Automated Breakpoint Generation for Debugging

Cheng Zhang[1,3], Juyuan Yang[2], Dacong Yan[4], Shengqian Yang[4], Yuting Chen[2]
[1]Department of Computer Science & Engineering and [2]School of Software, Shanghai Jiao Tong University
[3]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[4]Computer Science and Engineering Department, Ohio State University
Email: cheng.zhang.stap@sjtu.edu.cn, juyuanyang@gmail.com
{yan, yangs}@cse.ohio-state.edu, chenyt@cs.sjtu.edu.cn

***Abstract*— In debugging processes breakpoints are frequently used to inspect and understand the run-time program behavior. Although modern development environments provide convenient breakpoint utilities, it mostly requires considerable human effort to create useful breakpoints. Before setting a breakpoint or typing a breakpoint condition, developers usually have to make some judgements and hypotheses based on their observations and experiences. To reduce such kind of effort, we propose an approach to automatically generating breakpoints for debugging. We combine the nearest neighbor queries method, dynamic program slicing, and memory graph comparison to identify suspicious program statements and states. Based on this information, breakpoints are generated and divided into two groups, where the primary group contains conditional breakpoints and the secondary group contains unconditional ones. We implement the presented approach on top of the Eclipse JDT platform. Our objective experiments and user study indicate that the generated breakpoints can be effective in aiding debugging work.**

***Index Terms*— software debugging, breakpoint generation**

I. Introduction

Debugging has long been a laborious task for software developers [1], [2]. Despite the existence of various automated debugging techniques, the practical debugging activities are mainly manual and interactive. It is difficult to mechanically deal with diversified bugs, some of which may be extremely tricky. Thus the fundamental tool in debugging is the human brain; the common debugging process consists of several iterations of observing, making hypotheses, and testing the hypotheses [3].

Most modern development environments provide comprehensive utilities to aid debugging work. Among these utilities, breakpoint is one of the most frequently used [4]. By using breakpoints developers can suspend the program execution and explore the program state to find clues to bugs. During the debugging process, however, developers usually have to ask themselves a question: *where to set the breakpoints?* Setting breakpoints involves their observations, judgements and hypotheses on the current task and even experiences of similar situations. It is common that a veteran sets a small number of breakpoints to reveal a bug immediately, whereas a novice sets excessive useless breakpoints, but still fails to find the bug. Therefore, for a developer, a set of well defined breakpoints presented in a familiar debugger may be a quite pleasant starting point for the subsequent debugging work. Some modern IDEs, such as Eclipse[1], provide the functionality to export and import breakpoints, facilitating reuse and sharing of them. Additionally, we believe that breakpoints can also be automatically generated to further accelerate debugging processes.

To reduce the manual effort of debugging, researchers have proposed a variety of automated debugging techniques. Some techniques [5]–[8] focus on the abnormity of execution traces by collecting and analyzing the information of executions of the passing and failing test cases. Various measurements are taken to indicate the statements that are more relevant to failing runs and identify the most relevant statements as the places where the faults reside. Another group of techniques put emphasis on the abnormal run-time program states [9]–[11]. They observe the selected program states at certain locations and identify the most suspicious states which are much more likely to appear in failing runs. Ko and Myers [12] view the debugging process as a question and answer session and develop the innovative Whyline tool to prepare candidate questions and help to find the answers by using both static and dynamic analysis techniques. Nevertheless, developers may still encounter problems when using these techniques for real world debugging tasks. For example, they may want to try different techniques to find the fault more effectively. Then the problem is how they could use the outcomes of different techniques. They may have to familiarize themselves with various kinds of reports in the form of tables of suspicious states, lists of faulty statements or some novel visualizations. Moreover, when the reports fail to indicate the faulty statements clearly, which is often the case, developers probably have to switch to their normal methods and make further effort to find the bugs. In this case, they may need a seamless integration of the innovative techniques and the practical productivity tools. Thus it is helpful to improve the usability and the way of combining automated debugging techniques to provide continuous support for debugging.

In this paper, we propose a method to generate breakpoints based on a synergy of existing fault localization techniques. With regard to the typical situation of debug-

[1]http://www.eclipse.org/

doi:10.4304/jsw.8.3.603-616

ging, where there are much fewer failing test cases than passing ones, we first use the nearest neighbor queries method [6] to select a passing case that is the most similar to the failing case (in terms of execution flow) and compare them to find the suspicious basic blocks. Then we use backward dynamic slicing [13] to augment the report of the nearest neighbor queries in order to reveal the causes of the differences between the two execution flows. Line breakpoints are generated at the conditional statements contained in the dynamic slices. At last we extract memory graphs at specific breakpoints, compare the graphs to identify the most interesting differences of program states, and generate breakpoint conditions based on the state differences. The goal of our work is to automatically generate useful breakpoints to aid real world debugging activities. We have performed a group of objective experiments to show the effectiveness of our technique in fault localization. In addition, we have performed a user study to demonstrate the usefulness of our approach.

An earlier version of the proposed approach has been implemented and publicly demonstrated [14]. By comparison, this paper improves the step of breakpoint condition generation, making it more efficient and portable. Different from the tool demonstration paper [14], this paper explains the key ideas with necessary background (Section II) and a running example (Section III). Also, this paper gives more detailed explanations on the considerations, strategies, algorithms, and trade-offs in our approach (Section IV). In addition, a comprehensive evaluation has been performed and the results are shown and discussed extensively (Section VI).

In summary, the main contributions of our work are:

- We propose the idea of automated breakpoint generation by leveraging the power of fault localization techniques. This approach can be a viable way of utilizing academic productions in practical debugging environments.
- We implement a prototype of our approach on top of the Eclipse JDT platform and use it to perform objective experiments and case studies. The result is promising.

II. Background

A. Nearest Neighbor Queries

In [6], Renieris and Reiss adopt the k-NN algorithm [15] to select from a set of passing runs the one that most resembles the failing run (in terms of execution trace), compare it with the failing run, and report suspicious basic blocks from the differences. The distance between a pair of runs (one passing and one failing) is defined in terms of common distance metrics, such as Hamming distance, Ulam distance, etc. Before computing the distance, execution traces are converted to either binary coverage spectra or exact execution counts. In binary coverage spectra, the execution count of a basic block is mapped to 1 or 0, indicating whether the basic block has been executed or not; in exact execution count spectra, how many times each basic block has been executed matters. Take the asymmetric Hamming distance (AHD) and binary coverage spectrum as an example. Binary coverage spectra are usually represented as binary vectors. Thus the Hamming distance between two runs f (failing run) and p (passing run) is defined as the number of positions at which the two representing binary vectors disagree. The asymmetric Hamming distance, which considers the non-symmetric difference $f - p$, is used to report features that are only present in the failing run f. A sample calculation is given in Table I.

TABLE I.
AHD BETWEEN TWO RUNS, USING BINARY COVERAGE SPECTRA

sample	Block 1	Block 2	Block 3
Failing Run f	3	0	2
Passing Run p	0	1	3
Block Difference	1	0	0

$Distance(f, p) = (1 + 0 + 0)/3 = 1/3$

When all the pairwise distances have been computed, the pair with the minimum distance value is selected and compared. Finally, those basic blocks executed only in the failing run will be reported as suspicious.

B. Dynamic Program Slicing

Program slicing was first introduced by Weiser [16] to extract parts of program code corresponding to program behaviors of interest. Since then it has been widely applied to program testing, debugging, understanding and other maintenance activities. During program debugging it can be used to reduce the amount of code developers need to inspect. In program slicing, a program is sliced with respect to a slicing criterion, $\langle p, V \rangle$, where p is a specified program point and V is a set of variables of interest. The conventional program slicing is static, because it is performed based on static program dependencies and finds all the statements that might affect the slicing criterion. Although being helpful to debugging, static program slicing often produces too large slices to reduce the developers' effort. In order to narrow down the search space further, Korel and Laski [13] proposed dynamic program slicing which is based on dynamic program dependencies and finds the statements that have actually affected the slicing criterion in a specific execution. The backward dynamic slice of a variable at a point in the execution trace includes all the executed statements which have directly or indirectly affected the value of the variable at that point. In [17], Zhang et al. compare three dynamic backward slicing methods: data slicing, full slicing and relevant slicing. As demonstrated in their study, a full slice can reveal dynamic data or control dependence chains of specified statements. Therefore, it would be helpful to perform a full backward dynamic slicing in order to locate the root cause of a failure accurately.

C. Memory Graph

The concept of memory graph is first proposed by Zimmermann and Zeller [18] to capture and explore

program states. Conceptually a memory graph represents a program's state at a specific moment during its execution in the form of a directed graph whose vertices contain information of variable values and edges represent the relations between variables. In its formal definition in [18], a memory graph G is defined as a triple $(V, E, root)$, where V is a set of vertices, E is a set of edges, and $root$ is a dedicated vertex. Each vertex v in V is a triple $(val, tp, addr)$ which stands for the value, type, and memory address of a variable. Each edge e in E is a triple (v_1, v_2, op) where v_1 and v_2 are the source and target nodes of e, respectively, and op constructs v_2's name from v_1's name indicating the way of accessing v_2 from v_1. The variables that are not accessed by any other variables and can be directly accessed from the current context are called *base variables*. The dedicated vertex $root$ is designed to reference all the base variables so that each v in G is accessible from $root$. The $root$ vertex itself contains no variable information.

As a typical application for memory graph in software debugging, memory graph comparison finds the common subgraph of two memory graphs and pinpoints the differences between them. Details of the comparison algorithms can be found in [18]. In our approach, we adapt the comparison method to generate breakpoint conditions.

```
1  public int method_to_test(int a){
2    int b = 5;
3    if(a < b)
4      a = methodA(a);
5    b += a;
6    return b;
7  }
8  public int methodA(int c){
9    int d = c + 1;
10   c += 1; // this statement should be c+=2
11   if(c < 4)
12     return c;
13   return d;
14 }
```

Figure 1. An example Java program. The faulty statement is at line 10, where the correct version should be `c+=2`.

III. EXAMPLE

In this section, we illustrate our approach by applying it to a simple contrived Java program shown in Figure 1. In the program, the correct version of the statement at line 10 should be `c += 2`. Suppose we have three test cases $t1$, $t2$, and $t3$, whose inputs and expected outputs, described in the form of *(input, expected output)*, are $(1, 8)$, $(4, 10)$, and $(7, 12)$, respectively. If we run the test cases, $t1$ fails, while $t2$ and $t3$ pass, because the actual output of $t1$ is 7 (instead of 8). Then the debugging task is to find out why $t1$ fails and fix the program to make it pass.

Our approach is applied just before the manual debugging session begins. Figure 2 illustrates an overview of the process of the approach. In the first step, we instrument the program and run the test cases, $t1$, $t2$, and $t3$, to collect their execution traces. Table II shows the test results and executed basic blocks, in the form of *(begin line, end line)*, for the three test cases. Then we calculate the distances between failed and passed test cases[2], and get $Distance(t1, t2) = 1/6$ and $Distance(t1, t3) = 3/5$. Since $Distance(t1, t2)$ is less than $Distance(t1, t3)$, we identify $t2$ as the test case nearest to $t1$ and report the difference in executed basic blocks between $t1$ and $t2$ as suspicious. In this example, the only statement contained in the suspicious basic block is at line 12.

TABLE II.
TEST RESULTS AND EXECUTED BASIC BLOCKS OF THE EXAMPLE

test case	test result	executed basic blocks, (begin line, end line)
$t1$	fail	(2, 3), (4, 4), (5, 6), (9, 11), (12, 12)
$t2$	pass	(2, 3), (4, 4), (5, 6), (9, 11), (13, 13)
$t3$	pass	(2, 3), (5, 6)

In the second step, we perform backward dynamic slicing using the suspicious statement and the relevant variable c as the slicing criterion, $\langle line12, c\rangle$. The resultant slice contains the statements at lines 2, 3, 4, 10, 11, and 12. In the third phase of our approach, we take the strategy that generates breakpoints at conditional statements. Thus, two breakpoints (noted as $bp3$ and $bp11$ hereafter) are generated at lines 3 and 11 , respectively.

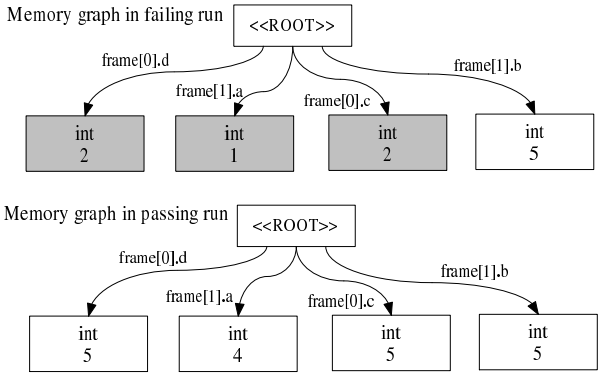

Figure 3. Memory graphs extracted at $bp11$

In the last step, we re-run the test cases $t1$ and $t2$, extract a memory graph during each run when $bp11$ is hit, and compare the two memory graphs to identify the program states which only appear in the failing memory graph. We regard these states as suspicious program states and generate the breakpoint condition for $bp11$ based on them. The memory graphs in this example are illustrated in Figure 3 where the suspicious program states are colored grey and irrelevant states, such as the *this* reference, are not displayed. As a result, the breakpoint condition of $bp11$ is set to `(c == 2)&&(d == 2)`. Here we choose to perform memory graph comparison at $bp11$ rather than $bp3$, because the run-time values of the branch predicate `c < 4` (at line 11) are different between the failing and passing runs, while those of the predicate `a < b` (at line 3) are both `true`. Moreover, we focus on

[2]Here we use AHD to perform nearest neighbor queries.

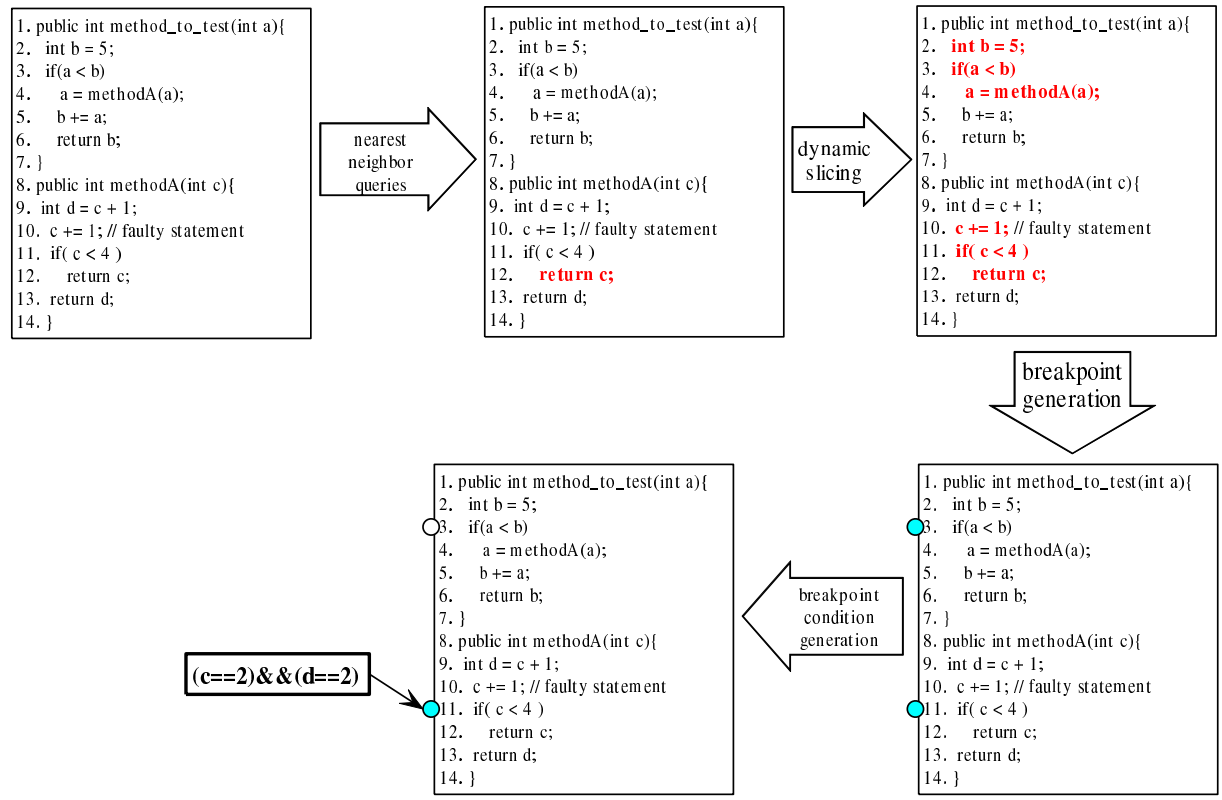

Figure 2. Overview of the automated breakpoint generation process

the variables accessible from the current context, whose names are prefixed with `frame[0]`, when generating breakpoint conditions. The rules of choosing breakpoints for memory graph comparison and generating breakpoint conditions will be explained in Section IV-B.

So far, two breakpoints, $bp11$ and $bp3$, have been produced. At last we classify the conditional breakpoints, such as $bp11$, as *primary breakpoints*, and the other unconditional ones, such as $bp3$ as *secondary breakpoints*. When being presented to users, primary breakpoints are enabled and secondary breakpoints are disabled (but not removed) by default. During the subsequent debugging work, if we run $t1$ (in debug mode), the execution will be suspended at line 11 and the breakpoint condition `(c == 2)&&(d == 2)` will indicate the suspicious program states. If we enable $bp3$ and re-run $t1$, we can find out that the value of variable a is 1 at that time. Since in the correct program the state `(a == 1)` implies `(c == 3)` at lint 11, the contrast between `(c == 2)` and `(c == 3)` strongly indicates the faulty statement at line 10 that has been executed between lines 3 and 11.

IV. Approach

The process of our approach consists of two phases: 1) selecting breakpoint locations, which includes the nearest neighbor queries and the dynamic program slicing, and 2) generating breakpoint conditions, which is mainly based on the memory graph comparison.

A. Selecting Breakpoint Locations

There are various kinds of breakpoints, such as line breakpoints, exception breakpoints, method entry breakpoints and watch points. According to our own experience, line breakpoints are the most frequently used. Thus, in the presented work, all the generated breakpoints are line breakpoints. For a line breakpoint, the primary property is its location, that is, at which line of code it is set.

Regarding the typical debugging scenario, where there is one failing test case (or developers usually deal with failing test cases one by one) and several passing test cases, we first adopt the strategy that generates breakpoints at a certain type of statements. All these statements are conditional statements where the execution flow of the failing test case diverges from that of its *nearest* passing test case. The nearest passing test case is the one that has an execution flow most resembling that of the failing test case. We focus on the nearest passing test case, because, as shown in [6], [7], 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 faults. On the contrary, if we compare two runs that are fundamentally different, it may be difficult to find useful information from the comparison.

We use the nearest neighbor queries method from [6] to find the nearest passing test case for a given failing one and identify the differences in execution traces. Compared with other fault localization techniques, the nearest

neighbor queries approach has an acceptable accuracy of 50% and less technical complexity. As described in Section II, it just requires execution traces of the failing and passing test cases and suits well to the typical debugging scenario which we focus on. We collect execution traces by instrumenting the programs and then running the test cases. In the nearest neighbor queries, we use AHD to compute the distance between execution traces and determine the nearest passing test case. Finally, the differences are recorded in the form of basic blocks. The conditional statements anterior to these basic blocks are proper locations to set breakpoints, because they may be close to the faulty statements and contain information to explain the divergence of execution flow. The Java program shown in Figure 4 presents a common case where the faulty statement will be executed only if `c < 3` is evaluated to `true`. Thus it may be quite helpful to generate a breakpoint at the `if` statement at line 8 to suspend the execution just before the faulty statement and show the value of `c` at that time.

```
1  public int method_to_test(int a){
2    int b = 5;
3    if(a < b) a = methodA(a);
4    b += a;
5    return b;
6  }
7  public int methodA(int c){
8    if(c < 3)
9      c += 3; // faulty statement
10   return c;
11 }
```

Figure 4. Another example Java program

However, it is often inadequate to generate breakpoints only at such conditional statements. There are several cases where control-flow based techniques, like the nearest neighbor queries method, do not perform well. For example, in Figure 1, the control-flow difference is found at line 12, but it is just the late symptom of executing the faulty statement at line 10. Although the developer may discover the fault by manually analyzing data dependence and searching backward, the task could be laborious if the dependence and program structure are complex. To address this issue, we use backward dynamic slicing to trace back to the program points where the variable values initially come into the incorrect program states. We add all the statements reported by the nearest neighbor queries method and the variables referenced by them to the slicing criterion set, and re-run the failing test case to perform a backward dynamic slicing. Breakpoints are then generated at the conditional statements contained in the resultant slice.

B. 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 on the breakpoint hit if the condition is evaluated to `true`. Thus a breakpoint holding a proper condition can suspend the execution 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 inspecting.

In our approach, we adapt the memory graph comparison to synthesize breakpoint conditions for some of the generated breakpoints. Since memory graphs can represent the run-time program states exhaustively in a structural way, the comparison results are usually precise for presenting incorrect states and also convenient for condition generation. In the example described in Section III, the breakpoint condition `(c == 2)&&(d == 2)` is directly composed based on the grey vertices and their incoming edges. However, extracting a memory graph may incur heavy performance overhead when the program states are large in number and have complex data structures. Therefore, it is prohibitively expensive to extract a memory graph at **every** breakpoint when it is hit. We believe that it is also unnecessary, since a large number of duplicate meaningless breakpoint conditions may be generated. Regarding that the breakpoint locations are selected based on the differences in execution flow, we propose the following rule for choosing the breakpoints and opportunities (i.e., the hit counts of breakpoints) to perform memory graph comparisons.

Given a set of breakpoints $BP = \{bp_1, bp_2, ..., bp_N\}$ and a specific breakpoint $bp_i \in BP$, we collect the runtime values of the corresponding condition expression during the executions of the failing test case and the nearest passing test case. Since every breakpoint is located at a conditional statement which may be executed for several times during a single execution, there can be one or more runtime values for each condition expression. 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. In essence, 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 $cp_{i,j} \in CP_i$, we extract two memory graphs $G^f_{i,j}$ and $G^p_{i,j}$ when the breakpoint bp_i is hit for the jth time during the failing and passing runs, respectively. Then we compare the two graphs using the algorithm borrowed from [18] to find out the almost largest common subgraph of them, the edges only in the failing graph, and the edges only in the passing graph. Since there can be up to two comparison points for each breakpoint, at most two sets of graph differences may be produced for the breakpoint during the graph comparison. No conditions will be generated for the breakpoints having no corresponding graph differences. If a breakpoint has exactly one set of graph differences, this set will be used for generating the breakpoint condition. When there are two sets of differences for a breakpoint, we follow the rules below to select one of them for breakpoint condition generation:

- When n_c and n'_c are both zero, we choose d if $n_{ef} + n_{ep} \leq n'_{ef} + n'_{ep}$, otherwise we choose d'.
- When n_c is non-zero and n'_c is zero, we choose d.
- When n_c is zero and n'_c is non-zero, we choose d'.
- When n_c and n'_c are both non-zero, we choose d if $\frac{n_{ef}+n_{ep}}{n_c} \leq \frac{n'_{ef}+n'_{ep}}{n'_c}$, otherwise we choose d'.

Here d, n_c, n_{ef}, and n_{ep} (d', n'_c, n'_{ef}, and n'_{ep}) stand for the set of edges only in the failing graph, the number of edges in the common subgraph, the number of edges only in the failing graph, and the number of edges in only the passing graph for the first comparison (the second comparison). We propose these rules based on the idea that *if two memory graphs are similar to each other, the differences between them may be preferable for revealing incorrect program states*. Based on the set of memory graph differences comprising a set of graph edges, we compose the breakpoint condition using the algorithm shown in Figure 5.

The function `ConditionGeneration` takes the set of edges which only exist in the failing memory graph as its input and outputs a breakpoint condition in the form of a character string which is a syntactically correct boolean expression. During condition generation we focus on the variables accessible from the top stack frame (checked by the function `isAccessible`), because only these variables are visible and can be used to evaluate the breakpoint condition when a breakpoint is hit. The function `connect` composes the string output by concatenating all the subexpressions contained in `SubExprs` with the logical conjunction operator (i.e., &&).

At last each breakpoint is classified as *primary* or *secondary* according to whether it is conditional or not. Primary breakpoints will be enabled by default. Thus, when they are hit during an execution, their conditions will always be evaluated and the execution may be suspended accordingly. On the other hand, secondary breakpoints will be disabled by default, that is, they will not come into effect until users enable them. We divide the breakpoints into two groups in order to 1) distinguish the breakpoints that are more likely to reveal suspicious program states and 2) reduce the number of breakpoints that users have to inspect in the first iteration of debugging.

```
1: function CONDITIONGENERATION(Edges)
2:     SubExprs ← {}
3:     for all e_i ∈ Edges do
4:         node_i ← getTargetNode(e_i)
5:         op_i ← getOperation(e_i)
6:         if isAccessible(op_i) then
7:             VarName ← getVarName(op_i)
8:             if isArrayNode(node_i) then
9:                 SubExprs ← SubExprs ∪
10:                    {VarName.length == node_i.getLen()}
11:            end if
12:            if isNullNode(node_i) then
13:                SubExprs ← SubExprs ∪
14:                        {VarName == null}
15:            end if
16:            if isPrimitiveNode(node_i) then
17:                SubExprs ← SubExprs ∪
18:                    {VarName == node_i.getValue()}
19:            end if
20:            if isStringNode(node_i) then
21:                SubExprs ← SubExprs ∪
22:                    {VarName.equals(node_i.getStrValue())}
23:            end if
24:        end if
25:    end for
26:    return connect(SubExprs)
27: end function
```

Figure 5. Algorithm for breakpoint condition generation

V. IMPLEMENTATION

We have implemented our approach as an Eclipse plug-in, named *BPGen*. As shown in Figure 6, BPGen is composed of various components which can be divided, by functionality, into two subsystems: the fault localization subsystem and the breakpoint generation subsystem.

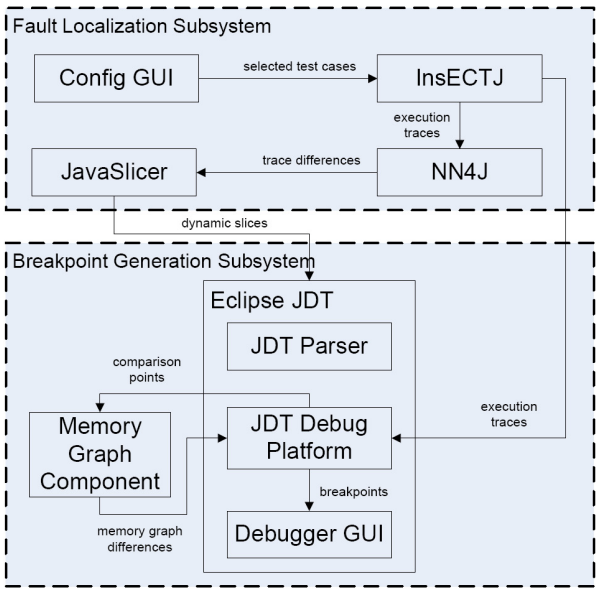

Figure 6. Architecture of the implementation

A. Fault Localization Subsystem

We combine several techniques in a pipe-and-filter architecture to implement our fault localization subsystem. The input of the subsystem is ordinary Java bytecode, which will be instrumented at runtime by InsECTJ [19]. InsECTJ is a generic instrumentation framework for collecting dynamic information for Java programs. While running the instrumented program, we record the number of times each basic block is executed as traces. Then we use NN4J to perform nearest neighbor queries on the collected traces to select a passing test case that most resembles the failing one, and report the differences between their traces. NN4J is our implementation of the nearest neighbor queries method, whose major algorithms are borrowed from [6]. After finding out the differences in execution traces, we re-run the failing case, using JavaSlicer [20] to compute a backward dynamic slice with the statements reported by NN4J and the variables referenced by these statements as slicing criteria. The calculated slice is the final output of the fault localization subsystem.

B. Breakpoint Generation Subsystem

We build the breakpoint generation subsystem on top of Eclipse JDT. One major function of this subsystem is to generate breakpoints at branch statements contained in the slice computed by the fault localization subsystem. We implement this function using the Java parser and the debug platform provided in Eclipse JDT. The parser parses a Java source file into its AST representation, with which we can easily point out the branch statements in the source file. The breakpoint framework in the Eclipse JDT debug platform provides a group of utilities to manipulate breakpoints. We directly use the framework to generate breakpoints. The other major function of the subsystem is to identify comparison points, perform memory graph comparisons and generate breakpoint conditions. We compute the comparison points by analyzing the execution traces collected by InsECTJ. In previous version, we reuse the memory graph component of the DDstate plugin[3] to implement a breakpoint listener. Once the listener has been registered to the debug model, memory graphs will be extracted and compared when related comparison points are reached during a debugging session. In latest version, we use Java Debug Interface to implement our debugger and a simple memory graph. When the debugger meets a breakpoint, it finds all the accessible variables and compares them. At last we implement the condition generation algorithm directly and leverage the breakpoint framework to set the conditions and default status (i.e., enabled or disabled) of breakpoints.

VI. Evaluation

We have performed an objective experiment and a user study to evaluate our approach. This section describes the designs, results, and observations.

[3]http://www.st.cs.uni-saarland.de/eclipse/

A. Evaluation of Fault Localization

It is reasonable that a breakpoint close to a faulty statement will be more helpful to discover the fault than another breakpoint far from the faulty statement. Moreover, we believe that developers may get meaningful information or even discover the faults directly, if they scrutinize the automatically generated breakpoints before running test cases. In this case, the breakpoints can be viewed as a group of suspicious statements reported by fault localization techniques like the nearest neighbor queries method. Moreover, we must consider the number of breakpoints generated. Because it makes no sense to indicate the faulty statements by generating a breakpoint at every line of code. Furthermore, when observing an execution of a test case, developers may be seriously annoyed by frequent interruptions caused by too many breakpoints. Therefore, we carried out an experiment to check whether the breakpoints are generated at proper locations and whether there are too many breakpoints generated.

Distance to Fault. For a statement in a program, we use its *distance to fault* to indicate how far it is from the faulty statements. We make use of the concept of program dependence graph (PDG) [21] to compute the distance to fault, which is inspired by the evaluation framework proposed in [6]. A program dependence graph consists of nodes and directed edges, where a node represents a statement in the program and an edge connecting two nodes represents the control dependence or data dependence between them. Suppose, in a PDG, the faulty statement is represented by node $v1$ and another statement is represented by node $v2$, the distance to fault of $v2$ is the number of edges included in the shortest path connecting $v1$ and $v2$ in the PDG. The direction of the path is either from $v1$ to $v2$ or from $v2$ to $v1$. The distance to fault of a breakpoint bp is defined as that of the statement v_{bp} where bp is located. The distance to fault of a set of statements (or breakpoints) is defined as the smallest distance to fault of each statement (or breakpoint).

We assume that developers would try to find the fault by reading the source code of a program, starting from a generated breakpoint and searching along the edges on the PDG. We also assume that developers are able to recognize the fault immediately when they see the faulty statements. Moreover, they may pay much more attention on the two ends of each edge than other statements. With these assumptions, it is reasonable to use the distance to fault of a set of breakpoints to represent the possible least effort made by the developers, when they scrutinize the generated breakpoints without running any test case.

Experiment Setup. We used two Java programs from Software-artifact Infrastructure Repository (SIR) and the Apache common collections version 3.2.1 as the subjects which are described in Table III. We have selected some versions of the SIR programs, because they had usable junit tests and we could successfully compile and execute these programs and their tests in our experimental envi-

ronment. Since the faults in SIR programs are seeded, we have selected the Apache program to study real bugs. For SIR programs, each faulty version has exactly one fault seeded, as we did not focus on localizing multiple faults in this experiment. For the same reason, each of the selected bugs in the Apache program has a single faulty statement. A fault (in SIR programs) may be seeded by commenting out a necessary statement. In this case, we preserve the node and edges corresponding to the statement in the PDG and mark them as virtual. During the computation of distance to fault, a virtual node can be only used as the starting point or end point of a path.

From Table III, we can see that there are generally more than one failed test case for a fault. Since we do not know a priori which failed test case is the most suitable for performing nearest neighbor queries, we try every one of them in turn to find out the best one in the experiments. However, in practical debugging, the developer may easily select the failed test case which interests her most or there is just one failed test case assigned for her debugging work.

TABLE III.
SUBJECT PROGRAMS. LOC AND #TESTS ARE AVERAGES PER FAULT.

Program	Versions	Avg. LOC	Avg. #Tests (failed/all)
jtopas	v1, v2	7347	1.8/127
nanoxml	v1, v3, v5	6395	18.4/142
collections	v3.2.1	63852	2.0/2374

In the experiment, we make a comparison between the distances to fault of the breakpoint sets and that of the nearest neighbor reports. Furthermore, we studied the size of the generated breakpoint sets, with comparisons between the nearest neighbor reports, the dynamic slices, and the execution traces.

Experimental Results. Figure 7 shows the average distance to fault of the generated breakpoint sets and the nearest neighbor reports. For 19 of the 30 faults, the distances to fault of the generated breakpoints and the nearest neighbor reports are equal or have merely minor differences (less than 3). This is reasonable, because the breakpoints are generated mainly based on the nearest neighbor reports. Thus, when a report contains a faulty statement, some breakpoints may probably be generated very close to the statement. The minor differences are caused by our current strategy which only generates breakpoints at conditional branch statements. The consequence is that a breakpoint is usually located at the conditional branch statement nearest to the faulty statement, rather than the faulty statement itself. However, since most of the breakpoints have directly data and/or control dependencies with the faulty statements, the differences might be insignificant.

The generated breakpoints have significantly smaller distances to fault in 11 out of 30 cases. By analyzing source code and data, we found that the faulty statements in these cases are usually executed early in the executions of both failed and passed test cases. The difference is that the failed test cases made assertions

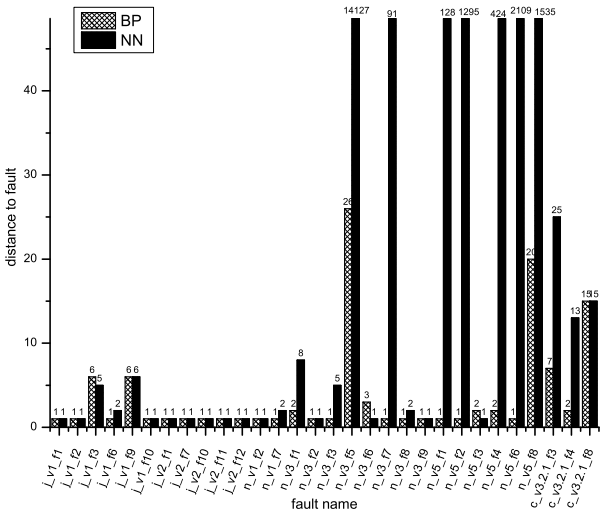

Figure 7. Distance to fault of generated breakpoints and nearest neighbor.

to check some faulty program states and failed, while the passed test cases only checked some correct program states and passed. Moreover, some checked states were values get from maps with some keys, like `value = getAttribute(key)`. These values are often contained in aggregative objects and retrieved with the keys whose values may be determined only at runtime. Consequently, the static dependencies between these assertions and the faulty statements are usually complex. In this case, dynamic program slicing can largely simplify the dependence analysis by using the run-time information. In the experiments, the nearest neighbor reports contained merely the method calls like `getAttribute(key)` and the backward dynamic program slicing accurately discovered the faulty statements. Consequently, breakpoints can be generated very close to the statements.

Table IV shows the size of the generated breakpoint sets, the nearest neighbor reports, and the dynamic slices. The first two lines, labeled 'Program' and 'Ver.', display the program names and versions, respectively. The line labeled '#Faults' shows the number of seeded faults in each version of a program. The lines 'Avg. BP', 'Avg. Con. BP', 'Avg. NN', and 'Avg. DS' present the average number of generated breakpoints, generated conditional breakpoints, statements in reports of nearest neighbor queries, and statements in dynamic slices, respectively. The line 'Avg. exec.' shows the average number of executed statements. The lines 'BP/exec.', 'NN/exec.', and 'DS/exec.' present the ratio of 'Avg. BP' to 'Avg. exec.', 'Avg. NN' to 'Avg. exec.', 'Avg. DS' to 'Avg. exec.', respectively.

By comparing the columns 'Avg. BP' and 'Avg. NN', we can see that the breakpoint sets are smaller than the nearest neighbor reports in most cases. It is the consequence of the strategy to generate breakpoints only at certain types of statements, although the reports are enlarged with dynamic slices. The data in lines 'BP/exec.' and 'NN/exec.' indicate that both the breakpoint sets and

TABLE IV.
SIZE OF BREAKPOINT SETS, NEAREST NEIGHBOR REPORTS, AND SLICES.

Program	jtopas		nanoxml			collections
Version	v1	v2	v1	v3	v5	v3.2.1
#Faults	6	5	2	8	6	3
Avg. BP	11	42	45	61	49	4
Avg. Con. BP	1	1	3	2	1	0
Avg. NN	24	49	64	52	20	87
Avg. DS	60	115	193	263	209	11
Avg. exec.	22421	2907	7077	14222	12233	539
BP/exec.	0.05%	1.43%	0.61%	0.43%	0.40%	0.68%
NN/exec.	0.11%	1.69%	0.90%	0.36%	0.16%	16.19%
DS/exec.	0.27%	3.97%	2.73%	1.85%	1.71%	2.10%

the nearest neighbor reports are relatively small compared with the dynamic slices or the executed statement. Thus, it may be acceptable for developers to scrutinize the breakpoint sets to locate the faults without running test cases. However, breakpoints are supposed to suspend the execution at runtime, so only the conditional ones are enabled by default.

From the lines 'Avg. DS' and 'DS/exec.', we can see that the dynamic slices are much smaller than the full slices and the relevant slices reported by Zhang et al. [17]. The discrepancy is due to the fact that we only count the statements contained in the code of the subjects, not in the libraries. We count the statements in the reports of nearest neighbor queries in the same way and breakpoints are also generated only in the code of the subjects.

Table V shows the runtime of our approach. Each column shows the average result of different versions for each subject program. Each line, except for the one labeled 'Avg. NN4J', presents one component of our approach described in Section V. 'Computing CP' is the step to generate breakpoints and identify comparison points, and 'Condition Generation' is the step to generate conditions for breakpoints. As each fault may lead to different number of failed test cases, and we perform nearest neighbor queries on each failed test case one by one with all passed ones, so the runtime data mainly depends on the number of failed test cases. The line labeled 'Avg. NN4J' displays such an average runtime for each failed test case in component NN4J. Therefore, the line 'Total' shows the runtime taken in our experiments, whereas, in the runtime of a real debugging case, the value of 'NN4J' should be replaced with 'Avg. NN4J', since the developer will probably choose the most interesting failed test case at the beginning of the debugging task.

TABLE V.
RUNTIME OF EACH STEP OF BREAKPOINT GENERATION.

	Runtime (sec.)		
Step	jtopas	nanoxml	collections
InsECTJ	125.71	2.23	71.48
NN4J	616.28	88.85	343.98
Avg. NN4J	349.65	4.55	172.30
JavaSlicer	118.43	138.09	1099.94
Computing CP	2.44	0.95	3.44
Condition Generation	2.60	1.26	4.03
Total	881.29	222.88	1522.86

We can see that the whole process may take more than 1500 seconds. The runtime seems too long to be acceptable, however, our approach is generally used only once, just before the interactive debugging begins. As the process is fully automated, it can be automatically launched if a test case has failed. When the debugging developer begins her work, the breakpoints have already generated. Since human time is much more expensive than machine time, we believe it is worthwhile to use our approach to generate a group breakpoints which may help accelerate the subsequent manual debugging process. Besides, if developer chooses only one failed test case, and not too many passed test cases which may be quite similar to the failed one based on previous debugging experience, our approach will be more efficient.

B. Design of the User Study

We have performed a user study to further investigate the usefulness of our approach. Note that the study was performed using our previous version of implementation [14] which is more stable. Since the architecture remains the same and the difference lies only in implementation, we believe the result of the user study is still valid for the latest version.

The user study involved six student volunteers each of whom has at least three-year experience in Java programming and debugging. They also use Eclipse as their primary Java development environment. Each participant was assigned four debugging tasks and provided with the corresponding breakpoints generated by our tool in two tasks. As the experimental setup, we retrieved four programs from Apache Commons[4] and SIR [22] and manually seeded a bug in each of them. Each bug was seeded by slightly modifying, instead of adding or deleting, one line of code in the program so that there was an explicit faulty statement for the participants to find. We believe that these are typical bugs introduced by careless programming or incomplete understanding of some corner cases. In fact, the two bugs in the SIR programs are originally seeded bugs provided in the benchmark suite. The bugs in SIR have been seeded by following a strict process which simulates the real bug introducing process. Details of the programs and bugs are described in Table VI.

TABLE VI.
SUBJECT PROGRAMS AND SEEDED BUGS

Bug	Program	Location	Description
b_1	Commons v2.0	FastArrayList.java at line 417	false → true
b_2	Commons v2.0	CursorableLinkedList.java at line 265	|| → &&
b_3	jtopas v0.6	StandardTokenizer.java at line 1658	1 → -1
b_4	nanoxml v2.2	ContentReader.java at line 132	'&' → ' '

To avoid the bias caused by the differences between individuals, we divided the six participants into two groups

[4]http://commons.apache.org/

so that the groups had similar programming experience. The programming experience of a group was estimated with the sum of every group member's duration of programming. In addition, we arranged the task assignments as described in Table VII. Therefore, each group was allowed to use the generated breakpoints in only two debugging tasks, which reduced the chance that a better group might amplify (diminish) the estimated usefulness of the breakpoints by using (not using) them.

TABLE VII.
DEBUGGING TASKS ARRANGEMENT

Group	Person	b_1	b_2	b_3	b_4
g_1	p_1	not use	use	use	not use
	p_2	not use	use	use	not use
	p_3	not use	use	use	not use
g_2	p_4	use	not use	not use	use
	p_5	use	not use	not use	use
	p_6	use	not use	not use	use

As a preparation for the study, we generated the breakpoints for each bug using our tool implementation before the participants began debugging. Additionally we gave each breakpoint user a brief introduction to the basic idea of the breakpoint generation. The failing test case used in our breakpoint generation was pointed out as the symptom of the bug. Thus the participants could check whether they made the correct fix by running the test case. As the participants were not familiar with the programs, it might be too difficult for them to correctly discover and fix the bugs. Therefore, we loosely set a time limit for each debugging task, that is, we suggested that they could choose to give up the current task if they had spent more than 90 minutes on it. However, they could choose to go on debugging after 90 minutes. To analyze and evaluate the debugging sessions, we recorded all the actions on-screen using a screen recorder. We evaluated the debugging performance based on the time spent when the bug was successfully fixed. We also asked the participants who gave up their tasks to point out the statements they felt suspicious and evaluated the debugging work by estimating the distance between these statements and the faulty ones. The distance is roughly represented by the number of lines of code between two statements.

C. Results

Figure 8 shows the time spent on the debugging tasks. Each bar represents the duration of a debugging session of a participant if the bug was successfully fixed. Thus for each of the four bugs there can be six bars which correspond to the six participants' debugging time. Note that there are only three and two bars for $bug3$ and $bug4$, respectively, because three participants failed to fix $bug3$ and four participants failed to fix $bug4$.

We can see that the group which was provided with the generated breakpoints spent averagely less time for three of the four bugs. We may also notice that all the participants spent much more time on $bug3$ and $bug4$ than

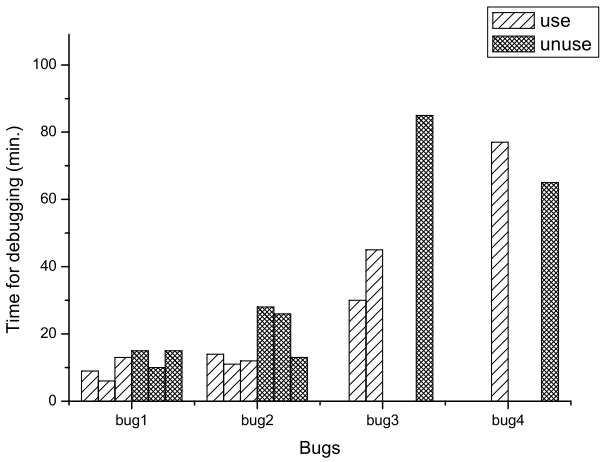

Figure 8. Time spent on each debugging task. There are only three and two bars for $bug3$ and $bug4$, respectively, because three participants failed to fix $bug3$ and four participants failed to fix $bug4$.

$bug1$ and $bug2$. It is because $bug3$ and $bug4$, which are seeded in nanoxml and jtopas, are far more complicated than $bug1$ and $bug2$, each of which involves just one Java source file. As a result, it is very difficult for the participants to find and fix the bugs without fully understanding the business logics of the two real-world applications. It is also the reason why some participants failed to fix $bug3$ and $bug4$.

To count in the performance of all the participants in all the debugging tasks, we rank them for each bug by following the rules below:

1) Participants who fixed the bug are ranked higher than those who failed to do so.
2) For successful participants, the less time they spent, the higher they are ranked.
3) For unsuccessful participants, we ranked them based on the evaluation of the suspicious statements they pointed out. The way of performing this evaluation has been described in Section VI-B.

We assign a score to each participant for each bug. More specifically, the score $(7 - n)$ is assigned to the participant who is ranked the *n*th place. For example, for a specific debugging task, the top 1 participant will score 6, while the last one will score 1. From Table VIII we can see that the participants using generated breakpoints have a higher score in average, which means they generally performed better than the other participants.

TABLE VIII.
SCORES OF PARTICIPANTS.

Bug	use			not use		
	p_1	p_2	p_3	p_1	p_2	p_3
b_1	6	5	3	4	2	1
b_2	6	5	3	4	2	1
b_3	6	5	2	4	3	1
b_4	5	3	1	6	4	2
Avg.	4.2			2.8		

Avg. means the average score of one participant for finishing debugging one bug with or without using the generated breakpoints.

We analyzed the recorded on-screen actions to inves-

tigate how much the generated breakpoints affected the debugging performance. During our analysis, we focused on how many times that each participant used print statements and breakpoints for debugging. Then we calculated the *using rate* with the following formula:

$$Rate_{using} = \frac{n_{ugb}}{n_{umb} + n_{up} + n_{ugb}}$$

where n_{up}, n_{umb}, and n_{ugb} stand for the number of times that the participant used print statements, manually added breakpoints, and generated breakpoints. Since print statements and breakpoints were major debugging utilities in the user study, we use the using rate to roughly estimate how often the generated breakpoints were used.

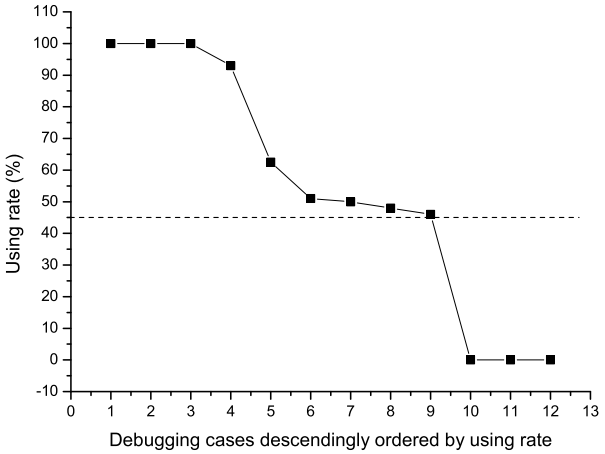

Figure 9. Using rate of the debugging cases supported by generated breakpoints

Figure 9 shows the using rates of the 12 debugging cases where generated breakpoints were provided. We can see that the using rate is higher than 45% in 9 out of 12 cases, which indicates that the generated breakpoints were frequently used during the debugging sessions. Moreover, in the three cases where the using rate is 0, the generated breakpoints were also used for many times for code inspection. However, the participants did not use them to suspend the program executions.

D. Additional Observations

During the analysis on the records, we found a common pattern of using the generated breakpoints. At the beginning of debugging sessions, the users usually browsed all the breakpoints, especially the conditional ones, and briefly inspected the source code near the breakpoint locations. Then they performed the ordinary debugging process, searching backwards from the failed assertion for clues to the bug. When the users decided to check run-time states at a program point, they tended to use a generated breakpoint near the point, instead of writing a print statement or setting a breakpoint by themselves.

For simple cases, such as $bug1$ and $bug2$, it was usually adequate to use the generated breakpoints to find the bugs. However, when the bugs was complicated, the user had to use utilities other than the generated breakpoints. Some participants commented that they found it helpful to open all the source files where the generated breakpoints were located, because they could obtain an overall impression of the execution traces. In these cases, the breakpoints can be viewed as a special kind of dynamic slice.

E. Threats to Validity

The main threat to the objective experimental results is that the subjects are all small programs. The current implementation generally suffers from high performance overhead. Therefore, it may be too time-consuming to do the experiments on large programs. However, in theory, the three core components can scale with proper optimization and implementations. In addition, we believe that, even for large programs, our technique should be used to aid fine-grained manual debugging in a relatively small scope, because the components of large programs are usually loosely coupled.

Since the user study was performed on a small sample of participants with a small number of bugs, the generality of the results is obviously limited. In addition, all the participants were unfamiliar with the subject programs, which deviates from practical debugging scenarios. It indeed affected the results, as some participants failed to accomplish their tasks. To reduce the possible bias, we carefully grouped the participants and allocated the debugging tasks, as described in Section VI-B. We also tried to objectively evaluate the participants' performances in unsuccessful cases and compared them using a ranking method.

VII. Related Work

Breakpoints are generally acknowledged as a helpful tool in program debugging. Chern and Volder [23] design a novel form of dynamic breakpoints named context-flow breakpoints to improve the debugging tools. The Control-flow Breakpoint Debugger(CBD) provides a pointcut language to specify dynamic breakpoints in program execution flows. Developers can use the language to attach much more specified runtime conditions to static breakpoints to produce dynamic ones, which are believed to be more helpful for debugging. Our work is greatly inspired by CBD. And we observe that although CBD facilitates developers in specifying breakpoints, it is still cumbersome to manually add breakpoints and describe them in detail. Our work aims to facilitate the process by automatically generating breakpoints based upon fault localization techniques.

Zeller et al. develop the Delta Debugging technique to systematically analyze the differences in program inputs, source codes, and run-time program states to isolate failure-inducing inputs, extract cause-effect chains, and link cause transitions to program failures [9], [24], [25]. Compared with the fully automated Delta Debugging techniques, our approach is primarily an aid to interactive debugging. Moreover, we put more 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 our approach, we borrow the part of memory graph comparison from [25] and reuse its DDstate implementation. Although DDstate implementation systematically changes the memory graph to isolate relevant states, we just compare the graphs and analyze the graph differences to find accessible suspicious states to generate breakpoint conditions.

As an essential step of debugging, fault localization has been extensively studied. Particularly the spectra-based fault localization techniques, which share the commonality of comparing program spectra, have been well investigated. Harrold et al. classify spectra-based approaches in [26] and adopt code coverage spectra to build the Tarantula tool [8]. Additionally, a variety of techniques, such as [10], [27], [28], have been proposed to improve the Tarantula approach. In [27], Hao et al. introduce a breakpoint-based interactive debugging aid, which is similar to our work. However, we combine different fault localization techniques to generate breakpoints, instead of using visualization to recommend breakpoint locations. Moreover, we generate conditions for some breakpoints to strengthen their usability. Another group of fault localization techniques [6], [7], [29] are based on comparing execution traces. They share the viewpoint that faults are more probable to be disclosed by comparing two runs that are similar rather than fundamentally different. In [6], Renieris and Reiss develop the nearest neighbor queries technique which adopts the k-NN algorithm to select a passing run that most resembles the failing one. This technique is claimed to be independent of input structure, which is good for building tools that are widely applicable. To achieve satisfactory report accuracy, it also assumes the availability of a large set of test cases, which might not always be possible in real world debugging situations. Although the nearest neighbor queries method is the base of our approach, we can probably overcome this restriction since the successive steps in our approach are capable of improving the accuracy. In addition to the fault localization techniques which report suspicious statements or states, Jiang and Su propose an approach to generate faulty control flow paths which are more useful to understand the bugs [30]. Similar to [30], our approach focuses on conditional statements. Thus a group of generated breakpoints are likely to convey control flow information to users. Although not refined by machine learning techniques, the breakpoints can also be useful in interactive debugging.

Program slicing was first introduced in [16], and proved to be helpful in program debugging [31], [32]. Since our breakpoint generation focuses on the dynamic aspects of programs, we use dynamic program slicing in our approach. While the slices are static positions in programs, we append dynamic information by attaching conditions to the calculated locations and producing conditional breakpoints. After all the size and precision of the slices can significantly affect the quality of the generated breakpoints. Therefore, we may further improve our approach by leveraging recent advancement in program slicing techniques, such as [33]. By combining dynamic and static program slicing, Ko and Myers develop a tool called Whyline [12] to support interrogative debugging. Using Whyline developers can easily query both dynamic and static dependencies in program elements and execution events, which may greatly accelerate the debugging process. Compared with Whyline, our tool is more of an enhancement to the breakpoint utility than a reinvented debugging approach. Thus our approach may not be as friendly as Whyline to users, but the generated breakpoints may be more acceptable to the developers. By recording every state change at runtime, omniscient debugger [34] 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, our approach aims to indicate which variables should be inspected at which statements.

VIII. Conclusion and Future Work

Breakpoint is one of the most frequently used utilities in debugging activities. In this paper, we have proposed an approach to generating breakpoints automatically. The evaluation has shown that the generated breakpoints can usually save some human effort for debugging.

In our future work, we will try to design new strategies to select better locations to generate breakpoints. Existing work on cause-effect relations in debugging, such as [25], is promising to improve our current strategy. In addition, our current approach generates the breakpoints and their conditions in one shot. It is interesting to investigate whether the breakpoints can be adaptively generated or adjusted as the interactive debugging session goes on.

Acknowledgment

We thank Professor Jianjun Zhao for his insightful discussion and the participants in the BPGen user study. The presented work was supported in part by National Natural Science Foundation of China (NSFC) (Grant No.61272102, 61100051, 91118004, 60970009).

References

[1] G. J. Myers, *The Art of Software Testing*. John Wiley and Sons, 1979.
[2] R. Pressman, *Software Engineering: A Practitioner's Approach, Sixth Edition*. McGraw-Hill, 2004.
[3] M. Telles and Y. Hsieh, *The Science of Debugging*. Coriolis Group Books, 2001.
[4] G. C. Murphy, M. Kersten, and L. Findlater, "How Are Java Software Developers Using the Eclipse IDE?" *IEEE Software*, vol. 23, no. 4, pp. 76–83, 2006.
[5] H. Agrawal, J. Horgan, S. London, W. Wong, and M. Bellcore, "Fault localization using execution slices and dataflow tests," in *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, 1995, pp. 143–151.

[6] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," *23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 30–39, 2003.
[7] L. Guo, A. Roychoudhury, and T. Wang, "Accurately choosing execution runs for software fault localization," *15th International Conference on Compiler Construction*, vol. 3923, pp. 80–95, 2006.
[8] 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*, pp. 273–282, 2005.
[9] H. Cleve and A. Zeller, "Locating causes of program failures," *27th International Conference on Software Engineering*, pp. 342–351, 2005.
[10] T.-Y. Huang, P.-C. Chou, C.-H. Tsai, and H.-A. Chen, "Automated fault localization with statistically suspicious program states," *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pp. 11–20, 2007.
[11] D. Jeffrey, N. Gupta, and R. Gupta, "Fault localization using value replacement," *International Symposium on Software Testing and Analysis*, pp. 167–178, 2008.
[12] A. J. Ko and B. A. Myers, "Debugging reinvented: asking and answering why and why not questions about program behavior," *Proceedings of the 30th International Conference on Software Engineering*, pp. 301–310, 2008.
[13] B. Korel and J. W. Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155–163, 1988.
[14] C. Zhang, D. Yan, J. Zhao, Y. Chen, and S. Yang, "BPGen: an automated breakpoint generator for debugging," in *ICSE '10*, 2010, pp. 271–274.
[15] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition*. Morgan Kaufmann, 2005.
[16] M. Weiser, "Program slicing," *Proceedings of the 5th International Conference on Software engineering*, pp. 439–449, 1981.
[17] X. Zhang, H. He, N. Gupta, and R. Gupta, "Experimental evaluation of using dynamic slices for fault location," *Proceedings of the Sixth International Workshop on Automated Debugging*, pp. 33–42, 2005.
[18] T. Zimmermann and A. Zeller, "Visualizing memory graphs," in *Software Visualization*, 2001, pp. 191–204.
[19] 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*, pp. 45–49, 2005.
[20] C. Hammacher, "Design and implementation of an efficient dynamic slicer for Java," Bachelor's Thesis, November 2008.
[21] S. Horwitz and T. W. Reps, "The use of program dependence graphs in software engineering," *Proceedings of the 14th International Conference on Software Engineering*, pp. 392–411, 1992.
[22] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact." *Empirical Software Engineering: An International Journal*, vol. 10, no. 4, pp. 405–435, 2005.
[23] R. Chern and K. D. Volder, "Debugging with control-flow breakpoints," *Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, vol. 208, pp. 96–106, 2007.
[24] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.
[25] A. Zeller, "Isolating cause-effect chains from computer programs," *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 1–10, 2002.
[26] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An empirical investigation of the relationship between spectra differences and regression faults," *Software Testing, Verification and Reliability*, vol. 10, no. 3, pp. 171–194, 2000.
[27] D. Hao, L. Zhang, L. Zhang, J. Sun, and H. Mei, "Vida: Visual interactive debugging," in *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 583–586.
[28] X. Wang, S. C. Cheung, W. K. Chan, and Z. Zhang, "Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization," in *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 45–55.
[29] W. N. Sumner and X. Zhang, "Algorithms for automatically computing the causal paths of failures," in *FASE '09: Fundamental Approaches to Software Engineering, 12th International Conference*. Springer, 2009, pp. 355–369.
[30] L. Jiang and Z. Su, "Context-aware statistical debugging: from bug predictors to faulty control flow paths," in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. New York, NY, USA: ACM, 2007, pp. 184–193.
[31] M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, 1982.
[32] F. Tip, "A survey of program slicing techniques." Amsterdam, The Netherlands, The Netherlands, Tech. Rep., 1994.
[33] M. Sridharan, S. J. Fink, and R. Bodik, "Thin slicing," in *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2007, pp. 112–122.
[34] B. Lewis, "Debugging backwards in time," *International Workshop on Automated Debugging*, pp. 225–235, 2003.

Cheng Zhang is a PhD student in the Department of Computer Science and Engineering, Shanghai Jiao Tong University. He received bachelor's degree in software engineering and master's degree in computer application technology from Shanghai Jiao Tong University. His research interests include program analysis, program debugging, and optimization.

Juyuan Yang is an undergraduate student of School of Software, Shanghai Jiao Tong University. He has been studying and working on applications of program analysis techniques. His research interests include software engineering, programming languages, program analysis, testing and verification, and data mining.

Dacong Yan is a PhD student at the Computer Science and Engineering Department of Ohio State University. He received Bachelor's degree in software engineering from Shanghai Jiao Tong University. His general research interests are developing static and dynamic program analyses in various application domains such as compiler optimization, performance tuning, and security.

Shengqian Yang received his B.S. degree in software engineering from Shanghai Jiao Tong University 2010. He is a PhD student at the Computer Science and Engineering Department of Ohio State University. His research interests are programming analysis and software engineering. In particular, he is interested in performance analysis, scripting language and concurrency.

Yuting Chen received the B.S. and M.S. degrees in Computer Science from Nanjing University, China, in 2000 and 2003, respectively. He received the Ph.D. degree in Computer Science from Hosei University in 2007. After that, he continued his research in University of Texas at Dallas as a research scholar. He is currently an assistant professor at the School of Software, Shanghai Jiao Tong University. His research interests include dependable systems development based on formal engineering methods, software verification and validation, applied formal methods, etc.