determine expectations which become the basis for answering to CSP at the point.

Query-Point debugging is an iterative process in which the developer incrementally increase her knowledge about the buggy execution. Execution trace line is the central reference view during debugging and, points on this line and their associated assertions are the developer's discoveries up to now. In addition to keeping track of past steps, it helps developer to only focus on the interval from the last non-buggy point to first buggy point. Queries are appropriate means for specifying new points.

The Query-Point debugging process can be explained in a few high-level steps:

```
Query-Point Debugging:
  1) Convert the bug to a buggy point on the execution
  trace line.
  2) Start backward check-search from the first buggy point
  or forward check-search from the last non-buggy point.
  3) Answer to CSP at 'next point'. Go to step 2.
Forward Check-Search:
  1) Define the next point.
Backward Check-Search:
  1) If there is a data-flow problem:
  1.1)A wrong value
  next point: the last place the
  variable has been changed.
  2) If there is a control-flow problem:
  2.1)A wrong instruction.
  next point: last fork point.
  2.2)A missed instruction.
  next point: last fork point.
```

To start debugging, the developer should determine at least one buggy point on the execution trace line. This point is usually the point the bug appears. Sometimes the developer has some initial data (e.g., the error location in source code) about this point, therefore the developer can use these initial data to query the execution and find the point. Otherwise the developer has to go through forward check-search to find the first buggy point on the execution trace line. After the first step, the developer continues by adding a new point and therefore reducing the examination interval at every iteration.

3. Implementation Techniques

We briefly explain three main implementation techniques: (a) Querying live execution; (b) Point fixation; and (c) Backward movement. Once a bug is reproducible, the live buggy execution can be used for examination. Querying the live execution is carried out by dynamic breakpoints assignment. It means that, the debugger determines places that might be in the result set of query and checks all these points during the execution. One advantage of querying the live execution is that the developer can define more complex constraints in a boolean method which can be called at every point.

Point fixation means recognizing the same points on the trace line in the following executions. The primary technique for point fixation is using the index of point in the query result set. There are some cases (e.g., when two threads do the same job) that the index in the result set is not enough for finding the same point. In these cases additional data around the point (e.g., a method parameter value) can be used to uniquely define the point. Backward movement can be managed by fixing the current point and re-execution.

Acknowledgments

I would like to thank Claude Petitpierre for advising me in this research.

References

[1] G. Pothier, É. Tanter and J. Piquer. Scalable omniscient debugging. In *OOPSLA*, 2007.

[2] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2005.