Feedback-Based Debugging

Yun Lin[1], Jun Sun[2], Yinxing Xue[3], Yang Liu[3], and Jinsong Dong[1]
[1]School of Computing, National University of Singapore, Singapore, [2]Singapore University of Technology and Design, Singapore, [3]School of Computer Engineering, Nanyang Technological University, Singapore

***Abstract*—Software debugging has long been regarded as a time and effort consuming task. In the process of debugging, developers usually need to manually inspect many program steps to see whether they deviate from their intended behaviors. Given that intended behaviors usually exist nowhere but in human mind, the automation of debugging turns out to be extremely hard, if not impossible.**

In this work, we propose a feedback-based debugging approach, which (1) builds on light-weight human feedbacks on a buggy program and (2) regards the feedbacks as partial program specification to infer suspicious steps of the buggy execution. Given a buggy program, we record its execution trace and allow developers to provide light-weight feedback on trace steps. Based on the feedbacks, we recommend suspicious steps on the trace. Moreover, our approach can further learn and approximate bug-free paths, which helps reduce required feedbacks to expedite the debugging process. We conduct an experiment to evaluate our approach with simulated feedbacks on 3409 mutated bugs across 3 open source projects. The results show that our feedback-based approach can detect 92.8% of the bugs and 65% of the detected bugs require less than 20 feedbacks. In addition, we implement our proof-of-concept tool, *Microbat*, and conduct a user study involving 16 participants on 3 debugging tasks. The results show that, compared to the participants using the baseline tool, *Whyline*, the ones using *Microbat* can spend on average 55.8% less time to locate the bugs.

I. Introduction

Software debugging is often regarded as one of the most time-consuming tasks in software development and maintenance [13], [16]. Given an observable fault, developers usually need to start with the fault-revealing code, speculate where the bugs are, and inspect the code line by line (or sometimes step by step) with the intended code specification *in mind*. When the code gets complicated, such a manual process of debugging inevitably demands huge amount of time and mental efforts.

Researchers have proposed a lot of techniques for automation of software debugging, such as spectrum-based fault localization [9], [10], [30], [32], [33], delta-debugging [15], [18], [25], [28], [36], and dynamic trace recording [12], [21], [24], [26], [27], [31], [35]. Spectrum-based fault localization regards test cases as executable requirement. Given a set of test cases, it quantifies the suspiciousness of source code lines by comparing the code coverage of passed or failed test cases. Delta-debugging analyzes differences between passed and failed test cases, such as test inputs and running program states, so as to simplify the test inputs [36], [38] and isolate root cause variable of bug [37]. However, in the process of development, developers usually lack sufficient passed test cases [14] to apply or take full advantage of these techniques. Some dynamic trace recording techniques, such as omniscient debugging [12], [26], can record the execution trace for a single run and allow developers to trace back and analyze the faults. Nevertheless, when the trace length gets long (especially caused by loops), the effort for stepwise checking becomes overwhelming.

In this paper, we propose a tool-supported and feedback-based debugging approach, which requires only one failed test case and aims to reveal the root-cause step in the execution. Our rationale lies in the observation that, the specification of detailed code usually exists nowhere but in human mind. Therefore, we leverage light-weight user feedback as "partial specification" to feed the debugger so that it can recommend suspicious steps. Given a buggy program, we first build a trace model which records the execution trace and captures causality relations (i.e., data/control dominance relation) among the steps. On each trace step, we allow the developers to provide four types of feedback (i.e., correct, wrong variable value, wrong path, and unclear). Our approach then takes the feedback and recommends suspicious steps based on causality relation among trace steps. After collecting a number of feedbacks, our approach begins to learn and approximate bug-free paths on trace, which helps reduce the number of feedbacks to expedite the debugging process. This iterative process starts with a user feedback on a fault-revealing trace step and finishes when the root-cause step is recommended.

We implement our approach as an Eclipse plugin, *Microbat* (A demo video of *Microbat* is available at [4]). We first conduct a simulation experiment by using *Microbat* to find 3409 mutated bugs with simulated feedbacks on three open source projects. The results show that *Microbat* is able to detect 92.8% of the mutated bugs and 65% of detected bugs require less than 20 feedbacks. In addition, we conduct a user study involving 16 participants on 3 real-world bugs. The result shows that, compared to the participants using the baseline tool *Whyline* [11], the ones using *Microbat* can spend on average 55.8% less time to locate the bugs.

This paper makes the following contributions: 1) We propose a feedback-based debugging approach, which incorporates four types of feedback to recommend suspicious steps. 2) We develop *Microbat* tool for the practical use of our feedback debugging approach; 3) We conduct both simulation experiment and user study to evaluate our approach and tool. The results show that *Microbat* is both effective and practical.

The rest of the paper is structured as follows. Section II presents a motivating example. Section III describes our approach. Section IV presents our tool *Microbat*. Section V evaluates our approach with a simulation experiment. Section VI shows our user study on real-world bugs. Section VII reviews

TABLE I
DEBUGGING CODE EXAMPLE

```
1  public int calculate(String expr){
2    int brktStartIdx = -1;
3    while(containsBracket(expr)){
4      char[] list = expr.toCharArray();
5      for(int i=0; i<list.length; i++){
6        if(ch == '(')
7          brktStartIdx = i;
8        else if(ch == ')'){
9          String simpleExpr = expr.substring(brktStartIdx+1, i);
10         int value = evaluateSimpleExpr(simpleExpr);
11         String beforeExpr = expr.substring(0, brktStartIdx);
12         String afterExpr = (i >= expr.length()) ? ""
13           : expr.substring(i + 1, expr.length());
14         expr = beforeExpr + value + afterExpr;
15         break;
16       }
17     }
18   }
19   int result = evaluateSimpleExpr(expr);
20   return result;
21 }
```

related work. Section VIII concludes the paper.

II. MOTIVATING EXAMPLE

Table I shows our motivating example, which is adopted from a code training website [6]. Given a valid algorithmic expression consisting of integers, brackets, or plus/minus signs, e.g., "1-((1+2)-1)", this program should compute its correct value. Overall, the program parses the expression by iteratively replacing the expression inside the most inner pair of brackets with its value (line 9–15). For example, the expression "1-((1+2)-1)" will be iteratively reduced into expressions "1-(3-1)" and "1-2". Finally, it will be evaluated to a number returned as the result (line 19). In our example, however, given the complicated expression of "(((1+((1+2)+(2-1))-(1-3))+1)+1)+1", it returns a wrong value of 6 instead of the correct value of 10.

With a traditional debugger, developers usually need to set a number of breakpoints for tracking down the bug. However, they will have to answer some following questions.

(1) Where to set breakpoints? In our example, given that the *result* variable in line 20 is wrong, every statement possibly influencing it is suspicious, which makes almost every line in Table I as a potential breakpoint.

(2) How many breakpoints are appropriate? Too many breakpoints may suspend the debugging execution when unnecessary. However, any miss of a breakpoint may cause the execution suspended after the bug has already occurred, which requires the developer to re-run the program from the very beginning.

(3) How to avoid over inspection effort caused by loop? When the breakpoints are set inside a (nested) loop, developers have to manually inspect variable values each time a breakpoint is reached, e.g., a breakpoint set on line 11 in Table I. With the number of iterations increases, the effort of inspecting variable values soars dramatically.

In this work, we propose *Microbat* to address the above issues. For the case in Table I, *Microbat* first generates the execution trace by a single run and records all the read or written variables and their values in each trace step. Given the visualized trace (see Section IV), developers are able to start debugging in a backward way. Specifically, developers can start from the very end of the trace where the fault is observed, and provide his feedback on this step, such as which variable in this step is with wrong value, or whether this step should be executed, then *Microbat* is able to recommend certain step responsible for its cause.

In our example, the developer can first observe the program state in the step running into line 20 in Table I, where the *result* variable has the wrong value of 6 instead of the expected 10. Thus, he can select this variable *on this step*, indicating its wrong value as feedback, and ask *Microbat* to recommend a suspicious step for further inspection. Using the feedback, *Microbat* recommends a step by (1) simple causality analysis, (2) bug-free path inference, and (3) clarity guidance.

Simple Causality Analysis. Simple causality analysis aims to parse the dynamic data/control dominance relation between steps to alleviate the burden of setting breakpoints. In above case, *Microbat* first recommends the most recent step writing the *result* variable (data dominance), i.e., the step running into line 19. On this step, it reads a variable *expr* of value *"5+1"* and writes the variable *result* of value *6*.

Given that *"5+1"* equals 6 and the value of the written variable *result* has been indicated as wrong, the read *expr* variable must be wrong. Thus, the developer can further select the *expr* variable to indicate its wrong value as feedback. With simple causality analysis, *Microbat* then recommends a step running into line 14, which writes the *expr* variable.

Bug-free Path Inference. Bug-free path inference aims to reduce the inspection effort. With only above causality analysis, the developer will repeatedly inspect the steps running into line 14 and line 10 in every iteration. In the worst case, he would need to go through all the iterations if the bug happens at the very beginning of the execution.

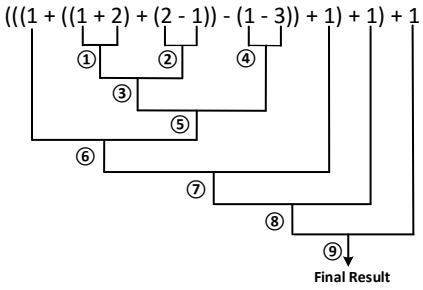

Fig. 1. Execution of Example Program

Figure 1 shows the 9 iterations along with their execution order when parsing the expression "(((1+((1+2)+(2-1))-(1-3))+1)+1)+1". Since the developer inspects the variables in a backward way, he will first give wrong-variable-value feedback on the 9th and 8th iteration sequentially. *Microbat* then approximates some possible bug-free paths along the loop trace. In this case, *Microbat* can skip the 6th and 7th iteration, and recommend a step in the 5th iteration. The rationale is as follows. Based on developer's feedback, the bug does not occur in the 8th or 9th iteration. In addition, *Microbat* finds that the cases in 6th–9th iteration are similar in that they all parse the addition of two positive integers, for example "5+1" in 9th iteration, "4+1" in the 8th iteration. Therefore, *Microbat* approximates that the bug may not happen in the 6th or 7th

iteration either. Hence, *Microbat* stops in the 5th iteration as the case of a positive integer (4) minus a negative integer (-2) has never been encountered.

The developer now inspects a step running into line 10 in Table I, in which the read *expr* is "4- -2" while the written value of *value* variable is 2. Then, the developer can select the returned variable from *evaluateSimpleExpr()* method so that *Microbat* can further conduct simple causality analysis inside the method invocation.

Clarity Guidance. In some cases, developers could get lost when inspecting the correctness of program state. *Microbat* enables the developers to provide an *unclear* feedback, then *Microbat* will try to suggest its *context* step such as method invocation or loop head (i.e., the step starting a loop iteration). Suppose the developer cannot make sure the correctness of a step s running into line 14, he can provide the *unclear* feedback so that *Microbat* will recommend its loop head step running into line 5 (by one unclear feedback) or line 3 (by two unclear feedbacks). By this means, he can be aware of the context information such as (1) which iteration of Figure 1 is s located in and (2) what is the reduced expression at the beginning of this iteration. With feedbacks provided on context steps, *Microbat* manages to present bigger picture and gradually guides the developer back to understand the step he gets lost at the first place.

III. Approach

Given a trace of steps, our approach aims to find the root-cause step which deviates from developer's expectation and eventually causes the observable fault after program execution.

A. Trace Model

Given a run of the buggy program, we can obtain a **trace** consisting of a number of **step**s. Each step corresponds to an executed source code line, which can *define* (i.e., write) or *use* (i.e., read) some **variable**s. Given a variable var, if var is defined by step s_1 while used by step s_2, then we say that step s_1 *data dominate*s s_2 on var. In addition, the variable var is called as the **attributed variable** of the data dominance relation. On the other hand, given a conditional statement con_stat, if con_stat is executed in step s_1 while the evaluation value of con_stat (i.e., true or false) decides the execution of step s_2, then we say that step s_1 *control dominate*s s_2. Given two steps s_1 and s_2, if s_1 control or data dominates $s2$, then we say that s_1 is control or data **dominator** of s_2, and s_2 is the control or data **daminatee** of s_1. In addition, we say that s_1 is the **contextual parent** of s_2 if either of following conditions happens:

- s_1 starts a loop iteration l, and s_2 is executed in l but not in any nested loop iteration or method invocation in l.
- s_1 starts a method invocation m and s_2 is executed in m but not in any nested method invocation or loop iteration in m.

The contextual parent-child relation can organize our trace into a **step tree**, in which the root is the entry method and the leaves are the steps invoking no method and starting no loop iteration. Given a step, we define its layer on step tree as its **abstract level**. By default, the abstract level of the entry method is 0.

B. Recommendation Mechanism

We support four types of feedback as follows:

- **Correct Step:** The step is executed in correct control flow and all the values of visible variables in this step are correct.
- **Wrong Variable Value:** At least one variable in this step is of wrong value. Once a developer provides such feedback, he should further select the specific variables of wrong value.
- **Wrong Path:** The step should not be executed.
- **Unclear:** The developer is not confident to make any of the above feedback on this step.

We consider correct-step, wrong-variable-value, and wrong-path feedback as **clear** feedback. We recommend a suspicious step if the developer provides a clear feedback and recommend a step to help understand the code if he provide an unclear feedback. For clarity, we start illustrating our approach when developers only provide clear feedback, then we proceed to the case when they provide unclear feedback.

1) Overall Mechanism with Clear Feedbacks: When a developer specifies an incorrect step by providing wrong-variable-value or wrong-path feedback, we move *forward* on trace to a suspicious previous step by data or control dominance relation. The developer can iteratively provide feedbacks and move forward to locate the root-cause step. However, moving forward only by dominance relation can be either (1) too slow so that it requires a great amount of feedbacks or (2) too fast so that we skip the root-cause step by a single dominance relation. Figure 2 uses a state machine to present our overall recommending mechanism with only clear feedbacks. The states consists of:

- Simple Casuality Analysis: we simply recommend steps based on dominance relation.
- Bug-free Path Inference: we skip some inferred and approximated bug-free paths to move "faster".
- Inspect Details: when we find moving (and recommending steps) by dominance relation is too fast, we go through steps in between a dominance relation.

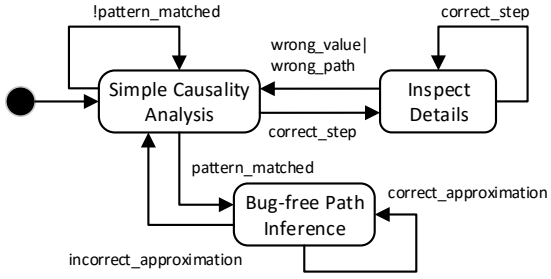

Fig. 2. Overall Recommending Mechanism with Clear Feedback

In Figure 2, we assume the developer starts debugging from a fault-revealing step, therefore, he starts in *Simple Causality Analysis* state. The developer stays in this state until we find recommending steps by dominance relation is either too slow or too fast.

We detect the case of being too slow by inferring bug-free paths and summarizing bug-free path patterns to approximate

potential bug-free paths. Once we find some path matches bug-free path patterns (i.e., bug-free prone) in *Simple Causality Analysis* state, we transfer to *Bug-free Path Inference* state where we expedite the moving by skipping steps. In *Bug-free Path Inference* state, we also reply on user feedback to confirm our approximation of bug-free paths. When our approximation is confirmed, we stay in this state. Otherwise, we transfer back to *Simple Causality Analysis* state.

We detect the case of being too fast as follows. Given a path of dominance relation, if the developer provides a wrong-variable-value or wrong-path feedback at the end step and a correct feedback at the start step, the root-cause step must lie in between them. In such case, we enter *Inspect Detail* state. In this state, if the developer provides correct feedback, we sequentially explore and recommend the steps between the start and end step. Otherwise, we transfer back to *Simple Causality Analysis* state.

Since *Simple Causality Analysis* and *Inspect Detail* state are straight-forward, we focus our illustration on *Bug-free Path Inference* state.

2) Bug-free Path Inference: Bug-free path inference aims to infer bug-free paths by user feedback and approximate potential bug-free paths. During debugging, we discriminate the steps on more bug-free paths and recommend those on more "bug-prone" paths to expedite the debugging process.

In our approach, we first infer and record the bug-free path based on user feedbacks. We extract path pattern for each bug-free path. We approximate the paths conforming to the pattern of a bug-free path to be bug-free. During simple causality analysis, if a path of dominance relation is approximated to be bug-free, we will skip this path and recommend a further forward step. Moreover, when we detect that we have over-approximated some bug-free paths, we adopt a binary-search based mechanism for complement.

Next, we explain the details of pattern extraction (Section III-B2a), step skipping (Section III-B2b), and binary search (Section III-B2c).

a) Pattern Extraction: Pattern extraction aims to identify bug-free paths and extract their pattern keys.

Bug-free Path: Given a step $step$ on trace, if one of its read variables is marked as being of wrong value, then we call this step as an **attributed step**. An attributed step means that its incorrectness is spread from some step executed before. Given a path of a data dominance relation on the trace, if it satisfies that both its start step and end step are attributed steps, we consider it as a **bug-free path**[1].

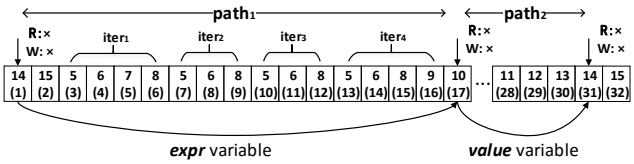

Fig. 3. Path Example

Figure 3 shows a part of trace of the buggy program in Table I. Each rectangle represents a trace step, the upper number indicates the corresponding line number in Table I and the lower number in brackets indicates its order. The dots between the 17th step and 28th step indicate that the steps inside method invocation in line 11 in Table I are omitted. In addition, the curve lines indicate data dominance relations and their attributed variables. Suppose the developer provided his wrong-variable-value sequentially on *value* variable on the 31th step (line 14) and *expr* variable on the 17th step (line 10), then we have a bug-free path $< step_{17}, step_{31} >$.

Pattern Key Extraction: For a path, we abstract it into a more compact form, i.e., pattern key, so that the paths conforming to the pattern key is considered as similar.

TABLE II
PATH ABSTRACTION EXAMPLE

```
1  private int evaluateSimpleExpr(String simpleExpr) {
2    String[] operators = parseOperators(simpleExpr);
3    String[] numberStrings = simpleExpr.split("\\+|-");
4
5    String numString1 = retrieveNum(numberStrings, 0);
6    Integer num1 = Integer.valueOf(numString1);
7    for (int i = 0; i < operators.length; i++) {
8      String operator = operators[i];
9      String numString2 = retrieveNum(numberStrings, i+1);
10     if (operator.equals("+")) {
11       num1 = num1 + Integer.valueOf(numString2);
12     } else if (operator.equals("-")) {
13       num1 = num1 - Integer.valueOf(numString2);
14     }
15   }
16   return num1;
17 }
```

Table II shows the details of the method invoked in line 10 and 19 in Table I. The execution of the loop (line 7–15) causes the iterations going through either path $A =< 7, 8, 9, 10, 11, 12, 15 >$, $B =< 7, 8, 9, 10, 12, 13, 14, 15 >$ or $C =< 7, 8, 9, 10, 12, 15 >$ in which the elements represent line number. In this example, each iteration path (i.e., A, B, or C) represents the case when evaluating the addition of two numbers, subtraction of two numbers, or simply one number.

Given a path p containing a number of iterations, we approximate its semantic similarity with other paths by (1) whether they contain similar iterations and (2) whether the iterations are executed in similar order. Thus, we regard p as a string of iterations, e.g., $p =< A, A, B, B, B, A >$ or $< A, B, A, B >$. Then we summarize p into a regular expression by abstracting its consecutive repetitive substrings [8], e.g., A^*B^*A or $(AB)^*$. Such regular expression is the **pattern key** of p.

Note that, if p contains nested loop iterations, e.g, $<< A, A, A, B, B, B >, < A, A, B, B >>$, we first reduce p into p' by generating pattern keys for iterations in the most inner loop, e.g., $p' =< A^*B^*, A^*B^* >$. Thus the nested hierarchies in p' is one level flatter. Then we apply the same procedure on p' to further flatten the nested hierarchies, e.g., $(A^*B^*)^*$. Such a procedure is applied until all the nested hierarchies in p is flattened into one patten key.

For the example in Table II, based on the user feedbacks, if a path $< A, B, B >$ (A for addition and B for subtraction) can be implied to be bug-free, thus its pattern key is AB^*. Then we will consider another path $< A, B >$ as more likely

[1]It is possible that there are over two bugs in the trace, thus a bug may exist in our defined "bug-free path". In such case, our approach focuses on locating the first bug appearing in the trace.

Algorithm 1: Recommendation with Step Skipping

Input : a wrong-variable-value step $step$
Input : a wrong read variable of $step$, var
Input : existing bug-free paths bug_free_paths
Output: recommended step $step_{rec}$

1 $step_d \leftarrow dom(step, var)$;
2 $path_s \leftarrow < step_d, step >$;
3 **if** $path_s.conformTo(bug\_free\_paths)$ **then**
4 $is\_skip \leftarrow$ true;
5 **while** is_skip **do**
6 $is\_skip \leftarrow$ false;
7 **for** *each read variable* $read\_var$ *on* $step_d$ **do**
8 $step_{new\_d} \leftarrow dom(step_d, read\_var)$;
9 $path \leftarrow < step_{new\_d}, step_d >$;
10 **if** $path.conformTo(bug\_free\_paths)$ **then**
11 $step_d \leftarrow dom(step_{new\_d}, read\_var)$;
12 $is\_skip \leftarrow$ true;
13 **break**;
14 **end**
15 **end**
16 **end**
17 **end**
18 **return** $step_d$;

Algorithm 2: Recommendation with Binary Search

Input : a list of skipped dominators $list$
Output: recommended step $step_{rec}$

1 $start \leftarrow 0, end \leftarrow list.length - 1, current \leftarrow start$;
2 **while** $start < end$ **do**
3 $feedback \leftarrow$ user provide feedback on $list[current]$;
4 **if** $feedback$ *is correct* **then**
5 $current \leftarrow \frac{1}{2}(current + end)$; $start \leftarrow current$;
6 **else if** $feedback$ *is wrong-variable-value* **then**
7 $path \leftarrow < list[current - 1], list[current] >$);
8 $var_{attr} \leftarrow findAttr(path, list[current])$;
9 **if** $var_{attr} = feedback.var$ **then**
10 $current \leftarrow \frac{1}{2}(current + start)$; $end \leftarrow current$;
11 **else**
12 **return** $simCA(list[current], feedback)$;
13 **end**
14 **else**
15 **return** $simCA(list[current], feedback)$;
16 **end**
17 **end**
18 **return** $list[current]$;

to be bug-free as it conforms to the pattern key (AB^*) of the former path. The interpretation is as follows. If adding numbers once then subtracting numbers twice ("3+4-1-5") is bug-free, then we *approximate* that adding numbers once then subtracting numbers once ("1+3-1") is more likely to be bug-free, comparing to other cases such as subtracting numbers twice ("1-2-3").

b) Step Skipping: When the developer provides feedbacks and gets recommended steps with simple casuality analysis, he is also marking the bug-freeness of the paths of dominance relation. For the example of Figure 3, if the developer provides a wrong-variable-value feedback on *value* variable on the 31th step, then provides a wrong-variable-value on *expr* variable on the recommended data dominator, i.e., the 17th step. Apart from recommending the 1st step as the data dominator, we can also mark the path starting with the 17th step and ending with the 31th step as *bug-free*. With the increase of feedbacks, we can have more bug-free paths for us to conduct step skipping.

Algorithm 1 shows how we skip steps with regard to recorded bug-free paths. Given a step $step$ where the developer provides a wrong-variable-value feedback on its read variable var, we first find the data dominator $step_d$ of $step$ by var (line 2). Instead of directly returning $step_d$, we check whether the path $path_s$ of $< step_d, step >$ conforms to one of the bug-free paths bug_free_paths (line 2–3). If not, we return $step_d$, otherwise, we consider $path_s$ prone to bug-free and try to skip $step_d$ as follows. We go through all the read variables on $step_d$ and check whether there exists a path $path$ (starting by $step_d$'s dominator $step_{new\_d}$ and ending by $step_d$) is also bug-free prone (line 7–9). If yes, we can further skip $step_{new\_d}$ and check its even forward dominators (which is assigned to $step_d$) on trace (line 11–13). Otherwise, we stop skipping and return the most forward dominator $step_d$.

c) Binary Search: As mentioned before, our skipping strategy may over-approximate bug-free paths, which makes us over-skip some steps. We can detect such case when the developer provides a correct-step feedback immediately after we recommend a step by step skipping. Given a previous step marked as correct, and a later step marked as wrong-variable-value, the root cause step should lie in between. Therefore, we adopt a binary search based strategy as Algorithm 2.

Step skipping will result in a sequential list of skipped dominators $list$ in execution order. Algorithm 2 applies binary search on $list$. During the binary search, if the developer provides a correct feedback, we search backward on trace in a binary way (line 4–5). On the contrary, if the developer provides a wrong-variable-value feedback, we search forward on trace in a binary way (line 6–13).

Moreover, we leverage developer's feedback on dominators in $list$ to confirm our approximation during step skipping. When the developer provides a wrong-variable-value feedback on certain dominator $list[current]$ in $list$, we can check whether the wrong variable chosen in this feedback is the same to attributed variable of the skipped path of dominance relation $< list[current - 1], list[current] >$ (line 7–8). If yes, we can confirm that we make correct approximation when skipping $< list[current - 1], list[current] >$ and continue the binary search procedure (line 9–10). Otherwise, we regard the developer is no longer debugging on the track of our approximation during step skipping. Therefore, we stop the binary search and adopt simple causality analysis on $list[current]$ instead (line 11–13). With the same reason, we stop the binary search in the same way when the developer provides a wrong-path feedback (line 15).

3) Clarity Guidance: Once the developer cannot decide the correctness of a step, he can provide an unclear feedback. We aim to guide the developer better understand the unclear step so that he can resume where he gets lost.

Algorithm 3 shows how clarity guidance works. Given an unclear step $step$, we first back up the debugging context of $step$ such as the details of step skipping or binary search, in addition, we maintain a stack $stack$ and push $step$ into it (line 1). Then we retrieve the context step $step_{con}$ of $step$ by $getContext()$ method (line 2). The $getContext()$ method

Algorithm 3: Recommendation with Unclear Feedback

Input : an unclear step $step$
Output: recommended step $step_{rec}$

```
1  back up context of step; stack ← ∅; stack.push(step);
2  step_con ← getContext(step) and recommend step_con;
3  while true do
4      feedback ← user provide feedback on step_con;
5      if feedback is unclear then
6          step_con ← getContext(step_con);
7          stack.push(step_con);
8          recommend step_con;
9      else if feedback is correct then
10         step_con ← stack.pop();
11         if stack = ∅ then
12             resume context of step_con;
13             return step_con;
14         else
15             recommend step_con;
16         end
17     else
18         return simCA(step_con, feedback);
19     end
20 end
```

returns the step executed before $step$ if its abstract level of $step$ is 1, and returns the contextual parent (i.e., loop head or method invocation step) of $step$ otherwise (see definition in Section III-A). The context aims to provide a big picture so that the developer can better understand the unclear step.

Then, the developer needs to provide feedback on $step_{con}$. If the developer provides an unclear feedback on $step_{con}$, we further retrieve and recommend its context step and push it into $stack$ (line 5–8). If the developer provides a correct feedback on $step_{con}$, we pop $stack$ to get the most recent unclear step and assign it to $step_{con}$ (line 9–10). In this case, if $stack$ is empty, it means that $step_{con}$ is the very first step he gets unclear, therefore, we resume its backed up context and recommend $step_{con}$ (line 11–13). Otherwise, we consider that the developer is partially clear as he can now tell the correctness of the context of some unclear step. Thus, we recommend $step_{con}$ for his further feedback (line 14–16). In addition, if the developer provides a wrong-variable-value or wrong-path feedback, we consider that the developer has gotten new debugging clue. Hence, we stop the procedure and conduct simple causality analysis for $step_{con}$ (line 18).

IV. Tool Support

We implemented our approach as an Eclipse plugin. A screenshot can be checked at *Microbat* Github website [3]. *Microbat* consists of three views, i.e., *Trace* view, *Feedback* view, and *Reason* view. The recorded trace will be presented in *Trace* view. In *Trace* view, the steps are organized in a tree structure conforming to the contextual parent-child relation and each step is labeled with its execution order, class file name, and line number. Once the developer clicks a step on *Trace* view, the corresponding line of code will be highlighted in Java Editor, and its detailed information will be showed on *Feedback* view. At the top of *Feedback* view shows the four types of feedback. Given a selected step, *Feedback* view lists its read and written variables, as well as a snapshot of program states. Once a feedback is provided, the developer can click the *Find Bug* button to make *Microbat* to recommend a step. After a step is recommended, *Reason* view shows its recommendation explanation in natural language. In addition, the developer can click *Undo* button to get back to the state before the recommendation for the current step.

V. Simulation Experiment

We conduct a simulation experiment to answer the following research questions:

- **RQ1**: How effectively and efficiently can *Microbat* facilitate the debugging process?
- **RQ2**: What is the contribution of bug-free path inference to the debugging process?
- **RQ3**: What is the impact of unclear feedback?

In the simulation experiment, we generate mutants which can kill a given test case as buggy code, and apply *Microbat* with simulated feedbacks on the trace of mutants to see whether *Microbat* can recommend a step running into where the mutation happens.

We first collect test cases from three Apache open source projects (see Table III). For each passed test case, we mutate its tested code with a standard mutator. The mutator replaces algorithmic operators, logical operators, and number constants, e.g., replacing "+" with "-". In each mutation, only one source code line is modified. If a mutation kills the test case, we generate the correct trace before mutation, $trace_c$, and the buggy trace after mutation, $trace_m$. Then, we can reference $trace_c$ to check the correctness of the steps in $trace_m$.

We customize a dynamic programming algorithm [19] [23] to match the steps between the two traces. If a step in $trace_m$ cannot be matched to a step in $trace_c$, we simulate a wrong-path feedback. Otherwise, we difference the read and written variables between two matched steps to check whether the variable values on the step of mutated trace are correct. If not, we simulate a wrong-variable-value feedback, otherwise, we simulate a correct-step feedback.

As for the unclear feedback, we design the simulation as follows. If the step is the first fault-revealing step at the end of the trace, the "simulated developer" will not provide a unclear feedback. Otherwise, given a step s which has an abstract level (see definition in Section III-A), l, and it is the kth times checked by our "simulated developer", then, the probability to simulate an unclear feedback is $P(l,k) = (1 - \frac{1}{e^{l-1}})/k$. Intuitively, this is designed such that the lower level s is or the less times s is checked, the more likely an unclear feedback is simulated on s.

We call each simulated debugging process on a mutated trace as a **trial**. In a trial t, we consider the mutated line of source code $line_m$ as the root cause of the bug. Let the trace length be l_t, if *Microbat* can recommend a suspicious step which runs into $line_m$ within l_t feedbacks, we consider the trial as effective. In addition, we limit the generated trace length for each trial to 10,000 steps to avoid infinite loop bugs introduced by mutation.

For each mutation, we generate multiple trials by enabling the feature of bug-free path inference and controlling the amount of provided unclear feedbacks to observe their difference. We control the amount of simulated unclear feedbacks

to be 0%, 0.5%, 1%, 5%, and 10% of the trace length. We choose the trials with inference feature enabled and provided 1% unclear feedbacks as **representative trial**s.

A. RQ1: Effectiveness and Efficiency

Table III shows the experiment results on representative trials, including the number of test cases (TC), number of mutation (MU), average trace length (ATL), average clear/unclear feedback number (ACF/AUF), median clear/unclear feedback number (MCF/MUF), and the effective ratio (ER).

TABLE III
EXPERIMENT RESULT

Project	TC	MU	ATL	ACF	AUF	MCF	MUF	ER
Apache Math2.2	374	2103	2310.8	45.9	14.6	15.0	6	92.4%
Apache Lang3.3	471	1008	233.0	8.5	0.5	2.0	0	93.8%
Apache CLI1.3	80	298	818.4	52.9	3.5	2.0	0	91.3%
Total	925	3409	1565.9	35.5	9.4	6	0	92.8%

Table III shows that *Microbat* can find 92.8% of the mutated bugs with our recommendation paradigm. We investigated the failed trials and found that *Microbat* could miss some data dominance relation due to third party library calls. Our implementation does not analyze the third party library, thus some missing the data dominance relation results in *Microbat* failing to recommend data dominator step in such cases.

Table III also shows that, compared to the average trace length of 1565.9 steps, *Microbat* generally requires the developer to provide on average 35.5 clear feedbacks (with on average 9.4 unclear feedbacks) and a median of 6 feedbacks. Figure 4 shows the distribution of required feedback number versus the trace length. In general, our statistic shows that 65% of the representative trials require less than 20 clear feedbacks to locate the bug (the details can be checked at [4]).

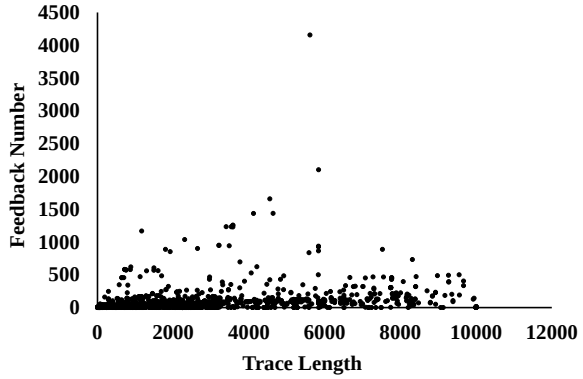

Fig. 4. Feedback Number versus Trace Length

We investigated the trials with a large number of feedbacks, we found that these cases happen when the bug lies in between a dominance relation where dominator is correct and the dominatee is incorrect. In such case, *Microbat* will transfer to *Inspect Detail* state to sequentially inspect the steps. When the path of the dominance relation is long, it causes a great number of correct feedbacks. An extreme case happens in one trial of the test case *testUnstableDerivative()* in *Math* project. The simulated developer provided a wrong-variable-value feedback on the 5611st step and a correct feedback on the 495th step where the variable is defined. However, the mutated bug happens at the 4536th step. After providing a correct-step feedback at 495th step, *Microbat* transits the debugging state to *Inspect Detail* state. The simulated developer then sequentially provided 4042 correct feedbacks until he finally found the mutated bug. We call such case as **long-dominance effect**. We will discuss more about it in Section V-D.

B. RQ2: Contribution of Bug-free Path Inference

We regard the inference feature takes positive effect if it can save feedbacks and negative effect otherwise. Overall, the feature takes positive effect on 514 (i.e., 15.1%) trials and negative effect on 10 (i.e., 0.2%) trials. Among these trials, the feature saves an average of 6.3 (i.e., 10.4%) feedbacks per trial, and a maximum of 110 feedbacks in one trial. More comprehensive details can be checked at [4].

C. RQ3: Impact of Unclear Feedback

a) Impact on Feedback Number and Effective Ratio: Table IV shows that, when the unclear feedback ratio increases, the clear feedback number (both on average and median) increases while the effective ratio almost keeps intact.

TABLE IV
IMPACT ON ER, ACF, AND MCF

Unclear Ratio		0%	0.5%	1%	5%	10%
Feedback Number	ACF	29.0	32.9	35.5	48.9	55.4
	MCF	4.0	5.0	6.0	8.0	10.0
ER		92.7%	92.8%	92.8%	92.4%	92.3%

b) Impact on Inference Feature: Table IV shows that, on those 524 inference-effective trials, the positive effect decreases and negative effect increases with the increase of unclear feedback ratio. Our investigation finds that the reason lies in the randomness of simulated unclear feedbacks. First, randomly provided unclear feedback sometimes makes *Microbat* hard to stably summarize bug-free path pattern. Meanwhile, if the random unclear feedback leads *Microbat* to *Inspect Detail* state when long-dominance effect happens, a large number of correct feedbacks makes the case turn negative. Nevertheless, the majority of trials (430 out of 524) still take positive effect under 10% unclear feedbacks.

TABLE V
IMPACT ON INFERENCE FEATURE

Unclear Ratio		0%	0.5%	1%	5%	10%
Inference Feature	Positive	514	455	436	408	430
	Negative	10	39	64	92	83

D. Discussion

In this work, our collected feedbacks are *partial* specification. The gap between the partial information collected through feedbacks and the real specification causes *Microbat* sometimes require a large number of feedbacks when the long-dominance effect happens. Obviously, there is a trade-off between the effort for developers to provide feedback and the accuracy of step recommendation. In this work, we favour the developers' effort over recommendation

accuracy. Our future work will explore more alternatives of such trade-off.

In summary, we conclude that *Microbat* can detect the majority of our mutated bugs with an acceptable number of feedbacks; the bug-free path inference makes considerable contribution to reduce the feedbacks; and the increase of unclear feedback number impacts little on trial effective rate, but sometimes offsets the effect of bug-free path inference and requires more feedbacks to find the bug.

E. Threat to Validity

The main threat in our simulation experiment lies in that the mutated bugs are still different from the real-world bugs in practice. Nevertheless, Andrew et al. [11] empirically assess the effect of mutation and their result shows that the use of mutation operator yields trustworthy results and seeded faults are harder to detect.

VI. User Study

We conducted a user study to investigate whether our technique can help developers debug in practice. We design the study to answer the following research questions:

- **RQ1**: Whether can *Microbat* help developer debug the program more efficiently in practice?
- **RQ2**: How does developers use *Microbat* in practice?

A. Study Design

In this study, we asked the participants to finish three debugging tasks. We chose *Whyline* [21], [22] as the baseline tool to compare with *Microbat* (A demo of *Whyline* can be checked at [7]). *Whyline* can record the execution trace, allow developers to ask why or why not questions on trace steps (e.g., *why does the variable equal 3?* or *why is this statement executed?*), and answer the questions by showing a relevant source code line. The user study for *Whyline* showed that novice programmers with *Whyline* were twice as fast as expert programmers without it [21]. The main difference between *Microbat* and *Whyline* lies in that *Microbat* (1) allows richer types of feedback such as correct and unclear, and (2) supports more sophisticated inference for suspicious steps such as bug-free path inference and clarity guidance.

We recruited 16 graduate students or research staffs as participants in this study from two universities in Singapore. We surveyed all the participants and divided them into two equivalent groups based on their programming experience. Participants were matched in pairs by their capability and each pair was randomly allocated to the experimental or control group. The experimental group used *Microbat* and the control group used *Whyline* to accomplish the same tasks in the study. We gave tutorials of both tools three hours before the study and asked the participants to familiarize themselves with an exercise using their respective tool.

We chose three bugs as debugging tasks which were once used as the debugging problems in the final exam of software testing course in Nanjing University (ranking top 5 in China) in May, 2016. The source code of debugging tasks can be checked at [2]. Table VI shows brief description of the tasks. Despite these programs consist of only 70∼145 lines of code, we regard them as non-trial because (1) the code involves complicated logic (the statistic of the final exam in Nanjing University shows that 15.2% of the students failed to locate the bugs); (2) the participants had to spend some effort to understand the code details as they were unfamiliar with the code in advance.

TABLE VI
Task Description

Task	Name	LOC	General Description	Bug Reason
#1	Simple Calculator	145	Given a valid algorithmic expression, parse it into correct value.	Some negative signs are parsed to minus sign.
#2	Longest Consecutive Sequence	70	Given an integer array, find the size of its largest subset which consists of consecutive elements.	Duplicated elements in the set are not considered.
#3	Search In Rotated Sorted Array	85	Given a sorted array is rotated at some unknown pivot, find an element in O(lg(n)) time.	Some boundary checking is wrong.

Before the study, we explained the general idea of how each buggy program works to reduce their effort for program comprehension. In the study, the participants were given a failed test case and required to find the bug with respective debugger. Since bug fixing is out of the capability of both tools, we did not require them to fix the bug. Instead, they should write down the detailed reason why the buggy programs fail with the given test case. In order to conduct post-mortem analysis on participants' behaviors on *Microbat*, we instrumented the tool to record the usage frequency of each feature. In addition, we required the participants in both groups to run a full-screen recorder throughout the experiment session.

B. Results: Debugging Efficiency (RQ1)

In this study, all the participants in both groups successfully figured out why the bugs happen. Therefore, we evaluated the task completion time as their performance.

TABLE VII
Performance of both Group (min)

Par\Task	Task #1		Task #2		Task #3	
P1/P9	5.7	15.5	8.1	18.0	10.0	12.1
P2/P10	10.0	10.2	9.8	25.5	7.8	25.4
P3/P11	9.7	25.5	4.2	10.1	7.0	19.5
P4/P12	12.1	36.2	9.5	32.7	10.5	25.1
P5/P13	20.4	35.2	7.3	35.1	13.5	35.3
P6/P14	16.0	42.3	11.4	34.8	6.5	13.0
P7/P15	12.2	27.2	10.7	47.4	11.2	22.5
P8/P16	33.2	48.6	22.9	39.5	12.6	43.4
Avg	14.9	30.1	10.5	30.4	9.9	24.5
p-value	0.012		0.012		0.012	

Table VII shows the time used by the participants in both groups to accomplish three debugging tasks. In Table VII, P1∼P8 are the participants in *Microbat* group and P9∼P18 are the ones in *Whyline* group. Overall, *Microbat* group accomplished the tasks in shorter time compared with *Whyline* group. We introduced the following null and alternative hypotheses to evaluate how different the performance of both groups is.

- **H0:** The primary null hypothesis is that there is no significant performance difference between the two groups.
- **H1:** An alternative hypothesis to H0 is that there is significant performance difference between the two groups.

We used Wilcoxon's matched-pairs signed-ranked tests to evaluate the null hypothesis H0 in terms of the completion time on each task at a 0.05 level of significance. Table VII shows that all the p-values are less than 0.05, thus we reject the null hypothesis for the completion time of all the three tasks and conclude that there is a significant performance difference between the two groups. In addition, Table VII shows that *Microbat* group completed those tasks in shorter average time. Hence, we conclude that *Microbat* group accomplished all the three debugging tasks in significant shorter time.

C. Results: User Behavior (RQ2)

Table VIII shows the frequency of each feature of *Microbat* is used in each debugging task. The features include four types of feedback provided by the participants, frequency of bug-free path inference taking effect (noted by "inference"), participants' manual clicks on the trace steps (noted by "exploration on trace"), and undoing certain feedback on a trace step (noted by "undo").

TABLE VIII
USER BEHAVIOR OF *Microbat* GROUP

	Task\Par	P1	P2	P3	P4	P5	P6	P7	P8	Avg
wrong-variable-value feedback	#1	12	12	18	16	15	14	31	17	17.13
	#2	19	15	6	7	18	8	24	15	14.00
	#3	9	21	8	3	5	6	10	9	8.88
wrong-path feedback	#1	0	0	0	0	0	0	0	0	0.75
	#2	0	0	0	0	0	0	0	0	0.00
	#3	1	2	1	0	1	2	2	1	1.25
correct feedback	#1	0	7	1	0	0	0	0	0	4.13
	#2	5	3	4	3	19	2	1	7	5.50
	#3	10	15	12	0	0	3	8	2	5.00
unclear feedback	#1	0	1	0	0	0	0	0	0	0.13
	#2	0	0	0	0	0	2	0	0	0.25
	#3	0	0	0	0	0	0	0	0	0.00
inference	#1	1	6	3	0	2	2	4	2	2.50
	#2	6	2	5	1	9	3	8	8	5.25
	#3	0	0	0	0	0	2	0	0	0.25
exploration on trace	#1	4	5	1	3	2	2	15	57	11.13
	#2	15	21	1	23	7	13	1	8	11.13
	#3	15	32	3	36	1	8	1	35	16.38
undo	#1	0	1	2	0	2	0	11	0	2.0
	#2	0	0	0	0	28	7	15	0	6.25
	#3	0	14	0	0	0	2	3	0	2.38

Overall, we have the following observations. First, wrong-variable-value feedback is the most frequent among all four types of feedback. Second, the amount of unclear feedback is fairly low (only P2 and P6 provided one such feedback). Third, the bug-free path inference took effect for many participants in Task#1/Task#2 but not in Task#3. Fourth, the participants also actively explored additional steps other than those recommended ones (average 11.13 times for Task#1 and Task#2, and 16.38 times for Task#3). Last but not least, some participants would make wrong feedbacks so that they need to apply "undo" to correct their previous mistakes.

D. Analysis on Study Results

We analyzed the recorded videos and interviewed some participants to uncover the reason of the results showed in Table VII and Table VIII.

1) Why Microbat group debug faster?: We found that the reason lies in the bug-free path inference and the more explicit context information provided in *Microbat*.

First, the bug-free path inference reduced the number of inspected steps. For example, the trace in Task#1 consists of 864 steps. It involves 6 loop iterations, each of which further involves an average of 8 nested loop iterations. With the inference feature, *Microbat* can skip a large number of less suspicious iterations and recommend a more relevant one. In contrast, the generated questions in *Whyline* are only relevant to data and control dominance relations. When the iteration number increases, the participants in *Whyline* group usually need to manually go through a large number of iterations, which takes considerable time and effort.

Second, the more explicit context information provided by *Microbat* speeds up the debugging process. Most participants started debugging in a backward manner. After a step (or a source code line) is recommend by *Microbat* or *Whyline*, they usually need to grasp the *context* of the recommended step. Otherwise, they would easily get lost in the trace and fail to provide a clear feedback (for *Microbat*) or select a correct question (for *Whyline*). *Microbat* organizes the trace steps in a visualized hierarchical way so that participants can explore the tree structure to keep track of which iteration or which method invocation a step belongs to. For example, the buggy program in Task#2 adopts a greedy strategy to search the size of the largest consecutive subset (see specification at [1]). In each iteration of search, the participants should be aware of how many consecutive subsets had been formed. The participants in *Microbat* group can retrieve such contextual information by simply exploring the parent or children of a step and checking relevant variables in program state. In contrast, the participants in *Whyline* group had to retrieve such information by iteratively checking the predecessors and successors of the recommended step in a stepwise manner, which would usually break their *mental flow* and affect the debugging efficiency.

2) Why few unclear feedback were provided?: Our interview with some participants in *Microbat* shows that they often *cannot* make a decisive wrong-variable-value, wrong-path, or correct feedback. However, they did not prefer to provide the unclear feedback in *Microbat* either. We found the reason as follows. Despite participants would get unclear about certain step during debugging, they usually knew how to explore the trace to make it clear. Since the participants were the first time to use *Microbat*, they had not built much confidence in the tool. In addition, they needed to keep their debugging mental flow when inspecting the trace. Hence, when they had a clue to understand a step, their choice is conservative, i.e., manually exploring the trace rather than relying on the tool's recommendation. It also explains why the frequency of exploration on trace in Table VIII is high (average 11.13 for Task#1/Task#2 and 16.38 for Task#3).

3) Why bug-free path inference took effect differently on tasks?: The recorded video shows that some participants in *Microbat* group adopt different strategies when accomplishing Task#1/Task#2 and Task#3. When accomplishing Task#1/Task#2, they located the bug in a backward manner as we expected. Therefore, the wrong-variable-value feedback was provided more frequently (average 17.13/14.00 in Task#1/Task#2) and the bug-free path inference can take effect (average 2.50/5.25 in Task#1/Task#2). However, some

participants located the bug in Task#3 in a forward manner rather than backward manner. The reason is as follows. The buggy program in Task #3 adopts a binary search strategy to find an element in a rotated sorted array (see specification at [5]). After providing several feedbacks at the end of trace, some participants got no clue of the correct search range on an intermediate step even after checking its contextual steps. Therefore, they decided to start from the very beginning step and explored the trace in a top-down manner, i.e., going through the trace from high-level steps to low-level steps, and finally locate the bug. It also explains why the frequency of exploration on trace increases (average 16.38 times) in Task#3. In contrast, other participants took some time to summarize the loop invariants, based on which they provided correct feedbacks on intermediate steps so that they can debug in a backward manner as in Task#1/Task#2.

In summary, the user study shows that *Microbat* outperforms the state-of-the-art tool in debugging efficiency. Nevertheless, it also reveals possible useful improvement of our tool, such as supporting loop invariant summarization and wrong user feedback detection. We will pursue these improvements in our future work.

E. Threats to Validity

There are mainly three threats in our user study. First, our recruited participants were not very familiar with the buggy programs, which may potentially incur their spending lots of effort on understanding the code. In order to mitigate this threat, we describe the general idea of how each program works with one given test case. Second, we assume that the experimental group is equivalent with the control group in their capability and experience, which may be threatened by the actual differences between the two groups. To mitigate this threat, we allocated participants with comparable capability and experience into different groups based on our pre-study survey. Third, we used three debugging tasks in this study, which may not be representative for all the cases. Further studies are required to generalize our findings in large-scale industrial systems.

VII. Related Work

Spectrum-based fault localization techniques [9], [10], [30], [32], [33] are widely used to locate bugs in terms of lines of source code. These techniques compare the code coverage of passed and failed test cases to provide the most suspiciousness code to developers. Reps et al. [30] first proposed the idea of spectrum-based fault localization, and the researchers keep improving technique over the years. Renieris et al. [29] proposed a simple spectrum-based technique and implemented a tool called WHITHER. Wang et al. [33] improved the effect of fault localization by addressing the coincident correctness problem. Abreu et al. [9] further proposed an approach to detecting multiple faults by combining spectra and model-based diagnosis. An overview of spectrum-based techniques can be checked in [10].

Similar to spectrum-based techniques, delta-debugging [15], [18], [25], [28], [36], [37], [38] also requires a set of passed and failed test cases. However, these techniques compare the difference of test cases in more aspects than code coverage, such as test input [38], program states [15], [37], path constraints [28], etc. Zeller et al. [36] first proposed the idea of delta debugging and used it in regression testing. Then, they exploited the technique to simplify test case [38], isolate bug-causing variable [15], [37], and etc. Followed by their work, Misherghi et al. [25] proposed an improvement to refine the result of delta-debugging. With similar ideology, Qi et al. [28] and Yi et al. [34] referenced the "delta" in versions of regression testing to facilitate fault localization.

Different from these techniques, our approach assumes no comparison with a passed test case. In addition, our approach locates the bug in finer grain in terms of buggy step instead of line of source code.

Similar to our approach, a lot of techniques [12], [20], [21], [22], [26], [27], [31], [35] leverage program execution trace for the fault localization. Ressia et al. [31] proposed an object centric debugging approach which facilitates tracking a specific object instance during the execution. Yuan et al. [35] proposed a tool called SherLog which infers the reason of program failure by combining recorded program log and source code. Pohier et al. [26], [27] proposed omniscient debugger which records the whole execution trace of a debugged program and enables user to explore it. Ko et al. [20], [21], [22] built a tool called *Whyline* which provides an interface to allow user to select some questions on program output and the tool can find possible explanation by dynamic slicing on the recorded program trace. Our approach is different from these works in that we (1) allow richer types of feedbacks and (2) support more sophisticated inference for suspicious steps.

Additionally, Lo et al. [17] proposed a feedback-based approach to improve spectrum-based fault localization approach with user feedback on recommended suspicious program statements. In contrast, our approach allows developers to provide feedback on execution steps to localize the fault.

VIII. Conclusion and Future Work

In this paper, we propose a feedback-based debugging approach which incorporates developers' feedback on recorded program execution steps. By inferring and approximating the bug-free paths, our approach aims to iteratively guide them to localize the root-cause step. Our simulation experiment shows that our approach can effectively and efficiently locate the buggy step, and our case study indicates that our tool *Microbat* is practical to facilitate the debugging tasks. In our future work, we would pursuit new features such as solving long-dominance effect, summarizing loop invariant on trace steps, and detecting mistaken feedback on *Microbat*.

IX. Acknowledgement

We thank the anonymous reviewers for their valuable comments and suggestions. This research has been partially supported by the National Research Foundation, Singapore (No. NRF2015NCR-NCR003-003), and the National Science Foundation of China (No. 61572349, 61272106).

REFERENCES

[1] Longest consecutive sequence. https://leetcode.com/problems/longest-consecutive-sequence/. Accessed Augest 20, 2016.

[2] Microbat experiment code. https://github.com/llmhyy/microbat_experiment. Accessed Augest 20, 2016.

[3] Microbat github website. https://github.com/llmhyy/microbat. Accessed Feb 2, 2017.

[4] Microbat webpage. http://linyun.info/microbat/. Accessed Feb 2, 2017.

[5] Search in roated sorted array. https://leetcode.com/problems/search-in-rotated-sorted-array/. Accessed Augest 20, 2016.

[6] Simple calculator problem. https://leetcode.com/problems/basic-calculator/. Accessed Augest 20, 2016.

[7] Whyline video. https://www.youtube.com/watch?v=3L4MK2dG_6k. Accessed Augest 20, 2016.

[8] *Biological Sequences and the Exact String Matching Problem*, pages 43–63. 2006.

[9] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99, 2009.

[10] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780 – 1792, 2009.

[11] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering*, pages 402–411, 2005.

[12] E. T. Barr and M. Marron. Tardis: Affordable time-travel debugging in managed runtimes. Technical report, 2014.

[13] B. Beizer. *Software Testing Techniques (2Nd Ed.)*. 1990.

[14] M. Beller, G. Gousios, A. Panichella, and A. Zaidman. When, how, and why developers (do not) test in their ides. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 179–190, 2015.

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

[16] J. S. Collofello and S. N. Woodfield. Evaluating the effectiveness of reliability-assurance techniques. *Journal of System and Software.*, 9(3):191–195, 1989.

[17] L. Gong, D. Lo, L. Jiang, and H. Zhang. Interactive fault localization leveraging simple user feedback. In *IEEE International Conference on Software Maintenance*, pages 67–76, 2012.

[18] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 263–272, 2005.

[19] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343.

[20] A. J. Ko and B. A. Myers. Designing the whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 151–158, 2004.

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

[22] A. J. Ko and B. A. Myers. Finding causes of program output with the java whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1569–1578, 2009.

[23] Y. Lin. Technique report: A tree-based approach to identifying and explaining regression bug. http://linyun.info/microbat/report.pdf, 2017. Accessed Jan 31, 2017.

[24] T. Liu, C. Curtsinger, and E. D. Berger. Doubletake: Evidence-based dynamic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, 2016. accepted.

[25] G. Misherghi and Z. Su. Hdd: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering*, pages 142–151, 2006.

[26] G. Pothier and . Tanter. Back to the future: Omniscient debugging. *IEEE Software*, 26(6):78–85, 2009.

[27] G. Pothier and E. Tanter. Summarized trace indexing and querying for scalable back-in-time debugging. In *Proceedings of the 25th European Conference on Object-oriented Programming*, pages 558–582, 2011.

[28] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: An approach for debugging evolving programs. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 33–42, 2009.

[29] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of International Conference on Automated Software Engineering*, pages 30–39, 2003.

[30] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European SOFTWARE ENGINEERING Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 432–449, 1997.

[31] J. Ressia, A. Bergel, and O. Nierstrasz. Object-centric debugging. In *Proceedings of the 34th International Conference on Software Engineering*, pages 485–495, 2012.

[32] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *Proceedings of 31st International Conference on Software Engineering*, pages 56–66, 2009.

[33] X. Wang, S. C. Cheung, W. K. Chan, and Z. Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *Proceedings of the 31st International Conference on Software Engineering*, pages 45–55, 2009.

[34] Q. Yi, Z. Yang, J. Liu, C. Zhao, and C. Wang. A synergistic analysis method for explaining failed regression tests. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, pages 257–267, 2015.

[35] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, pages 143–154, 2010.

[36] A. Zeller. Yesterday, my program worked. today, it does not. why? In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 253–267, 1999.

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

[38] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transaction on Software Engineering*, 28(2):183–200, 2002.