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 tool-supported and 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 reason the feedback patterns, which helps reduce required feedbacks to expedite the debugging process. We conduct an experiment to evaluate our approach with simulated feedbacks on 2299 mutated bugs across 3 open source projects. The results show that our feedback-based approach can detect 95.7% of the bugs and 60% of 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 result shows 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 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], [29], [31], [32], delta-debugging [15], [18], [24], [27], [35], and dynamic trace recording [12], [21], [23], [25], [26], [30], [34]. 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 [35], [37] and isolate root cause variable of bug [36]. However, in the process of development, developers usually lack sufficient passed test cases [14] to apply these techniques. Some dynamic trace recording techniques, such as omniscient debugging [12], [25], 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. To the best of our knowledge, when developers lack sufficient test cases (which is common in practice), existing state-of-the-art techniques are either unapplicable or insufficient to reduce the debugging effort.

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 first buggy step in the program execution. Our rationale lies in the observation that, in many cases, the specification of detailed code 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 and control dominance relation) among the steps. On each trace step, we allow the developers to provide various types of light-weight feedback (i.e., correct, wrong variable value, wrong path, and unclear). Our approach then takes the feedback and recommends suspicious step based on causality relation among trace steps. After collecting a number of feedbacks, our approach begins to learn and reason the feedback patterns, which helps reduce the number of feedbacks to expedite the debugging process. This iterative process starts with a user feedback on an fault-revealing trace step and finishes when the root cause buggy step in the trace is recommended.

We implement our approach as an Eclipse plugin, *Microbat* (A demo video of *Microbat* can be checked at [5]). We first conduct a simulation experiment by using *Microbat* to find 2299 mutated bugs with simulated feedbacks on three open source projects. The results show that *Microbat* is able to detect 95.6% of the mutated bugs and 60% of detected bugs requires 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 to iteratively guide developers to locate the bug by recommending suspicious steps; 2) We develop a proof-of-concept tool, *Microbat*, 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.

TABLE I
DEBUGGING CODE EXAMPLE

```
0  class AlgorithmeticExpressionParser{
1    public int calculate(String expr){
2      int bracketStartIndex = -1;
3      while(containsBracket(expr)){
4        char[] list = expr.toCharArray();
5        for(int i=0; i<list.length; i++){
6          if(ch == '(')
7            bracketStartIndex = i;
8          else if(ch == ')'){
9            String simpleExpr = expr.substring(bracketStartIndex
                 +1, i);
10           int value = evaluateSimpleExpr(expr);
11           String beforeExpr = expr.substring(0,
                   bracketStartIndex);
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   }
22   ...
23 }
```

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 the effectiveness of our approach with a simulation experiment. Section VI shows our user study of *Microbat* on real-world bugs. Section VII reviews related work. Section VIII concludes the paper.

II. MOTIVATING EXAMPLE

Table I shows our motivating example, which is adopted from a code training website [7]. Given a valid algorithmic expression consisting of integers, brackets, or plus/minus signs, e.g., "1-((1+2)-1)", this program should compute the 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 suffer from answering following questions, which takes lots of manual debugging efforts.

(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 or her feedback on this step, such as which variable in this step is with wrong value, or whether this step should be in the execution, then *Microbat* is able to recommend certain step responsible for its cause.

In our example, the developer can easily observe the program state in the step running into line 20 in Table I, in which 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* can recommend a step by (1) simple causality analysis, (2) loop pattern analysis, 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 *expr* variable (data dominance), i.e., the latest step running into line 19, and highlights its corresponding source code line. On this step into line 19, 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* (the same to the selected read variable *result* in line 20) 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.

Loop Pattern Analysis. In order to reduce the inspection effort, *Microbat* leverages *loop pattern analysis*. 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.

Figure 1 shows that there are 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, in this case, he gives wrong-variable-value feedback on the 9th and 8th iteration sequentially. *Microbat* then is able to summarize some possible bug-free path pattern along the loop trace. In this case, based on the pattern, *Microbat* can skip the 6th and 7th iteration, and recommend a step in the 5th iteration. The rationale behind is as follows. First of all, *Microbat* is able to summarize that the cases in 6th–9th iteration are similar in that they all parse the addition

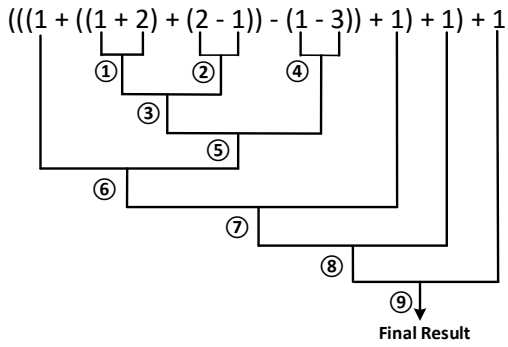

Fig. 1. Execution of Example Program

of two positive integers, for example "5+1" in 9th iteration, "4+1" in the 8th iteration. Based on developer's feedback, the bug does not occur in the 8th or 9th iteration, therefore, *Microbat abduct*s that the bug may not happen in the 6th or 7th iteration either. However, *Microbat* has never encountered the case happening in the 5th iteration, in which a positive integer (4) minus a negative integer (-2). Thus, *Microbat* stops in the 5th for developer's further feedback.

In this case, the developer inspects the 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, with simple causality analysis, the developer can select the returned variable from *evaluateSimpleExpr()* method so that *Microbat* can further do recommendation 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 more "abstract" step such as method invocation or loop head. In our case, suppose that the developer cannot make sure the correctness of a certain step, e.g., a step running inside the method body of *evaluateSimpleExpr()* method (see line 10), he can provide the *unclear* feedback so that *Microbat* will recommend the latest step running into a "method invocation" step in line 10 or a "loop head" step inside the *evaluateSimpleExpr()* method. If the follow-up feedback is unclear again, *Microbat* will recommend an even more abstract step; if the follow-up feedback is wrong-variable-value or wrong-path, *Microbat* will leverage the simple causality analysis; if the follow-up feedback is correct-step, *Microbat* will recommend a step close to the most recent unclear step. The recommendation for more "clear" steps will not stop until the developer makes the feedback showing that he has understood his unclear step.

III. Approach

Given a trace of steps, our approach aims to find the first step which deviates from developer's expectation and eventually causes the observable fault after program execution. We call such a step as the **ringleader** step.

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 or 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.

We construct the trace model by two runs of the buggy program. In the first run, we collect some preliminary code information. We start by collecting the source code involved in this execution to get a narrowed scope for analysis. Then, for the code in scope, we extract the read and written *static* variables of each source code line and build its CFG (control flow graph). In the second run, we construct trace model by building the dominance relation among executed steps. During the execution, we construct a step each time when a source code line is reached. For each step, we dynamically retrieve the runtime value of the read and written static variables in the corresponding line. A variable attached with runtime value is a **runtime variable**, and all the variables in our trace model are runtime variables. In contrast to static variables, a runtime variable will be defined only once, i.e., the time when it is written. For example, each time when line 15 of Table I is reached in the execution, there will be a new runtime *expr* variable defined. A static variable can correspond to many runtime variables. In this paper, if two runtime variables var_{r1} and var_{r2} origin from the same static variable, they are called **homologous variables**. Finally, we construct control dominance relations based on the built CFG, and data dominance relations based on the defined and used runtime variables among steps.

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 leat one variable in this step is of wrong value. Once a developer provides such a 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.

Figure 2 depicts our mechanism for recommending suspicious steps. During the debugging, based on historical feedbacks of the developer, we infer what **debugging state** he is in. The states incorporate debugging contextual information for us to recommend different steps even with the same provided feedback type. Figure 2 shows a state machine consisting of five debugging states and their transitions interpreted from feedbacks. A rounded rectangle represents a state while an edge represents a transition. In addition, a state with dashed line represents a composite state, which further consists of a number of sub-states.

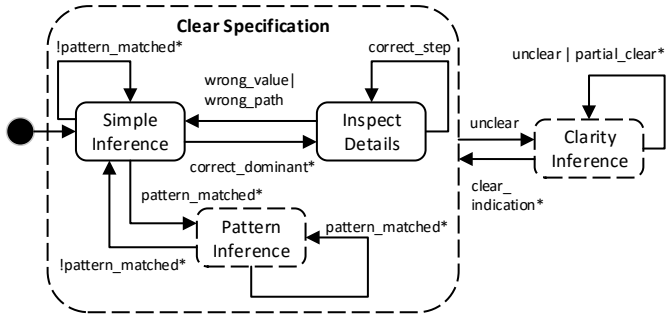

Fig. 2. Overall Recommendation State Machine

Composite States: The composite states in Figure 2 are *Clear Specification* state and *Clarity Inference* state. When the developer can decide whether a step is correct, he is in *Clear Specification* state, otherwise, he is in *Clarity Inference* state. In *Clear Specification* state, our approach infers a suspicious step and recommends it for inspection. In *Clarity Inference* state, our approach recommends a more semantically abstract step to guide the developer to understand the step (see Section III-B3).

Sub-States: The composite *Clear Specification* state consists of three sub-states, i.e., *Simple Inference* state, *Pattern Inference* state, and *Inspect Details* state. The *Simple Inference* state is the debugging state in which we only leverage the basic dominance relations for recommendation. During the *Simple Inference* state, we actively monitor the user feedbacks for pattern summarization. Once certain feedback is matched with some summarized pattern, we transit to the *Pattern Inference* state in which we recommend the steps with pattern in a more intelligently way (see Section III-B1). In addition, when our approach detects that the ringleader step could lie in between a dominator step and its dominatee step, we transit to the *Inspect Details* state in which we recommend the suspicious steps in finer grain (see Section III-B2).

Transitions: Once the developer provides a feedback, we interpret it into an event to trigger a state transition. Some interpretations are straight-forward, which are the feedbacks themselves, e.g., the *correct_step* event and the *unclear* event. However, some other interpretations require analyzing developer's historical feedbacks and the information incorporated in trace model, e.g., the *pattern_matched* event and the *clear_indication* event. These events are attached with an asteroid in Figure 2, which will be explained in Section III-B1, Section III-B3, and Section III-B2.

In the following, we explain *Pattern Inference* state, *Inspect Details* state, and *Clarity Inference* state respectively.

1) Pattern Inference: Our rationale lies as follows. Loop is usually the main cause of a long execution trace, nevertheless, the cases handled in loop are often enumerable. In our motivating example, when parsing the expression "(((1+((1+2)+(2-1))-(1-3))+1)+1)+1", the program goes through 9 iterations (see Figure 1) in the same loop (line 3–18 in Table I). However, the iterations can be summarized into 3 **loop cases**: 1) one positive integer adds a positive integer, e.g., the 7th–9th iterations; 2) one positive integer subtracts a positive integer, e.g., the 2nd iteration; 3) one positive integer subtracts a negative integer, e.g., the 5th iteration. The iterations falling into the same loop case can be regarded as "equivalent". Based on such an observation, we try to summarize equivalent loop iterations, and make an *abduction* that, once an iteration of a loop case can be inferred as bug-free from the developer's feedbacks, all the iterations falling into the same loop case can be skipped for inspection. By this means, we can skip a large number of steps to speed up the debugging process. As the abduction can be too aggressive in some cases, we adopt an adjusting mechanism for complement (see Section III-B1b).

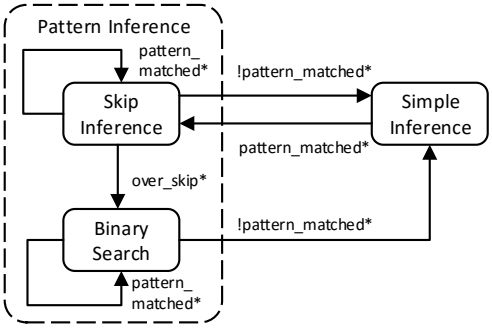

Fig. 3. Pattern Inference State Machine

Figure 3 shows the mechanism of how the *Pattern Inference* state works. The composite *Pattern Inference* state consists of two sub-states, i.e., *Skip Inference* and *Binary Search*.

When the developer is debugging in *Simple Inference* state, we record all the feedbacks and summarize some step paths as **path pattern** (see Section III-B1a). Each path pattern represents a loop case inferred as bug-free. Once the developer's feedback indicates that he is going through a path which conforms to some summarized path pattern, a *pattern_matched* event is triggered, which transits the debugging state from *Simple Inference* to *Skip Inference*.

In the *Skip Inference* state, our approach recommends a suspicious step with the matched pattern (see Section III-B1) by abduction. Given developer's new feedback on the recommended step, (1) if it triggers a *pattern_matched* event again, we keep staying at *Skip Inference* state; (2) if it indicates the abduction is too aggressive (see Section III-B1c), the *over_skip* event it triggered so that the state is transited to *Binary Search*; (3) otherwise, it triggers a *!pattern_matched* event so that the state is transited to *Simple Inference*.

In the *Binary Search* state, we adopt a binary search strategy

in all the skipped iterations in *Skip Inference* state. Based on developer's feedback, we infer whether he is still exploring the steps relevant to the pattern (see Section III-B1c). If he does, a *pattern_matched* event is triggered, which keeps the debugging state in *Binary Search* state. Otherwise, a *!pattern_matched* event is triggered, which transits the debugging state back to *Simple Inference* state. Next, we will explain the details of pattern extraction, step skipping, and binary search.

a) Pattern Extraction: The pattern extraction technique aims to leverage developer's feedback to identify some step path which is *potentially* bug-free. Given a step $step$ on trace, if one of its read variables is marked as having wrong value by the developer, then we call this step as an **attributed step**. An attributed step means that its incorrect program state is caused by some step executed before. In contrast, if a step has been checked by the developer and all its read and written variables are not marked as wrong value, then we call this step as a **clean step**. Given a step path on the trace, we call it as a **clean path** if it satisfies that both its start step and end step are attributed steps or clean steps. A clean path is regarded as a path free of bug.

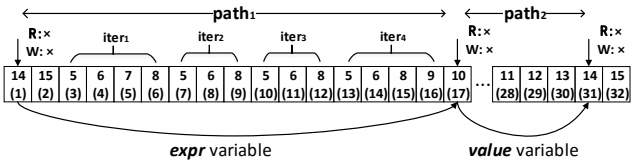

Fig. 4. Path Example

Figure 4 shows a part of trace when the buggy program is in the 8th iteration (see Figure 1), in which the *expr* variable has the value "(4+1)+1". 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 (See definition in Section III-A). Suppose the developer provided his wrong-variable-value sequentially in the 31th step (line 14 with *value* variable) and the 17th step (line 10 with *expr* variable), then we have a clean path $< step_{17}, step_{31} >$.

Then, for each clean path, we merge its consecutive repetitive step sequence in loop iterations. For example, the consecutive step sequence line 5, 6, and 8 in the second and third iterations are repetitive in Figure 4. Therefore, one duplicated sequence will be removed so we have a *merged path* as "14, 15, 5, 6, 7, 8, 5, 6, 8, 5, 6, 8, 9, 11". Note that the fourth iteration with sequence 5, 6, 8, 9 will not be merged despite it shares 5, 6, 8 with the second/third iteration. If a path sequence contains nested loop iterations, the merge process starts from the most nested iterations to the most outside ones. We call the merged path of a clean path as its **pattern key**. All the clean paths sharing the same pattern key fall into the same **path pattern**. In addition, the first clean path used to generate a path pattern is called its **label path**, and the variable marked as wrong on the end step of label path is called its **cause variable**. We consider all instances of a path pattern equivalent, i.e., semantically share the same loop case. In our motivating example, the loop cases, such as the addition of two positive integer and the substraction of one positive integer and one negative integer, are distinguished by this means.

b) Step Skipping: Given a step $step_{cur}$ with a wrong-variable-value feedback on variable var, we can have its to-be-recommended data dominator $step_{sus}$. Then we check whether one of $step_{sus}$'s data dominators $step'_{sus}$ can (1) form a path with $step_{sus}$, i.e., $< step'_{sus}, step_{sus} >$, to share the same pattern key with one of the summarized path patterns $patt$ and (2) the attributed variable of the dominance relation is homologous to the cause variable of $patt$'s label path. If $step'_{sus}$ satisfies both conditions, we skip $step_{sus}$ and consider recommending $step'_{sus}$ instead.

When such a case happens, we first find the label path of $patt$, let its cause variable be var_{cau}. Then, we find the homologous variable var_{homo} of var_{cau} in the read variables of $step_{sus}$. We take var_{homo} as a wrong variable by default (i.e., abduction) and simulatively provide a wrong-variable-value feedback on $step_{sus}$ so that a new suspicious step $step'_{sus}$ can be reached based on dominance relation. Next, we will check the data dominators of $step'_{sus}$ to see whether a new clean path can be formed to match recorded path pattern once again. If it does, we skip $step'_{sus}$ again in the same vein, otherwise, skip process stop and the $step'_{sus}$ is recommended.

For the example in Figure 4, suppose we have *already* summarized a path pattern $patt$ = "14, 15, 5, 6, 7, 8, 5, 6, 8, 5, 6, 8, 9, 10" and the cause variable of its label path is a runtime *expr* variable (see line 10 in Table I). Let the $step_{cur}$ be the 31th step (line 14) in which the *value* variable is marked as wrong, then the 17th step (line 10) will be considered as $step_{sus}$ based on dominance relation. Clearly, we can see that the 1st step is a data dominator of the 17th step which can form a possible clean path matching $patt$. In addition, the attributed variable of the dominance relation is homologous to the cause variable *expr*. Therefore, we can skip the 17th step and consider the 1st step in this case. The process continues until we cannot find a data dominator step to form a possible clean path matching summarized pattern once again.

c) Binary Search: As mentioned before, our skipping strategy is based on an abduction that a path is bug-free (so it is skipped) if its equivalent path has been indicated as bug-free. However, such an abduction can sometimes be over aggressive, therefore, it is possible for us to over-skip some steps. We can find the over-skip case happens when the developer provide a correct-step feedback on the step recommended in *Skip Inference* state. It is because that it indicates that the program state in one iteration is correct while it turns to be incorrect in a certain follow-up iteration. Therefore, the ringleader step should lies in between.

When such case happens, we adopt a binary search strategy among all the skipped steps. Given a recommended step $step$ which is one of the skipped step, we keep the binary search if the developer (1) provides a correct-step feedback, which means that we still over-skipped some steps and the ringleader step happens after $step$, or (2) a wrong-variable-value feedback on $step$ with the variable homologous to our abducted cause variable in the process of skipping, which

means that we still need skip some steps and the ringleader step happens before $step$. Otherwise, we return to *Simple Inference* state.

2) Inspect Details: The *Inspect Details* state is used to recommend the suspicious step in between a dominance relation. Given a path corresponding to a dominance relation in which the start step is the dominator and the end step is dominatee, the debugging state is transited to *Inspect Details* state if the end step is provided as a wrong-variable-value feedback and the start step is provided as a correct-step feedback. When such a case happens, we will sequentially recommend the step in between the start step and the end step if the developer continually provides the correct-step feedback. If the developer provides wrong-variable-value or wrong-path feedback, the debugging state will be transited to *Simple Inference*.

3) Clarity Inference: Figure 5 shows how *Clarity Inference* state works. Once the developer cannot decide the correctness of a step, he can provide a unclear feedback, which triggers an *unclear* event. Then we back up the debugging context and transit the debugging state to *Clarity Inference*. The aim of *Clarity Inference* is to guide the developer better understand the unclear step so that he can resume where he gets lost. The composite *Clarity Inference* state consists of two sub-states, i.e., *Unclear* state and *Partial Clear* state. A provided unclear feedback triggers a *unclear* event to transit any state to *Unclear* state, recommending a semantically more abstract step accordingly. If the developer provides a wrong-variable-value or wrong-path feedback on the recommended step, a *clear_indication* event is triggered, and the debugging state will transit back to *Clear Specification* state (more specifically, the *Simple Inference* sub-state). If the developer provides a correct-step feedback on the recommended step, we will transit to *Partial Clear* state. In *Partial Clear* state, we further recommend a step to guide the developer back to where he provide the unclear feedback. Given the feedback, if there is no more unclear step, a *clear_indication* event is triggered, then we resume the state in *Clear Specification*.

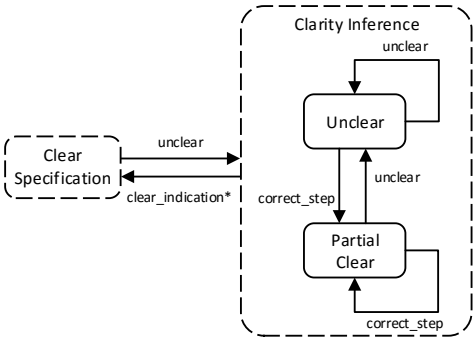

Fig. 5. Clarity Inference State Machine

a) Unclear State: In *Unclear* state, given current step $step\_cur$ marked as unclear, we recommend its contextual parent (see Section III-A) to the developer. If the abstract level of $step\_cur$ is 1, we recommend the nearest step executed before $step\_cur$.

b) Partial Clear State: We maintain a unclear stack st to record the visited unclear steps. Each time a unclear feedback is provided, we push the unclear step into st. Once the developer provides a correct-step feedback on current step $step_{cur}$, the debugging state is transited in *Partial Clear* state. Then we pop the peek unclear step in st and recommend to the developer. Once the stack becomes empty, we resume the sub-state in *Clear Specification* state and its context before entering *Clarity Inference* state.

IV. Tool Support

We implemented our approach as an Eclipse plugin. A screenshot can be checked at *Microbat* Github website [4]. *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 current step.

V. Simulation Experiment

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

- **RQ1**: How effective and efficient can *Microbat* facilitate the debugging process?
- **RQ2**: How is the contribution of loop 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 II). For each passed test case, we mutate its tested code with a standard mutator. The mutator will replace algorithmic operators, logical operators, and number constants, e.g., replacing "+" with "-". In each mutation, only one source code line will be 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, our simulation can reference $trace_c$ to check the correctness of the steps in $trace_m$.

We leverage a dynamic programming algorithm [19] 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 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 a 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**. For each mutation, we generate multiple trials by controlling the feature of loop inference and the amount of provided unclear feedbacks to observe their difference. In this experiment, we control the amount of unclear feedbacks to be 0%, 0.5%, 1%, 5%, and 10% of the trace length. Therefore, each mutation has 2 (enable or disable loop inference) $\times$ 5 (unclear amount) = 10 trials. We choose the trial with loop inference enabled and amount of unclear feedback set to 1% of trace length as the **representative trial** for the mutation. 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 as l_t, if *Microbat* can recommend a suspicious step which runs into $line_m$ within l_t feedbacks, we consider the trial as successful, otherwise, we consider the trial as failed. In addition, we limit the generated trace length for each trial to 10,000 steps to avoid infinite loops introduced by mutation.

A. RQ1: Effectiveness and Efficiency

Table II shows our experiment results, which shows the number of test cases (#TC), number of mutation (#MU), as well as the successful ratio (SR), average trace length (ATL), average feedback number (AFN), and median feedback number (MFN) of representative trials.

TABLE II
EXPERIMENT RESULT

Project Name	#TC	#MU	SR	TL	AFN	MFN
Apache Math2.2	165	1104	97.2%	2304.1	54.0	31
Apache Lang3.3	367	987	95.4%	278.0	23.3	5
Apache CLI1.3	46	208	88.5%	502.7	25.7	1.5
Total	578	2299	95.7%	1318.2	46.1	11

Table II shows that *Microbat* can find 95.7% of the mutated bugs with our recommendation paradigm. We investigated the failed trials and found that the main reason lies in that *Microbat* could miss some data dominance relation due to third party library calls. Our implementation of *Microbat* does not analyze the Java class in third party library, thus some missing the data dominance relation results in *Microbat* failing to recommend data dominator step in some cases.

Table II also shows that *Microbat* generally requires the developer to inspect an average of 46.1 steps and a median of 11 steps, compared to the average trace length of 1318.2 steps. Figure 6 shows the distribution of required feedback number with trace length. In general, our statistic shows that

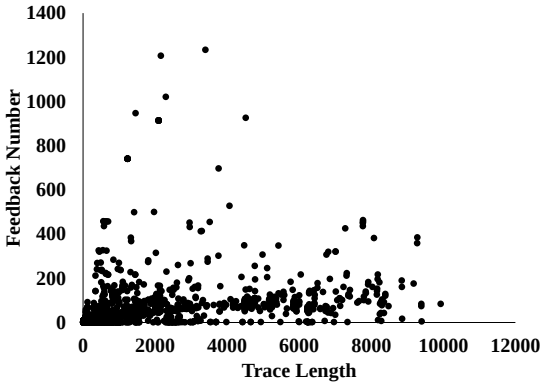

Fig. 6. Feedback Number versus Trace Length

60% of the representative trials require less than 20 feedbacks to locate the bug (the details can be checked at [3]).

We investigated the trials with large number of feedbacks, the result shows that the main reason lies in over estimation of data dominance. Given an observed wrong-value variable, the bug causing its wrong value may happen in the long trace path between data dominator and dominatee. For example, in *CLI* project, the mutation of one trial in the test case *ValueTest#testLongWithArg* incurs an incorrect control flow so that a variable *description* cannot be assigned after its initialization. When our simulated developer observed the incorrect variable value at the 1152th step, his provided wrong-variable-value feedback made *Microbat* recommend the data dominator of the 110th step where the variable is defined. However, the mutated bug happens at the 823th step. After providing a correct-step feedback at 110th step, *Microbat* transits the debugging state from *Simple Inference* state to *Inspect Detail* state. The simulated developer then sequentially provided 714 correct-step feedbacks until he finally finds bug at the 823th step. We will discuss more about the issue in Section V-D.

B. RQ2: Contribution of Loop Inference

Overall, the loop inference takes effect on 424 trials (i.e., 18.44%) in our simulation experiment. Among these trials, the loop inference saves an average of 18.45% feedbacks per trial. More comprehensive details can be checked at [3].

C. RQ3: Impact of Unclear Feedback

Table III shows how unclear feedbacks influence the ratio of trial success and the ratio of required feedback number compared to trace length with loop inference enabled or disabled. We have the following observation from Table III. First, with or without loop inference, the success ratio changes little when unclear feedbacks increase, which means that developer can still locate the bug even he is not clear enough on some steps. Second, the effect of loop inference remains when unclear feedbacks increase, however, the developer may provide more feedbacks.

D. Discussion

In this simulation experiment, the required feedback numbers for some trials are larger than we expected. The

TABLE III
IMPACT OF UNCLEAR FEEDBACK

Unclear Ratio		0%	0.5%	1%	5%	10%
Success Ratio	loop	100%	99.8%	99.5%	99.5%	99.8%
	non-loop	99.8%	99.5%	99.1%	99.5%	98.6%
Feedback Ratio	loop	1.9%	3.4%	4.0%	5.5%	6.2%
	non-loop	2.3%	3.9%	4.7%	6.5%	7.3%

reason for the over estimation lies in that our collected feedbacks are *partial* specification. The gap between the partial information collected through feedbacks and the real specification causes *Microbat* make some over-estimated recommendation. 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. Nevertheless, *Microbat* still only requires less than 20 feedbacks to find 60% of the bugs. In our future work, we 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; our loop inference makes considerable contribution to reduce the feedbacks; and the increase of unclear feedback number impacts little on trial success rate and the effect of loop inference, but 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 debugging in practice. We design the study to answer the following research questions:

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

A. Study Design

In this study, we asked the participants to finish three debugging tasks. We choose *Whyline* [21], [22] as the baseline tool to compare with *Microbat* (A demo of *Whyline* can be checked at [8]). *Whyline* can record the execution trace, allow developers to ask why or why not questions on trace steps (e.g., *why did 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 loop and clarity inference.

TABLE IV
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.

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 on 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 IV 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.

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 run a full-screen recorder throughout 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 V shows the time used by the participants in both group to accomplish three debugging tasks. In Table V, 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.

TABLE V
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	

- **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 V 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 V 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 VI 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, loop inference took effect (noted by "loop 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 VI
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
loop 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 loop 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 V and Table VI.

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

First, the feedback pattern summarized by *Microbat* 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. Based on summarized loop pattern, *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 correct 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 solution to get a step clear, 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 VI is high (average 11.13 for Task#1/Task#2 and 16.38 for Task#3).

3) Why loop 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 loop 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 [6]). 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 more effort on program comprehension rather than debugging. 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], [29], [31], [32] 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. [29] first proposed the idea of spectrum-based fault localization, and the researchers keep improving technique over the years. Renieris et al. [28] proposed a simple spectrum-based technique and implemented a tool called WHITHER. Wang et al. [32] 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], [24], [27], [35], [36], [37] also requires a set of passed and failed test cases. However, these techniques compared the difference of test cases in more aspects than code coverage, such as test input [37], program states [15], [36], path constraints [27], etc. Zeller et al. [35] first proposed the idea of delta debugging and used it in regression testing. Then, they exploited the technique to simplify test case [37], isolate bug-causing variable [15], [36], and etc. Followed by their work, Misherghi et al. [24] proposed an improvement to refine the result of delta-debugging. With similar idealogy, Qi et al. [27] and Yi et al. [33] 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, compared to spectrum-based techniques, 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], [25], [26], [30], [34] leverage program execution trace for the fault localization. Ressia et al. [30] proposed an object centric debugging approach which facilitates tracking a specific object instance during the execution. Yuan et al. [34] proposed a tool called SherLog which infers the reason of program failure by combining recorded program log and source code. Pohier et al. [25], [26] 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 a interface to allow user to select some questions on program output and the tool can find possible explanation by dynamic slicing on 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] also 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 learning and summarizing the patterns of the feedbacks, our approach aims to iteratively guide them to localize the first step introducing the bug. Our automatic experiment shows that our approach can effectively and efficiently locate the buggy step, and our case study

indicates that our tool *Microbat* is of practical use to facilitate the debugging tasks. In our future work, we would pursuit new features such as loop invariant summarization on trace steps and wrong feedback detection on *Microbat*.

IX. Acknowledgement

This research has been supported by the National Research Foundation, Singapore (No. NRF2015NCR-NCR003-003, NRF2014NCR-NCR001-30), 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 experiment result. http://linyun.info/microbat_evaluation. Accessed Augest 20, 2016.
[4] Microbat github website. https://github.com/llmhyy/microbat. Accessed Augest 20, 2016.
[5] Microbat video. https://www.youtube.com/watch?v=jA3131MWuzs. Accessed Augest 20, 2016.
[6] Search in roated sorted array. https://leetcode.com/problems/search-in-rotated-sorted-array/. Accessed Augest 20, 2016.
[7] Simple calculator problem. https://leetcode.com/problems/basic-calculator/. Accessed Augest 20, 2016.
[8] Whyline video. https://www.youtube.com/watch?v=3L4MK2dG_6k. Accessed Augest 20, 2016.
[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] 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.
[24] G. Misherghi and Z. Su. Hdd: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering*, pages 142–151, 2006.
[25] G. Pothier and . Tanter. Back to the future: Omniscient debugging. *IEEE Software*, 26(6):78–85, 2009.
[26] 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.
[27] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: An approach for debugging evolving programs. In *Proceedings of the 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.

[28] 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.

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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