A Feedback-Based Approach for Recommendation of Buggy Step in Program Execution

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 lots of program steps to see whether they deviate from their intended specification in mind, which further involves figuring out where to set breakpoints, infering the program behaviors, and remembering a lot of code details. In this paper, we propose a tool-supported and feedback-based debugging approach to iteratively guide the developers to localize the fault. In our approach, we first record the execution trace of buggy code and allow developer to provide light-weight feedback on the steps of trace (e.g., wrong variable value, wrong path, and unclear). Then we iteratively collect developer's debugging feedbacks on trace steps and recommend suspicious steps on the trace. Moreover, our approach can further summarize their feedback patterns, which can help reduce the number of feedbacks and even infer bug explanation in certain cases. We implement our proof-of-concept tool, *Microbat*, based on our approach. We evaluate *Microbat* by applying it on 1,887 mutated bugs across 3 open source projects. The experiment results show that *Microbat* is able to detect 93.1% of the bugs and each bug requires an average of 3.4 feedbacks. In addition, we conduct a case study by manually applying *Microbat* on 13 real world bugs. The result confirms that our approach is useful and practical.

1. INTRODUCTION

Software debugging is usually regarded as one of the most difficult tasks in software development and maintenance [5, 8]. Given an observable fault, developers usually need to start with the fault-revealing code, have a guess where the bugs are, and manually inspect the code line by line (or sometimes step by step) with their intended specification in mind. When the code logic involves great complex, the process of debugging inevitably demands huge amount of time and mental efforts.

Researchers have proposed a lot of techniques for automation of software debugging (or fault localization), such as spectrum-based fault localization [1, 2, 22, 24, 25], delta-debugging [7, 10, 17, 20, 28, 29, 30], and dynamic trace recording [4, 13, 14, 15, 16, 18, 19, 23, 27]. Both spectrum-based fault localization and delta debugging require comparing at least one passed test case and one failed test case to locate the bug. The former quantifies the suspiciousness of statements by considering the code coverage of passed or failed test cases and reports a ranked list of lines of source code for inspection [1, 2, 10, 20, 22]. The latter analyzes the differences of passed and failed test cases from aspects such as test inputs and running program states, so that it has been used to simplify the test inputs [28, 30], isolate root cause variable [29], etc. However, when encountering a bug in the process of software development, the developer often has only one test input to reproduce the bug [6]. Even they endeavor to construct a set of test cases, these techniques still cannot be applied if all the constructed test cases are failed. On the other hand, some dynamic trace recording technique such as ominous debugging [4, 18] can record the program execution trace for a single run and allow developer to trace back and analyze the faults. However, when the trace length gets long enough (especially caused by loop), the effort of step-by-step trace checking is still overwhelming. To the best of our knowledge, when developers lack correct test case for reference, 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 in nowhere but human mind. Therefore, we leverage light-weight user feedback as "partial specification" to feed the debugger so that it can recommend suspicious steps in an iterative and automatic way. In our approach, given a buggy program, we first build a trace model which records the execution trace and captures causality relations among the steps, then we enable the developers to provide various types of light-weight feedback (e.g., wrong variable value, wrong path, and unclear) on the trace steps. Basically, our approach can recommend suspicious step based on causality relation among trace steps. Moreover, we can also learn and summarize developers' feedbacks to recommend a suspicious step on the trace to speed up their fault localization process. The iterations of step recommendation starts with a feedback on the trace step with observable fault, and stops until the earliest buggy step on the trace is found.

We implement our approach as an Eclipse plugin, *Microbat*[1]. We first conduct an automatic experiment by using *Microbat* to find 1,887 mutated bugs with simulated feedbacks on three open source projects. The automatic experiment results show that *Microbat* is able to detect 93.1% of the mutated bugs and each bug take an average of 3.4 simulated feedbacks. In addition, we conduct a case study by manually applying *Microbat* on 13 real world bugs, the result also shows that *Microbat* is able to help find 12 out of 13 bugs in practice.

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 an evaluation for both our approach and tool. The results show that our approach is both effective and practical.

[1]The metaphor lies in that microbat is a species of bat which hunts *bug* by *iteratively* sending ultra wave as *feedback* to locate the prey.

The rest of the paper is structured as follows. Section 2 presents a motivating example of our work. Section 3 describes our approach. Section 4 presents the implementation of *Microbat*. Section 5 evaluates the effectiveness of our approach with an automatic experiment. Section 6 shows our case study of *Microbat* on real world bugs. Section 7 discusses our inspiration from our automatic experiment and case study. Section 8 reviews related work. Section 9 discusses related issues and concludes the paper.

2. MOTIVATING EXAMPLE

Table 1 shows our illustrating debugging example, the specification of which is taken from a code-training website[2]. Given a valid algorithmic expression consisting of non-negative integers, open bracket, closing bracket, plus sign, or minus sign, e.g., "1-((1+2)-1)", this program should parse it into a correct value. Overall, the program parses the expression by iteratively replacing the formula inside the most inner pair of brackets with its value (line 9–16). For example, the expression "1-((1+2)-1)" will be reduced into an expression "1-(3-1)". The process continues until the expression is reduced into value expression without any bracket (line 3). Finally, it will be evaluated to a number returned as the result (line 21). In our example, however, given a 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.

For clarity of our explanation, the details of some method declaration (e.g., *evaluateSimpleExpr*() in line 11 and line 21) are presented in Table 2.

Table 1: Debugging Code Example

```
 0  class AlgorithmeticExpressionParser{
 1    public String 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            //see details in Table 2
11            int value = evaluateSimpleExpr(expr);
12            String beforeExpr = expr.substring(0,
                    bracketStartIndex);
13            String afterExpr = (i >= expr.length()) ? ""
14              : expr.substring(i + 1, expr.length());
15            expr = beforeExpr + value + afterExpr;
16            break;
17          }
18        }
19      }
20      //see details in Table 2
21      int result = evaluateSimpleExpr(expr);
22      return result;
23    }
24
25    ...
26  }
```

Traditionally, developers find the bug by running a debugger with set breakpoints. In such case, they need to first figure out where the breakpoints should be set. In our example, given that the *result* variable in line 22 is wrong, every statement possibly influencing it is suspicious, which makes it legitimate to consider almost every line in Table 1 as a breakpoint. Too many breakpoints may suspend the debugging execution when unnecessary. However, any miss of a breakpoint may cause the debugging execution suspended after when the bug has already occurred, which needs re-running the program from the very beginning. Even worse, even with appropriate breakpoints, developers have to manually inspect variable values every time a breakpoint is reached. Thus, when the breakpoints are set inside a (nested) loop, for example line 11 in Table 1, with the number of iterations increases, the effort of inspecting variable values soars dramatically.

For above case, *Microbat* will first generate the execution trace by a single run and analyze all the read or written variables and their values in each step of the trace. Given the visualized trace (see Figure 8 in Section 4), developers are allowed to start debugging in a backward way. More 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.

For example, given the visualized trace, the developer can easily observe the program state in the step running into line 22 in Table 1, in which the *result* 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 the tool 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 and control dominance relation between steps to alleviate the burden of setting breakpoints. In above case, *Microbat* will first recommend the most recent step writing the *expr* variable (data dominance), i.e., the latest step running into line 21, and highlight its corresponding source code line. It can be observed in *Microbat* that this step reads a variable *expr* of value *"5+1"* and writes the variable *result* of value *6*.

Clearly, this step itself is correct. Given the value of written variable has been indicated as wrong, it can be inferred that the read *expr* variable must be wrong. Therefore, the developer can further select the *expr* variable to indicate its wrong value as feedback. Then, with simple causality analysis, *Microbat* will recommend the step running into line 15, which writes the previously selected *expr* variable. In this step, the developer will select *value* variable of wrong value 5, then *Microbat* will recommend a recent step running into line 11 for further inspection.

Loop Pattern Analysis. In order to reduce the times of variable inspection, *Microbat* leverages *loop pattern analysis* to this end. Only by above causality analysis, the developer will repeatedly inspect the steps running into line 15 and line 11 in every iteration in our example. 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 of "(((1+((1+2)+(2-1))-(1-3))+1)+1)+1". Since the developer inspects the variables in a backward way, he will first give feedback on the 9th iteration, then the 8th. After the developer gives feedback on the 8th iteration, *Microbat* is able to learn some loop pattern along the 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 of two positive integers, for example "5+1" in 9th iteration, "4+1" in the 8th iteration. According to developer's feedback, the bug does not occur in the 8th or 9th iteration, therefore, *Microbat abduct*s

[2]`https://leetcode.com/problems/basic-calculator/`

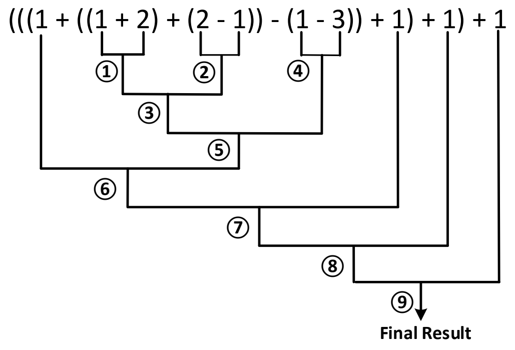


Figure 1: Execution of Example Program

that the bug may not happen in the 6th or 7th iteration either. On the other hand, *Microbat* has never encountered the case happening in the 5th iteration, in which a positive integer (4) minus a negative integer (-2). Therefore, *Microbat* stops in the 5th for developer's feedback.

In this case, the developer inspects the step running into line 11, 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 (see line 3 in Table 2) so that *Microbat* can further do recommendation inside the method invocation. Following the recommended step in line 19, line 16, line 11, and line 5 in Table 2, the developer can realize the bug lies in line 5 when the *simpleExpr* is "4- -2". The buggy program will parse the "-" in "-2" as a minus sign instead of a negative sign.

Table 2: The Invoked Code in Example

```
0   class AlgorithmeticExpressionParser{
1     ...
2
3     private int evaluateSimpleExpr(String simpleExpr) {
4       String[] operators = parseOperators(simpleExpr);
5       String[] numberStrings = simpleExpr.split("\\+|-");
6
7       String numString1 = retrieveNum(numberStrings, 0);
8       Integer num1 = Integer.valueOf(numString1);
9       for (int i = 0; i < operators.length; i++) {
10        String operator = operators[i];
11        String numString2 =
12          retrieveNum(numberStrings, i+1);
13        if (operator.equals("+")) {
14          num1 = num1 + Integer.valueOf(numString2);
15        } else if (operator.equals("-")) {
16          num1 = num1 - Integer.valueOf(numString2);
17        }
18      }
19      return num1;
20    }
21
22    private String retrieveNum
23      (String[] numberStrs, int index){
24      int num;
25      try{
26        num = Integer.valueOf(numberStrs[index]);
27      } catch(Exception e){
28        e.printStackTrace();
29      }
30      return num;
31    }
32  }
```

Clarity Guidance. In some cases, developers could get lost when inspecting the correctness of program state. *Microbat* also 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 into line 26 in Table 2, he can provide the *unclear* feedback so that *Microbat* will recommend the latest step running into a "method invocation" step line 12 or "loop head" step in line 9 to provide a "bigger picture". 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 mentioned above; if the follow-up feedback is correct-step, *Microbat* will recommend a step close to the most recent step with unclear feedback. The recommendation for more "clear" or abstract step will not stop until the developer makes the feedback showing that he has understood his unclear step.

3. APPROACH

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

3.1 Overview

Figure 2 shows the overview of our approach. Given a buggy program (and its fault-revealing input), we first run it and construct a trace model which records its execution trace and captures the causality relation between the trace steps. Then, on every step of the trace, developers can provide different types of feedback such as *correct step*, *wrong variable value*, *wrong path*, and *unclear*. Given the feedback on a step, our approach can recommend a suspicious step for further investigation. The step recommendation is conducted in two folds. First, when no feedback has ever provided before, our approach recommends a suspicious step based on causality relations in trace model. Second, after the developer has made several feedbacks on the trace, our approach will learn and summarize some patterns from the feedbacks so that our recommendation can further speed up the fault localization process. Clearly, the developer's feedback and system's step recommendation can form a number of iterations. The iterations start from the step in which the fault is observed and end at the step in which the ringleader step is found.

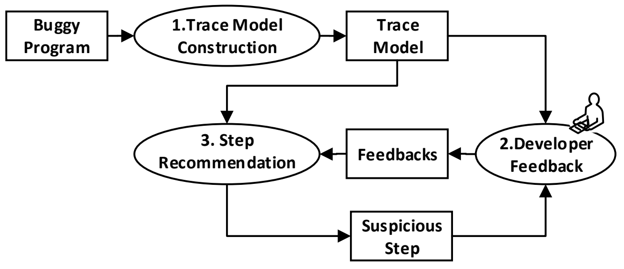


Figure 2: Approach Overview

3.2 Trace Model Construction

Figure 3 shows the meta model of our trace model, which depicts the trace model entities (i.e., trace, step, and variable) and their relations (i.e., contain, define, use, and data/control dominate). Given a run of the buggy program, we are able to achieve 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.

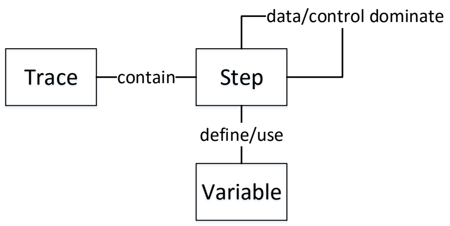


Figure 3: Meta Model of Trace Model

We construct the trace model by two runs of the buggy program. In the first run, we collect some preliminary code information for follow-up use. we start by collecting all the source code involved in this execution to get a narrowed scope so that we do not need analyze the whole program. Then, we parse the byte code in the scope to extract the corresponding PDG (program dependency graph). Based on the PDG, we map each source code line to its corresponding byte code and identify its read and written *static* variables. 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 called a **runtime variable**, and all the variables in our trace model are runtime variables. Contrary to static variables, a runtime variable will only be defined once, i.e., the time when it is written in certain step. For example, each time when line 15 of Table 1 is reached in the execution, there will be a new runtime *expr* variable defined. Clearly, 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 by referencing the control flow in PDG, and data dominance relations by referencing the defined and used runtime variables among steps.

3.3 Developer Feedback

There are four types of feedback supported in our approach:

- **Correct Step:** The step is executed in correct control flow and all the variables visible in this step are with correct value.
- **Wrong Variable Value:** Some variables in this step are 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.

Given a visualized the trace, developers are allowed to provide one of the four types of feedback on a certain step.

3.4 Recommendation Mechanism

Figure 4 shows the debugging state machine we used to recommend suspicious steps. When a feedback is provided, we will infer the **debugging state** in which the developer is. Figure 4 shows five debugging states and their transitions, a rounded rectangle represents a state while an edge represents a transition between two states. In addition, a state with dashed line represents a composite state, which further consists of a number of sub-states.

Overall, the states are categorized into *Clear Specification* state and *Clarity Inference* state. When the developer is able to decide whether a step is correct or not, he is in *Clear Specification* state, otherwise, he is in *Clarity Inference* state. In the former state, our approach infers a suspicious step and recommend it for the developer to inspect. In the latter state, our approach tries to recommend a step which is more semantically abstract in program in order to guide the developer to understand his unclear step. The details of *Clarity Inference* state will be further elaborated in Section 3.4.3.

The composite *Clear Specification* state consists of three substates, i.e., *Simple Inference* state, *Pattern Inference* state, and *Inspect Details* state. Basically, given a wrong-variable-value feedback or a wrong-path feedback on one step, we recommend its corresponding data or control dominator as the suspicious step. When our approach can only leverage the basic dominance relations, we call the debugging state as *Simple Inference* state. In our approach, we assume that the first feedback starts from a step with observable fault. Therefore, *Simple Inference* state is the initial debugging state. After the developer provides his feedback for a number of times, our approach is able to learn and summarize some step patterns of how the developer navigate through the trace. Based on these patterns, our approach is able to skip some of steps which should have been recommended by basic dominance relations so that the process of fault localization can be speeded up. Therefore, once we found a summarized pattern which can be used to skip some steps, we infer the developer is in *Pattern Inference* state. The details of *Pattern Inference* state will be further elaborated in Section 3.4.1. In *Simple Inference* state, we recommend step in grain of dominance relations between two steps. However, the ringleader step can lie in between a dominator step and its dominatee step. When our approach detects such a possibility, we infer the developer is in *Inspect Details* state so that we can recommend the suspicious steps in finer grain. The details of *Inspect Details* state will be further elaborated in Section 3.4.2.

Once the developer provides a feedback, we will interpret it into an event to trigger a state transition. Some of the interpretations are straight-forward, which are the feedback itself, 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 4, which will be explained in Section 3.4.1, Section 3.4.3, and Section 3.4.2.

3.4.1 Pattern Inference

Rationale. Loop is usually the main cause of a long execution trace of a program, nevertheless, the cases handled in loop are often enumerable. In our motivating example, when parsing the algorithmic 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 3–19 in Table 1). However, the iterations can be summarized into 3 **loop case**s: 1) the case when one positive integer adds a positive integer, e.g., the 7th–9th iterations; 2) the case when one positive integer subtract a positive integer, e.g., the 2nd and 4th iterations; 3) the case when one positive integer subtract a negative integer,

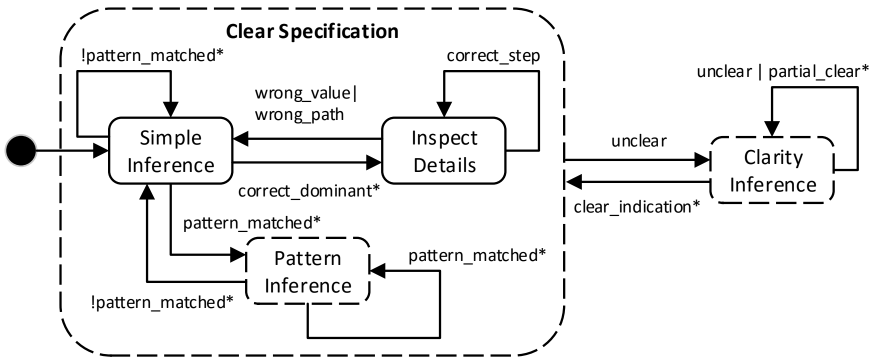


Figure 4: Overall Recommendation State Machine

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 a iteration of a loop case can be inferred as bug-free from 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. Of course, such an abduction can be too aggressive in some cases, therefore, we adopt an adjusting mechanism for complement, which will be explained later.

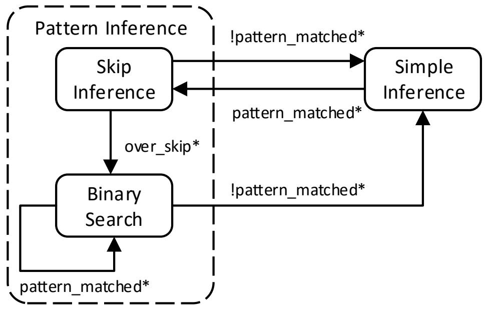


Figure 5: Pattern Inference State Machine

Figure 5 shows the general mechanism of how the *Pattern Inference* state works. The composite *Pattern Inference* state consists of two sub-states, i.e., *Skip Inference* state and *Binary Search* state. When the developer is debugging in *Simple Inference* state, we will record all the feedbacks and summarize some step paths as **path pattern** (we will provide its formal definition later). Each path pattern represents a loop case recorded as bug-free. Once the developer’s feedback indicates that he is going through a path which conforms to one of the 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 will recommend a suspicious step based on the *abduction*, i.e., skipping a number of iterations falling into the loop cases inferred as bug-free. The skipping stops either at the first iteration of loop or a new loop case is encountered. If the recommended step is provided a wrong-variable-value or wrong-path feedback, it means that the ringleader step happens before the recommended step, then a *!pattern_matched* event is triggered, which transits the debugging state back to *Simple Inference* state. However, if the recommended step is provided a correct-step feedback, it means that the bug happens in one of the skipped iterations, then an *over_skip* event is triggered, which transits the debugging state to *Binary Search* state. In *Binary Search* state, we adopt a binary search strategy in all the skipped iterations in *Skip Inference* state. Given a recommended step in *Binary Search* state, according to the developer’s feedback which indicates whether or not he is still exploring the path conforms to the summarized path pattern, either a *pattern_matched* event is triggered, which keeps the debugging state in *Binary Search* state, or a *!pattern_matched* event is triggered, which transits the debugging state back to *Simple Inference* state. Last but not least, once the bug is found in *Skip Inference* or *Binary Search* state, our approach can additionally infer the bug type. Next, we will explain the details of pattern extraction, step skipping, binary search, and bug type inference.

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**. If a step is an attributed step, it means that its incorrect program state is caused by some step executed before. In contrast, if a step $step$ has been checked by the developer and all the read and written variables are not marked as wrong value, then we call this step as a **clean step**. Given a step path $path$ on the trace, let us note its start step as $step_{start}$ and its end step as $step_{end}$, we call $path$ as a **clean path** if it satisfies that both the $step_{start}$ and $step_{end}$ are attributed steps or clean steps. A clean path is regarded as a path free of bug.

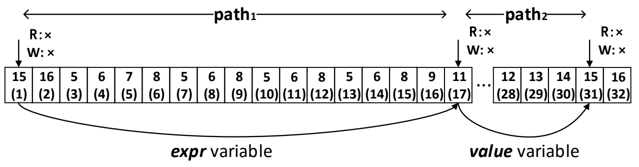


Figure 6: Path Example

Figure 6 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 1 and the lower number in brackets indicates its order. The dots between the 17th step and 28th step indicates that the steps inside method invocation in line 11 in Table 1 are omitted. In addition, the curve lines indicate data dominance relations and their attributed variables (See definition in Section 3.2). Suppose the developer has provided his wrong-variable-value sequentially in the 31th step (line 15), 17th step (line 11), and 1st step (line 15), then we have two clean paths (i.e., $path_1$ and $path_2$).

For each clean path, we will merge its consecutive repetitive sub-path with regard to its loop information. For example, the 10th, 11th, and 12th step will be merged, so that a merged path "15, 16, 5, 6, 7, 8, 5, 6, 8, 5, 6, 8, 9, 11" will be generated. We call such a merged path as the **pattern key** of a clean path. All the clean paths sharing the same pattern key fall into the same **path pattern**. In addition, the 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 (see Section 2), the loop cases, such as the addition of two positive integer, the substraction of one positive integer and one negative integer, are distinguished by this means.

Step Skipping.

In *Simple Inference* state, our approach recommend the suspicious step by dominance relation. For example, the developer marks the runtime *value* variable is wrong on the 31th step, then our approach recommend the 17th step where the *value* variable is defined. When the developer's wrong-variable value feedback can generate a clean path, we will record its pattern key. Given a step $step_{cur}$, our approach recommend a suspicious step $step_{sus}$. Then we check whether one of $step_{sus}$'s data dominator, $step'_{sus}$ can (1) form a *possible* clean path with $step_{sus}$, i.e., <$step'_{sus}$, $step_{sus}$>, to share the same pattern key with one of the recorded path patterns $patt$ and (2) the attributed variable of the dominance relation is homologous (see definition in Section 3.2) to the cause variable of $patt$'s label path. If $step'_{sus}$ satisfies both conditions, we will 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 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 some new clean path can be formed to match recorded path pattern once again. If yes, we skip $step'_{sus}$ once again in the same vein, otherwise, skip process stop and the $step'_{sus}$ is recommended.

For the example in Figure 6, suppose we have *already* recorded a path pattern $patt$ = "15, 16, 5, 6, 7, 8, 5, 6, 8, 5, 6, 8, 9, 11" before and the cause variable of its label path is a runtime *expr* variable used in line 11. Let the $step_{cur}$ be the 31th step (line 15) in which the *value* variable is marked as wrong, then the 17th step (line 11) 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 another *expr* variable used in line 11. 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 recorded pattern once again.

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 too aggressive, therefore, it is possible that we may over-skip some steps. When we find the developer provide a correct-step feedback on the step recommended in *Skip Inference* state, which means 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 (see definition in Section 3) 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, if the developer (1) provides a correct-step variable, 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 corresponding cause variable, which means that we still need skip some steps and the ringleader step happen before $step$, we keep the binary search. Otherwise, the debugging state returns to *Simple Inference* which recommends the step by dominance relation.

Bug Type Inference.

If we can generally locate the bug in *Skip Inference* or *Binary Search* state, which means the ringleader step is located in loop, then our approach can additionally infer the bug type. In our approach, there are two types of bug we can infer, i.e., missing-corner-case bug and branch-mistake bug.

Our insight is as follows. Consider a program p with 2 test cases, t_1 and t_2, so that t_1 fails and t_2 passes. If t_1 and t_2 go through *exactly* the same path in p, then we can infer that t_1 fails because p miss considering the case of t_1. Thus, it is a missing-corner-case bug. On the other hand, if t_1 and t_2 go through different paths of p, then we can infer that the bug lies in the differential branch t_1 go through. Thus, it is a branch-mistake bug.

In our case, we can regard the code in the loop as a sub-program p_{sub}, and a set of bug-free clean path as the passing tests for p_{sub}. In the process of *Skip Inference* or *Binary Search* state, once the developer find a step in which the read variables are correct while the written variable is incorrect, we can think that we now have a failed test case for p_{sub}. Therefore, we can infer the bug type by the above thought. For example, the step running into line 11 in Table 1 (i.e., 5th iteration in Figure 1) in which the read *expr* variable of *evaluateSimpleExpr()* method is "4- -2" but the returned *value* variable is 2, in this case, we can detect that, comparing the other inputs, such an input go through a different path. Therefore, we can infer a branch-mistake type of bug in this case. In effect, it goes through a wrong branch in line 48 in Table 2.

3.4.2 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. Otherwise, i.e., the developer provides wrong-variable-value or wrong-path feedback, the debugging state will be transited to *SimpleInference* once again.

3.4.3 Clarity Inference

Figure 7 shows how *Clarity Inference* state works. Once the developer cannot decide the correctness of a step, he can provide a unclear feedback, which triggers a *unclear* event. Then we will back up the debugging state at that time and transit the debugging state into *Clarity Inference* state. The aim of *Clarity Inference* is to guide the developer better understand the unclear step so that he can go back where he provide unclear feedback to resume the backed-up state in *Clear Specification* state. The composite *Clarity Inference* state consists of two sub-states, i.e., *Unclear* state and *Partial Clear* state. Whatever state the developer is in, a provided unclear feedback will trigger a *unclear* event, transiting the debugging state to *Unclear* state. In *Unclear* state, we tries to find and recommend a semantically more abstract step with regard to the recorded feedbacks. If the developer provide a wrong-variable-value or wrong-path feedback on the recommended step in *Unclear* 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 provide a correct-step feedback on the recommended step in *Unclear* 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*. Next, we present the details of our approach in *Unclear* state and *Partial Clear* state.

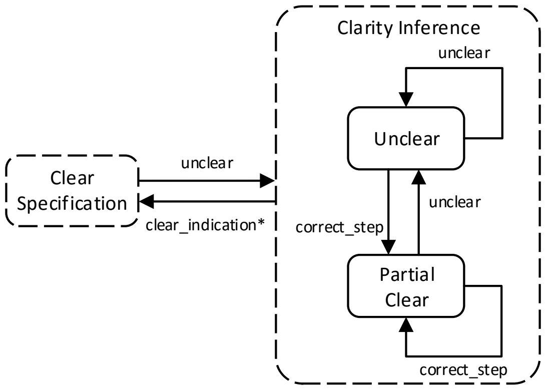


Figure 7: Clarity Inference State Machine

Unclear State.

We consider the abstract level of step in terms of syntactic structure, i.e., loop and method invocation. Given two steps s_1 and s_2, we say that s_1 is the **direct parent** of s_2 if either of following conditions happens:

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

The above relation can make our trace into a **step tree**, in which the root is the entry method and the leaves are the steps invoke no method or start no loop. Given a step, we define its layer on step tree as its **abstract level**. By default, we let the abstract level of the entry method be 0. In *Unclear* state, given current step *step_cur* marked as unclear, let its direct parent be par and the level of par be l, we first consider recommending *step_cur*'s direct parent, par. If the abstract level of par is 1, we recommend the nearest step executed before par.

Partial Clear State.

The developer can provide many unclear feedbacks, we record the "unclear" steps in a list $visitedUnclearSteps$. After the developer provide a correct-step feedback on current step $step_{cur}$ and the debugging state is transited in *Partial Clear* state, we select a step according to the routine as follows.

We first remove all the steps in $visitedUnclearSteps$ which is executed before $step_{cur}$. If there is no step left in *visitedUnclearSteps* (i.e., the "unclear" step is visited once again and provided with a correct-step feedabck), then we resume the *Clear Specification* state before entering *Clarity Inference* state. Otherwise, we try to select a abstract step in between $step_{cur}$ and the earliest step in $visitedUnclearSteps$ (noted as $earlyVStep$). Technically, let $earlyWStep$ be the earliest step with wrong-path or wrong-variable-value feedback on trace, we tries to find a dominator of $earlyWStep$ so that (1) it is executed before $earlyVStep$ and after $step_{cur}$, and (2) its abstract level is higher than that of $earlyVStep$. If multiple dominators conforms to such a criteria, we ranks them by their executed order and choose the median. If no such dominator exists, we choose $earlyVStep$ as the recommendation.

4. TOOL SUPPORT

We implemented our approach as an Eclipse plugin. As showed in Figure 8, *Microbat* consists of two buttons on toolbar and two views.

A click of the left *Start* button starts to build the trace model of the target program. The recorded trace will be presented in *Trace* view. In *Trace* view, the steps are organized in tree structure, and each step is labeled with its execution order, class file name, and line number. The parent-child relation between steps in *Trace* view conforms to the step-in relation in traditional debugger, i.e., if a step starts a method invocation (e.g., 117th step in Figure 8), the subsequent executed steps in the corresponding method are its children. Once the developer clicks a step on *Trace* view, the corresponding line of code will be highlighted in Java Editor, and its specific information will be showed on *Feedback* view.

At the top of *Feedback* view shows the four types of feedback, i.e., correct-step, wrong-variable-value, wrong-path, and unclear. In addition, *Feedback* view lists its read and written variable, as well as a snapshot of program states when the corresponding step is reached. When a wrong-variable-value feedback is provided, the developer needs to click at least a variable. Once a feedback is provided, the developer can click the *Find Bug!* button to make *Microbat* to recommend a step. In addition, he can also ask *Microbat* to infer bug type by clicking the *Infer Type* button.

Any non-unclear feedback will leave a mark (check or cross) on specific step on *Trace* view, which indicaties its correctness. Last but not least, the developer can click *Undo* button to get back to the state before the recommendation for current step.

5. AUTOMATIC EXPERIMENT

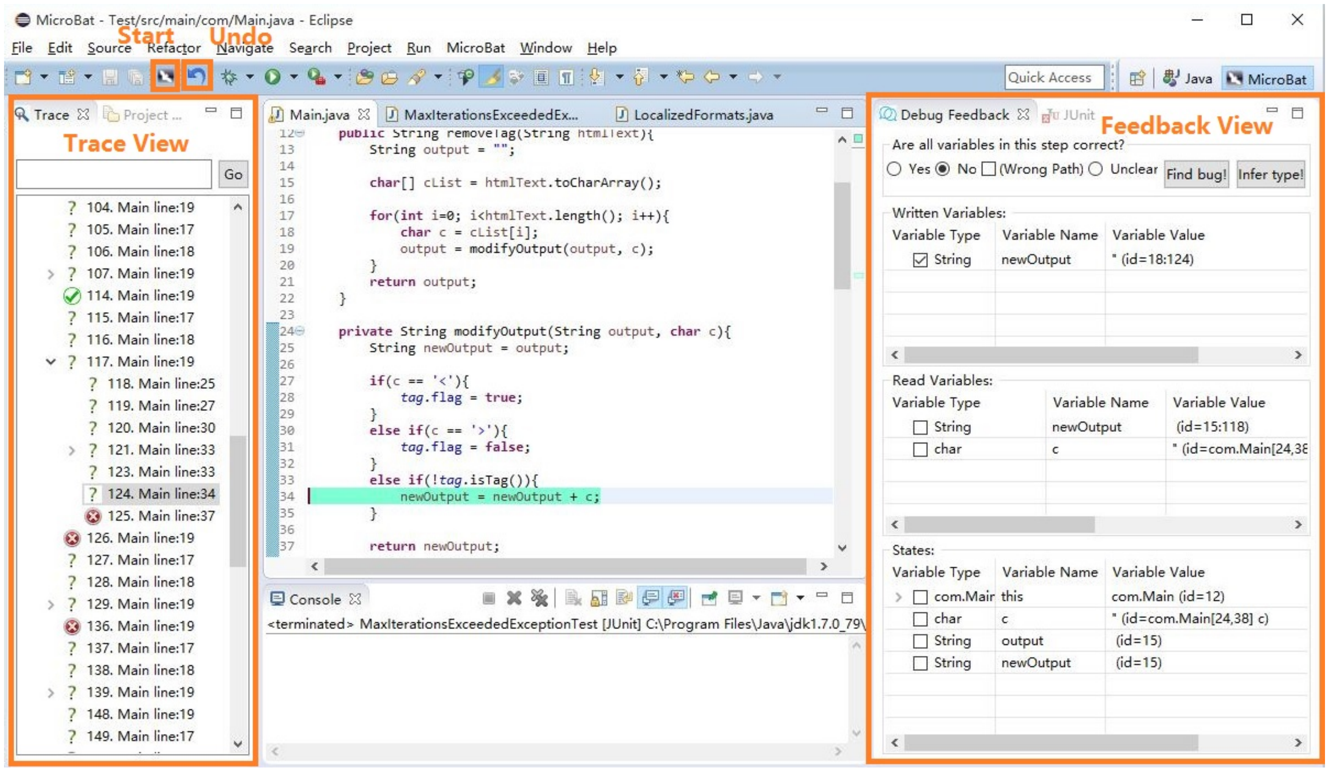
We conduct an experimental study to answer the following research questions:

- **Q1**: Whether *Microbat* can help find the bug in general?
- **Q2**: How efficient can *Microbat* find the bug?

5.1 Design

In order to answer the above two research questions, we conduct an automatic experiment to evaluate *Microbat*. The challenge of the automatic experiment lies in the simulation of developer's feedback. More specifically, given a buggy program and a fault-revealing input, we need to know the exact program state for each step so that we are able to simulate a correct feedback for *Microbat* to recommend suspicious step. In this work, we leverage test case mutation to this end.

We first collect test cases from three Apahce open source projects (see Table 5.2). For each passed test case, we mutate its tested code



```
? 104. Main line:19
? 105. Main line:17
? 106. Main line:18
> ? 107. Main line:19
✓ 114. Main line:19
? 115. Main line:17
? 116. Main line:18
v ? 117. Main line:19
    ? 118. Main line:25
    ? 119. Main line:27
    ? 120. Main line:30
  > ? 121. Main line:33
    ? 123. Main line:33
    ? 124. Main line:34
    ✗ 125. Main line:37
✗ 126. Main line:19
? 127. Main line:17
? 128. Main line:18
> ? 129. Main line:19
✗ 136. Main line:19
? 137. Main line:17
? 138. Main line:18
> ? 139. Main line:19
? 148. Main line:19
? 149. Main line:17
```

```java
    public String removeTag(String htmlText){
        String output = "";

        char[] cList = htmlText.toCharArray();

        for(int i=0; i<htmlText.length(); i++){
            char c = cList[i];
            output = modifyOutput(output, c);
        }
        return output;
    }

    private String modifyOutput(String output, char c){
        String newOutput = output;

        if(c == '<'){
            tag.flag = true;
        }
        else if(c == '>'){
            tag.flag = false;
        }
        else if(!tag.isTag()){
            newOutput = newOutput + c;
        }

        return newOutput;
```

```
<terminated> MaxIterationsExceededExceptionTest [JUnit] C:\Program Files\Java\jdk1.7.0_79\
```

Are all variables in this step correct?

Yes [✓ No] ☐ (Wrong Path) Unclear

Written Variables:

Variable Type	Variable Name	Variable Value
☑ String	newOutput	" (id=18:124)

Read Variables:

Variable Type	Variable Name	Variable Value
☐ String	newOutput	(id=15:118)
☐ char	c	" (id=com.Main[24,38

States:

Variable Type	Variable Name	Variable Value
> ☐ com.Mair	this	com.Main (id=12)
☐ char	c	" (id=com.Main[24,38] c)
☐ String	output	(id=15)
☐ String	newOutput	(id=15)

Figure 8: Screenshot of *Microbat*

with a standard mutator. If a mutation can kill the test case, we say that we have a **trial** for our evaluation. Given a trial, we can generate the correct trace before mutation, $trace_{cor}$, and the buggy trace after mutation, $trace_{mu}$. Then, our simulation can reference $trace_{cor}$ to see whether some steps in $trace_{mu}$ is correct or not.

Technically, we leverage an LCS-based dynamic programming algorithm [11] to match the steps between the two traces. If a step in $trace_{mu}$ cannot be matched to a step in $trace_{cor}$, we simulate a wrong-path feedback on it. Otherwise, we difference the read and written variables, as well as the programs state between two matched steps to check whether or not the variable values are correct. If not, we simulate a wrong-variable-value feedback, otherwise, we simulate a correct-step feedback.

As for develop's unclear feedback, we design the simulation as follows. If the step is the fault-revealing step at the end of trace, the "simulated developer" will not provide a unclear feedback. Otherwise, given a step s which has an abstract level (see definition in Section 3.4.3), l, and it is the kth times checked by our "simulated developer", then, the probability $P(l, k)$ to make a unclear feedback is

$$P(l,k) = \frac{1 - \frac{1}{e^{l-1}}}{k} \tag{1}$$

Literally, the lower level s is or the less times s is checked, the larger the probability to simulate a unclear feedback on s.

In a trial t, we consider the mutated line of source code $line_{mu}$ as the root cause of the bug. Let the trace length as l_t, if *Microbat* can recommend a suspicious step which runs into $line_{mu}$ within l_t feedbacks, we consider the trial as successful, otherwise, we consider the trial as failed.

In this experiment, we avoid the too long traces for implementation consideration. It is because that (1) our current implementation requires creating a snapshot of program state in each executed steps, and (2) our dynamic programming algorithm for step matching has a space complexity of $O(n^2)$ (n for trace length). Therefore, we limited the trace length to 8,000 steps. We will further discuss this problem in Section 7.

5.2 Results

Table 5.2 shows our experiment results on three open source projects, which shows the number of test cases we use (#TC), number of evaluated trials (#Trial), the ratio of successful trials (SR), the average length of traces (ATL), and the average number of feedbacks (AF).

Table 3: Experiment Result

Project Name	#TC	#Trial	SR	ATL	AF
Math2.2	306	1173	90.7%	293.0	3.8
Collections3.2.2	165	404	97.5%	48.4	2.0
Ant1.9.6	128	310	96.1%	115.0	4.2
Total	599	1887	93.1%	207.0	3.4

5.2.1 Effect of Microbat

Table 5.2 shows that *Microbat* is able to find the majority of the mutated bugs (totally 93.1%). It means that the simulated developer can locate the fault based on our recommendation paradigm.

Nevertheless, we further check the failed trials and find that the main failure reason lies in that our simulated developer my provided a larger number of unclear feedback in certain cases. For example, the test case *HypergeometricDistributionTest#testDegenerateFullSample* in Math project, its mutated trace consists of 463 steps, however, the majority of its steps has very low abstraction level (i.e., under the nested loop and method invocation). Therefore, based on Equation 5.1, the probability of giving unclear feedback is very high. In this trial, our simulated developer continually provide unclear feedback on these steps, which eventually make the number of recommendation iterations proceed to the trace length. Even worse, some test cases involves recursive method invocations, which makes the abstract level of some recursive method invocation even lower. Thus, in these trials, the simulated developer will provide even more unclear feedbacks on these steps, which incurs a number of failed trials. We will discuss more of this problem in Section 7.

5.2.2 *Efficiency of Microbat*

Table 5.2 also shows that *Microbat* generally requires the developer to inspect an average of 3.4 steps. In contrast, the average trace length is 207.0 and the longest one is 7050.

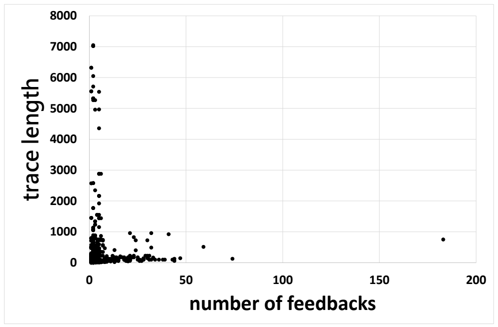


Figure 9: Trace Length and Feedback Number

Figure 9 shows the scatter plot of all the successful trials, in which each point represents a trial, and horizontal axis represents the number of required feedbacks and vertical axis represents the trace length. We can see that the majority of trials require less than 50 feedbacks to locate the bug. In addition, the number of feedbacks does not increase with the trace length. Nevertheless, we check the outlier which requires 183 feedbacks. We found that the unclear feedbacks account for phenomenon once again. For the outlier, 79 out of 183 feedbacks are unclear feedback.

In summary, our experiment shows that (1) *Microbat* is able to effectively and efficiently locate the bug in general, and (2) the performance of *Microbat* can get impaired if the developer provide a large number of unclear feedback.

5.3 Threat to Validity

There are mainly three threats in our experiment. First, the mutated bugs are still different from the real-world bug in practice. Nevertheless, Andrew et al. [3] empirically assess the effect of mutation and their result shows that the use of mutation operator not only yields trustworthy result but also seeded faults that seems to be harder to detect. Second, we matched steps among two traces with a dynamic programming algorithm, when defining the step similarity between two steps, we leveraged some heuristics such as variable similarity, line similarity, etc. Therefore, it is possible that we may fail to match proper steps in some cases. To address this threat, we sampled dozens of trials to inspect our match effect, the result shows that the algorithm works in the majority of cases. Third, the trace length of the trials in this study is comparatively small, we will try to extend our study on trials with longer trials.

6. CASE STUDY

We conducted a case study by recruiting two developers to use *Microbat* on 13 real world bugs, and check whether *Microbat* can help them locate the root cause of the fault. We select the 13 bugs from 3 projects in Defect4j database [12]. In the database, each bug is attached with a failed test case, a buggy revision and a fix revision, therefore, we can infer bug reason by differencing these two revisions. We recruited two developers who has Java programming experience over 6 years for this study. Our interview indicated that each developer was unfamiliar with the projects. We assigned 6 or 7 bugs to each developer. We told them that they did not need to fix the bug but they needed to specify underlying reason why the bug happens, in addition, they are free to give up when they felt exhausted. Moreover, they are required to record and report their feedbacks during the debugging. We checked the reason specified by our developers for each bug, if the provided reason is justifiable according to our answer, we consider the bug as a success of *Microbat*. On the other hand, if he told us he decided to give up the debugging, we consider it a failure.

Table 4: Real World Bugs

Project	Math	Chart	Lang	Total
#Bug	5	4	4	13
SR	80%	100%	100%	92.3%
ATL	3855	2287	3001	3048
AF	13	39	10	21

Table 4 shows the results of our case study. We can see that 12 out of 13 can be successfully debugged with *Microbat*, which indicates that *Microbat* can facilitate fault location tasks in general. For example, a bug in the *getLegendItems()* method of *AbstractCategoryItemRenderer* class in Chart project, with two feedback of wrong-variable-value and one correct-step variable, *Microbat* can quickly narrow the trace scope from 1415 to 15,

In addition, we further investigate the reason why *Microbat* fails. Based on our investigation and interview with our recruited developers, they indicated that *Microbat* was not helpful enough for "missing code" bug. For example, the bug in the *divide()* method of *Complex* class in Math project, the fault-revealing step shows that the *isInfinite* field in *Complex* class in the 5373th steps is with wrong value *false*. Therefore, our developer provided a wrong-variable-value on the step, however, *Microbat* quickly recommend a step in 16th step in which the *isInfinite* field is defined without any initialization. The developer told us that he knew that the *isInfinite* variable should have been initialized somewhere between 4200th and 4300th step, however, *Microbat* cannot support helpful recommendation under such scenario. For this case, *Microbat* will transit the debugging state into *Inspect Details* state. Given our current recommendation strategy in *Inspect Details* state, he may need to decide where the initialization should be added among over 4000 steps. The debugging task is soon turned to a program comprehension tasks, given the developer's unfamiliarity with the project, he decide to give up in this case. We will discuss more of the issue in Section 7.

Nevertheless, our case study shows that *Microbat* can still guide the developers to locate the fault in majority cases even they know little about the project. Based on their reported debugging feedback, we find that *Microbat* is able to effectively recommend intuitive steps and finally locate the step where the bug lies.

7. DISCUSSION

In this section, we discuss several issues revealed by our automatical experiment and case study.

Debugging and Program Comprehension.

Our approach assumes that the specification of detailed code exists in human mind. Our case study shows that such an assumption holds true in general but sometimes may not be the case. Especially for the developers who need to debug the code written by others, they may not be able to provide a clear or decisive feedback on certain steps. More specifically, they need to perform the program comprehension task first and the debugging task second.

The *Clarity Inference* state in our recommendation is designed to assist the program comprehension. In our automatic experiment, we see that our recommendation for unclear steps can finally guide the simulated developer back to where the bug lies. However, more interestingly, we observed in our case study that, when the real developers were unclear about a certain step, they would additionally explore some steps before or after our recommended step. After the exploration, they would return back to the recommended step and provide another feedback. Based on such an observation, we think that our recommendation in *Clarity Inference* state is useful but still a little bit far from enough. Developers usually need to inspect a "bigger" context of a specific step to understand the program. In such case, the executed trace is useful for their comprehension of the debugged program. Therefore, we believe that the trace can be further explored to even better facilitate the program comprehension tasks. We would further investigate the possibility in our future work.

Cost for Trace Capture.

Similar to other trace recording techniques [18, 19], the trace capture causes a significant slowdown on the debugged program. To the extreme case, when the bug is caused by an infinite loop, we would eventually get a trace with its length to our set limitation in *Microbat*. One of the possible solution is that we can equip *Microbat* with a feature to allow developer to set a "execution range". Therefore, we just build a trace-model for that execution range.

Nevertheless, such a problem origins from the trace-off between information completeness and the runtime performance of debugger. The more information we need, the worse performance we may have. Usually, it should be the developer who make a decision on choosing traditional debugger or trace-recording debugger. We may explore the possibility for the automation of such a decision based on some program analysis technique.

Reasoning in Some Debug State.

Our step recommendation is based on the dominance relation among steps, pattern summarization, as well as an empirical state machine. Generally, our recommendation relies more on the code logic rather than business logics. Therefore, it is hard to make more insightful recommendation in some debugging state such as *Inspect Details* state. Our case study also confirms this point. In order facilitate more sophisticated reasoning, we may need to incorporate more business-logic related feedbacks from the developers.

A possible attempt is to provide a query mechanism to allow the developer search for their specific steps. In the duration of query, we are able to extract business logics so that the reasoning can better facilitate our recommendation.

8. RELATED WORK

Spectrum-based fault localization techniques [1, 2, 22, 24, 25] 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. [22] first proposed the idea of spectrum-based fault localization, and the researchers keep improving technique over the years. Renieris et al. [21] proposed a simple spectrum-based technique and implement a tool called WHITHER. Wang et al. [25] improved the effect of fault localization by addressing the coincident correctness problem. Abreu et al. [1] 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 [2].

Similar to spectrum-based techniques, delta-debugging [7, 10, 17, 20, 28, 29, 30] also requires a set of passed and failed test cases. However, they compared the difference of test cases in more aspects than code coverage, such as test input [30], program states [7, 29], path constraints [20], etc. Zeller et al. [28] first proposed the idea of delta debugging and used it in regression testing. Then, they exploited the technique to simplify test case [30], isolate bug-causing variable [7, 29], and etc. Followed by their work, Misherghi et al. [17] proposed an improvement to refine the result of delta-debugging. With similar idealogy, Qi et al. [20] and Yi et al. [26] referencing the "delta" in versions of regression testing to facilitate fault localization.

Different from these techniques, our approach assume no comparison with a passed test case. In addition, compared to spectrum-based techniques, our approach locate the bug in finer grain in terms buggy step instead of line of source code.

Similar to our approach, a lot techniques [4, 13, 14, 15, 18, 19, 23, 27] leverage program execution trace for the fault localization. Ressia et al. [23] proposed an object centric debugging approach which facilitates tracking a specific object instance during the execution. Yuan et al. [27] proposed a tool called SherLog which infers the reason of program failure by combining recorded program log and source code. Pohier et al. [18, 19] proposed omniscient debugger which records the whole execution trace of a debugged program and enable user to explore it. Ko et al. [13, 14, 15] 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 different from these work in that (1) we allow the developer to provide richer types of feedback during the debugging and (2) we are able to learn and summarize patterns from developers feedback so that the debugging process can be speeded up.

Additionally, Lo et al. [9] proposed 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 allow developers to provide feedback on execution steps to localize the fault.

9. 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, in addition, our case study indicates that our tool *Microbat* is of practical use to facilitate the debugging tasks.

In our future work, we first aims to address the problems revealed in our evaluation. We would support more sophisticated reasoning for some debugging state and improve the runtime efficiency of building trace model. In addition, we would like to support more features such as supporting a query mechanism on trace so that developers have more freedom to explore the trace. Finally, we will have more cooperation with our industrial partner and further evaluate our *Microbat* tool in larger scale.

10. REFERENCES

[1] 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.

[2] 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.

[3] 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.

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

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

[6] 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.

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

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

[9] 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.

[10] 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.

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

[12] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, 2014.

[13] 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.

[14] 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.

[15] 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.

[16] T. Liu, C. Curtsinger, and E. D. Berger. Doubletake: Evidence-based dynamic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, 2017. accepted.

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

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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