BPGen: An Automated Breakpoint Generator for Debugging

Cheng Zhang[1], Dacong Yan[2], Jianjun Zhao[1,3], Yuting Chen[3], Shengqian Yang[3]
[1]Department of Computer Science and Engineering, Shanghai Jiao Tong University
[2]Department of Computer Science and Engineering, The Ohio State University
[3]School of Software, Shanghai Jiao Tong University
cheng.zhang.stap@sjtu.edu.cn, yan@cse.ohio-state.edu, zhao-jj@cs.sjtu.edu.cn,
chenyt@cs.sjtu.edu.cn, michyang@sjtu.edu.cn

ABSTRACT

During debugging processes, breakpoints are frequently used to inspect and understand runtime behaviors of programs. Although most development environments offer convenient breakpoint facilities, the use of these environments usually requires considerable human efforts in order to generate useful breakpoints. Before setting breakpoints or typing breakpoint conditions, developers usually have to make some judgements and hypotheses on the basis of their observations and experience. To reduce this kind of efforts we present a tool, named BPGen, to automatically generate breakpoints for debugging. BPGen uses three well-known dynamic fault localization techniques in tandem to identify suspicious program statements and states, through which both conditional and unconditional breakpoints are generated. BPGen is implemented as an Eclipse plugin for supplementing the existing Eclipse JDT debugger.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

General Terms

Design, Reliability

Keywords

Debugging, Breakpoint

1. INTRODUCTION

Program debugging has long been a laborious task for software developers. To reduce manual efforts for debugging, a variety of automated debugging methods have been proposed and tools have been developed both in academia and in industry, and most of them focus on either detecting the abnormal program execution flows [9, 14] or analyzing the abnormal runtime program states [8, 18]. Nevertheless, the use of these methods and tools in practice still faces strong challenges. One main challenge is that the execution of the program usually does not provide sufficient information to precisely handle various kinds of bugs, some of which may be extremely tricky. As a result it usually requires developers to interactively explore the program executions and fix the potential bugs. That is, developers still need to complete several iterations of observation, making of hypotheses, and verification of the hypotheses during debugging.

Most development environments provide necessary facilities to aid program debugging. Among them, breakpoints are frequently used in practice [13]. With breakpoints, developers can suspend the execution of a program and explore its state space to find clues to bugs. However, during debugging, the developers usually have to ask themselves a question: *where should the breakpoints be set?* Some IDEs, such as Eclipse[2], provide the functionality to export and import breakpoints so that these breakpoints can be reused and be shared among developers. Nevertheless, due to the diversity of debugging tasks, setting of breakpoints still largely relies on the developers' capabilities of observations and judgements on the debugging task in hand and even their experiences of similar situations. Thus it is ordinary that a veteran can set a small number of breakpoints and reveal a bug immediately, whereas a novice may set a number of redundant breakpoints, but still fail to find the bug. For this reason, a set of well predefined breakpoints could be a quite pleasant starting point for the developers to debug.

In this paper, we present a tool, named BPGen (Break-Point Generator), which is used to automatically generate breakpoints on the basis of a synergy of fault localization methods. BPGen is implemented as an Eclipse plugin for supplementing the existing Eclipse JDT debugger. The principle of BPGen is at first to apply the nearest neighbor queries method [14] to a failing test case, which is randomly selected by the developer, in order to find suspicious basic blocks in the program, and then to use dynamic program slicing [11] to produce the corresponding program slice as a supplement for the report of nearest neighbor queries so that the root cause of the basic blocks' suspiciousness can be captured. BPGen generates breakpoints on the conditional statements contained in the program slice. In the last step, BPGen extracts memory graphs [20], compares the graphs in order to identify the most suspicious program states, and generates breakpoint conditions based on these states. The main contributions of this work can be summarized as: 1) a novel approach, which combines nearest neighbor queries

ICSE '10, May 2-8 2010, Cape Town, South Africa

method, dynamic program slicing, and memory graph comparison, to breakpoint generation for aiding program debugging and 2) a tool, called BPGen, which supports the approach in Eclipse JDT debugger.

The rest of this paper is organized as follows. Section 2 describes the underlying approach of BPGen and its implementation and usage. Section 3 compares BPGen with related work. Section 4 concludes this paper and presents the future work.

2. BPGEN: AN AUTOMATED BREAKPOINT GENERATOR

2.1 Process of BPGen

BPGen contains two main activities when generating breakpoints: 1) selecting breakpoint locations using nearest neighbor queries method and dynamic program slicing, and 2) generating a set of breakpoint conditions, each of which is a boolean expression that will be evaluated at runtime.

2.1.1 Selecting Breakpoint Locations

In this work, we assume that a debugging task is to select one failing test case and make it pass by locating and eliminating the corresponding bug. In addition, since line breakpoints are much more frequently used than other kinds of breakpoints (e.g., exception breakpoints, method entry breakpoints and watchpoints) [3], we only focus on generating line breakpoints and their conditions.

In order to select breakpoint locations, we first randomly select a failing test case, and then use the nearest neighbor queries method [14] to find the passing test case nearest to the selected failing test case and also identify the differences between their execution traces. The nearest passing test case is the one whose execution trace most resembles that of the failing test case. The reason for us to focus on the nearest passing test case is that when the execution traces of both failing and passing test cases are similar (but not identical), the differences between the traces are more likely to indicate the bug of interest. On the contrary, if the two execution traces are significantly different, it is not easy to locate the bug by comparing them.

To perform nearest neighbor queries, we collect execution traces by instrumenting the program and then running its accompanying test cases. Execution traces are converted to binary coverage spectra where the number of execution times of a basic block is mapped to 1 or 0 to indicate whether the basic block is executed or not. Then asymmetric Hamming distance is used to compute the distances between execution traces and determine the nearest passing test case. Finally the differences are presented in the form of basic blocks, in which the statements provide some candidate locations for setting breakpoints in terms of the failing test case.

However, setting breakpoints only on the statements in these basic blocks may not help to reveal the real bug, because the execution of the faulty statement may be much earlier than the exhibition of the abnormal execution flow. For this reason, BPGen uses backward dynamic program slicing [11] to trace back to the statements where the program state initially becomes incorrect. We add all the statements reported by the nearest neighbor queries method and the referenced variables to the slicing criterion set, and rerun the failing test case in order to perform the backward dynamic program slicing. Finally BPGen generates breakpoints on the conditional statements in the resultant slice, as these statements are usually relevant to the divergence of execution flow or even the bug.

2.1.2 Generating Breakpoint Conditions

In most debuggers, a breakpoint can hold a condition which is a boolean expression and will be evaluated when the breakpoint is hit. The typical usage of a breakpoint condition is to suspend the execution of program on the breakpoint hit if its condition is evaluated to *true*. Thus a breakpoint holding a proper condition can suspend the execution of program just before or after the program enters an incorrect state. It is especially useful when a breakpoint is hit for a number of times, but only some of the hits are worth focusing on.

BPGen adapts the memory graph comparison [20] to synthesize conditions for breakpoints. Memory graphs can represent runtime program states exhaustively in a structural way, and thus the results of memory graph comparisons are usually precise for indicating incorrect states and convenient for generating breakpoint conditions. Unfortunately, memory graph comparison requires the extraction of memory graphs which usually causes heavy performance overhead when the program states are huge in number and complex in structure. Therefore, it is prohibitively expensive to perform a memory graph comparison at **every** breakpoint hit and a better strategy is necessary.

Given a set of breakpoints $BP = \{bp_1, bp_2, ..., bp_N\}$ and a specific breakpoint $bp_i \in BP$, BPGen collects the runtime values of the corresponding condition expression[1] during the executions of the failing test case and the nearest passing test case. We use $V_i^p = (v_{i,1}^p, v_{i,2}^p, v_{i,3}^p, ..., v_{i,m}^p)$ and $V_i^f = (v_{i,1}^f, v_{i,2}^f, v_{i,3}^f, ..., v_{i,n}^f)$ to note the two sequences of values, where $v_{i,k}^p$ (or $v_{i,k}^f$) stands for the value collected on the kth hit of bp_i during the passing (or failing) execution. Then we define the set of *comparison points* CP_i for bp_i:

$$CP_i = \begin{cases} \{cp_{i,j} | j = min(\{k | v_{i,k}^f = true, v_{i,k}^p = false\})\} \bigcup \\ \quad \{cp_{i,j} | j = max(\{k | v_{i,k}^f = true, v_{i,k}^p = false\})\}, \\ \qquad\qquad \text{if } v_{i,k}^p, v_{i,k}^f \in \{true, false\}; \\ \\ \{cp_{i,j} | j = min(\{k | v_{i,k}^f \neq v_{i,k}^p\})\} \bigcup \\ \quad \{cp_{i,j} | j = max(\{k | v_{i,k}^f \neq v_{i,k}^p\})\}, \quad \text{otherwise} \end{cases}$$

where $0 < k \leq min(m, n)$, max and min compute the maximum and minimum values of k, respectively. Since conditional statements include **if**, **for**, **while**, and **switch**, their condition expressions may have boolean values or values of other types (e.g., integer and enumeration types), which corresponds to the two cases in the definition of CP_i. Then CP_i contains, for the breakpoint location of bp_i, the first and last times (if they exist) when the execution flow of the failing test case diverges from that of the passing test case. Note that we do not consider all the divergent points for efficiency. Based on the definition of comparison points, we provide the following strategy to perform memory graph comparisons and to generate breakpoint conditions.

For each comparison point $cp_{i,j} \in CP_i$, BPGen extracts two memory graphs $G_{i,j}^f$ and $G_{i,j}^p$ when the breakpoint bp_i is hit for the jth time during the failing and passing executions, respectively. Then BPGen compares the two graphs using

[1]Every breakpoint is located at a conditional statement which can be executed for several times during a single execution.

the algorithm in [20] to find out their differences. Based on the graph differences, BPGen composes the breakpoint condition that is a conjunction of boolean expressions in the form of $a == b$, $obj_1.equals(obj_2)$, or $a.length == len$.

On the other hand, some breakpoints do not have corresponding comparison points, because the collected values do not satisfy the conditions in the definition of comparison points. In this case no breakpoint conditions will be generated for them. In terms of a rule in BPGen, conditional breakpoints are enabled by default and unconditional breakpoints are disabled. That is, when conditional breakpoints are hit, their conditions will always be evaluated and the execution may be suspended accordingly, while unconditional breakpoints will never come into effect until the user enables them. We enforce this discrimination in order to 1) distinguish the breakpoints that are able to reveal suspicious program states and 2) reduce the number of breakpoints the user has to inspect during the first iteration of debugging.

2.2 Implementation and Usage

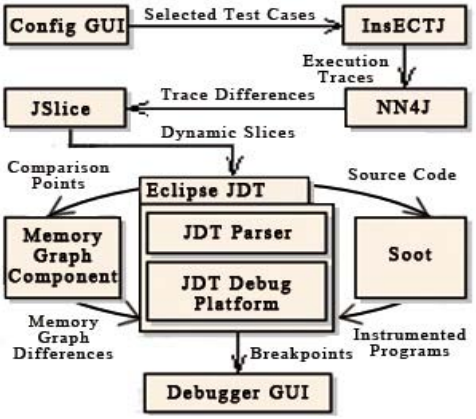

Figure 1: Architecture of BPGen

Implementation. BPGen is implemented as an Eclipse plugin whose architecture is shown in Figure 1. BPGen uses InsECTJ [15] to instrument the program and collect the execution traces. NN4J, an implementation of the nearest neighbor queries method, is used to perform nearest neighbor queries. BPGen uses JSlice [17] to compute dynamic program slices. The Java parser and debug platform provided in Eclipse JDT are used to generate breakpoints and their conditions. To record runtime value sequences, BPGen uses Soot framework [4] to slightly instrument the program at each breakpoint location. The memory graph component of the DDstate plugin [1] is used to implement a breakpoint listener. Once the listener is registered to the debug model, memory graphs will be extracted and compared when related comparison points are reached.

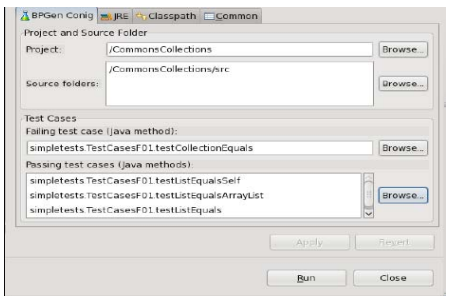

Figure 2: BPGen launch configuration

Usage. To use the BPGen plugin, a developer only needs to specify the failing test case and some passing test cases in a customized GUI as shown in Figure 2. After the **Run** button is pressed, BPGen starts to run in background.

As Figure 3 shows, the final output of BPGen is a group of breakpoints which can be displayed both in the *Breakpoints view* of the *Debug perspective* and in the Java editor showing the corresponding Java source code. Figure 4 shows the generated breakpoint condition in the *Breakpoint Properties* dialog of the conditional breakpoint of interest. After running BPGen the developer can begin to debug the program with these breakpoints in hand, which may save a lot of debugging efforts.

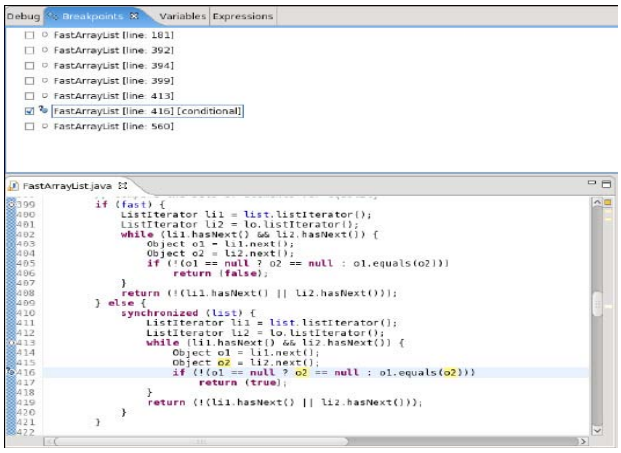

Figure 3: Generated breakpoints in Breakpoints view and Java editor

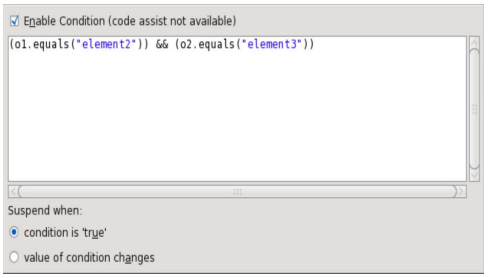

Figure 4: Generated breakpoint condition

3. RELATED WORK

Breakpoints are acknowledged as a powerful facility in program debugging. Chern and Volder [5] design Control-flow Breakpoint Debugger(CBD) which uses a pointcut language to express control-flow conditions for breakpoints and a menu-based GUI to help users specify the conditions. Being largely inspired by CBD, BPGen is orthogonal to CBD in that it generates conditions in terms of program states while CBD provides candidate conditions in terms of control-flow.

Zeller et al. develop the Delta Debugging techniques to systematically analyze the differences in program inputs, source codes, and program states to isolate failure-inducing inputs, extract cause-effect chains, and link cause transitions to program failures [19, 18, 6]. Compared with the Delta Debugging techniques which are fully automated, BPGen

is primarily an aid to interactive debugging. Moreover, we put emphasis on finding the locations to set breakpoints and the opportunities to extract and compare memory graphs, while Delta Debugging generally performs the searches in space and time in a divide-and-conquer style. In BPGen, we adopt the memory graph comparison from [20] and its DDstate implementation. Although the DDstate implementation systematically changes the memory graph to isolate relevant states, we just use it to compare the graphs and analyze the graph differences to find accessible suspicious states for generating breakpoint conditions.

Renieris and Reiss develop the nearest neighbor queries method [14]. Based on the nearest neighbor queries method, BPGen focuses on improving its usability in practice. Hao et al. introduce a breakpoint-based interactive debugging aid [7], which is similar to BPGen. However, BPGen combines various fault localization techniques to automatically generate breakpoints instead of using visualization to recommend breakpoint locations. Moreover, BPGen generates conditions for breakpoints to strengthen their usability.

Program slicing is a powerful assistance to program debugging [16]. While slices are static positions in programs, we append dynamic information by attaching conditions to the generated breakpoints. By combining dynamic and static program slicing, Ko and Myers develop a tool called Whyline[10] for interrogative debugging. Using Whyline a developer can easily query both dynamic and static dependencies in program elements and execution events. Compared with Whyline, BPGen is more of an enhancement to the breakpoint utility than a reinvented debugging approach, and the breakpoints generated by BPGen may be more familiar and acceptable to the developers. By recording every state change at runtime, omniscient debugger [12] enables the developers to debug backwards in time. Similar to Whyline, omniscient debugger greatly enhances the developers' ability to query runtime events in either direction along the time line. In contrast, BPGen aims to indicate which variables should be inspected at which statements.

4. CONCLUSION AND FUTURE WORK

In this paper, we have presented BPGen, a tool which is used to automatically generate breakpoints. BPGen adopts three well-known dynamic fault localization techniques, including nearest neighbor queries method, dynamic program slicing, and memory graph comparison, to generate both conditional and unconditional breakpoints. In the future, we will adopt more powerful strategies (e.g., using the cause-effect relations [18]) to generate breakpoints more effectively. We also plan to improve the usability of BPGen.

5. ACKNOWLEDGMENTS

We are grateful to Hao Xu and Fangyue Wang for programming assistance and to Karsten Lehmanm for his instructions on the usage of DDstate plugin. This work was supported in part by National High Technology Development Program of China (Grant No. 2006AA01Z158), National Natural Science Foundation of China (NSFC) (Grants No.60673120 and No. 60970009).

6. REFERENCES

[1] DDstate: Failure-Inducing States. http://www.st.cs.uni-saarland.de/eclipse/.

[2] Eclipse project. http://www.eclipse.org/.

[3] Results from NetBeans Debugger Survey. http://debugger.netbeans.org/survey.html.

[4] Soot: a java optimization framework. http://www.sable.mcgill.ca/soot/.

[5] R. Chern and K. De Volder. Debugging with control-flow breakpoints. In *Proceedings of the 6th international conference on Aspect-oriented software development*, pages 96–106, 2007.

[6] H. Cleve and A. Zeller. Locating causes of program failures. *27th International Conference on Software Engineering*, pages 342–351, 2005.

[7] D. Hao, L. Zhang, L. Zhang, J. Sun, and H. Mei. Vida: Visual interactive debugging. *31st International Conference on Software Engineering*, pages 583–586, 2009.

[8] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. *International Symposium on Software Testing and Analysis*, pages 167–178, 2008.

[9] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. *20th IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282, 2005.

[10] A. J. Ko and B. A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. *30th International Conference on Software Engineering*, pages 301–310, 2008.

[11] B. Korel and J. W. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[12] B. Lewis. Debugging backwards in time. *International Workshop on Automated Debugging*, pages 225–235, 2003.

[13] G. C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the eclipse ide? *IEEE Software*, 23(4):76–83, 2006.

[14] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. *23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 30–39, 2003.

[15] A. Seesing and P. Orso. InsECTJ : a generic instrumentation framework for collecting dynamic information within eclipse. *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 45–49, 2005.

[16] F. Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

[17] T. Wang and A. Roychoudhury. Using compressed bytecode traces for slicing java programs. *26th International Conference on Software Engineering*, pages 512–521, 2004.

[18] A. Zeller. Isolating cause-effect chains from computer programs. *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 1–10, 2002.

[19] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.

[20] T. Zimmermann and A. Zeller. Visualizing memory graphs. In *Software Visualization*, pages 191–204, 2001.