Improvement of unreachable code detection and fault localization for Bixie

Weijie Huo
Computer Science at University of Illinois at Champaign Urbana
Urbana, USA
whuo3@illinois.edu

Minhao Zhan
Computer Science at University of Illinois at Champaign Urbana
Urbana, USA
mzhan5@illinois.edu

***Abstract*— we present a modified unreachable code detector Bixie, a tool to detect unreachable code in Java code. Bixie detects unreachable code at a higher precision. However, there are cases like unreachable under multiple nested if statement it can not detect. In this paper, we introduce a improved version of Bixie so that it can cover the cases of multiple nested if statement.**

I. INTRODUCTION

A recent research on software development from University of Cambridge demonstrates that developers spend more than 50% of the development time on debugging. Fault localization is the major part of debugging, which means finding the segment of code that causes the undesired behavior. However, challenge of fault localization increases with the scale of the program and large-scale programs are pretty common today. This situation causes lots of trouble to developers. To alleviate efforts of debugging for developers, a large amount of automated testing tools are developed.[1]

In this paper, we consider the tool that detects and fault-localizes unreachable code. Unreachable code is the code segment in a program which will never be executed. Although it could not be declared that unreachable code is a bug, it leads to confusion of developers, and is often related serious bugs, for example kernel errors in both Linux and OpenBSD[1]

These are some existing unreachable code detection and analysis tool such as Bixie. However, Bixie has some issues on detection coverage and fault localization. For example, Bixie could not detect the unreachable code inside nested if statements. This situation will cause false positive, which could lead to developers missing the unreachable code. Issue of fault localization is that the line number of fault points to an empty line sometimes. This detect will cause confusion of developers, which they have to manually check the fault locations. Both cases above will cost developer extra time and efforts in debugging, so they are necessary to be fixed.

In this paper, we improved the unreachable code detection tool Bixie, on detection coverage, fault localization and fault readability by designing the new unreachable code detection and fault-localization approaches. To detect unreachable code, we construct all the path formulas for all blocks from the control flow graph, and then pass them to SMT solver princess to check satisfactory. Unreachable code will not have a feasible path procedure, which means its procedure formulas will never be evaluated to be true. To localize faults, we take the path formulas for the unreachable code as input to find the minimum unsatisfied core (Set of the all the conflicting statements that leads to unsatisfactory of the path formula) in it by splitting the path formulas into subset formulas, and then check satisfactory for all subset formulas. After all conflicting statements are found, we print the line numbers for each conflicting statement of the Unreachable code.

To demonstrate usefulness of our project, we conduct a series experiments on several poplar open source Java program and typical test codes. The goal of our evaluation is to demonstrate that unreachable code exists in real-world popular programs.

Contributions: We make the following key contributions:

1) We improve unreachable code detection approach, which could detect some common unreachable code that could not be detected by Bixie.
2) We design a unreachable code fault-localization approach for unreachable code that could not be detected by Bixie.
3) We add the feature to show the unreachable code locations, which Bixie doesn't show.

To this end, this paper explores our project of improving Bixie, an unreachable code detection and analysis tool, to achieve higher detection coverage, more accurate fault localization and high fault readability. In section 2, we provide background material on unreachable code and working principle of Bixie. Section 3 introduces some related works in unreachable code detection and their relative merits. Section 4 illustrates some typical unreachable code examples that widely exist in the real world.

Section 5 introduces the detailed design and implementation process of our project with challenges and solutions. In section 6, we show and discuss the evaluation result of our project on several popular open source projects and test code examples. In section 7, we conclude that our project provide the improvement to Bixie with the future work that could be done to improve our project.

II. BACKGROUND

A) *Unreachable code*

In computer programming, unreachable code is part of the source code of a program which can never be executed because there is no control flow path to the code from the rest of the

program. [2] Several reasons lead to unreachable code becoming issues in a program including unnecessary memory occupation, decreasing data locality and wasting maintenance effort. [3]

For memory occupation, unreachable code will never be executed; however, variables and reference in unreachable code still occupy memory space due to the compiler mechanism. These memory spaces are unnecessary for the functionality of the program, which lead to memory wasting. Memory efficiency is significant to computer program today since every developers cares about space complexity of the algorithm. Therefore, unnecessary memory occupation is one of the reasons that unreachable code is undesirable.

Unreachable code also decreases data locality. When compiling a program, CPU will cache program instructions to the CPU instruction cache. This process will be delayed by unreachable code due to the unnecessary caching of instructions in unreachable code, which leads to slower data locality (a typical memory reference feature of regular programs to accelerate memory access). [3]

Program maintenance is a problem when unreachable exists. Developers are responsible for documenting the code (e.g. Comments, Java Doc, Readme file) to maintain the code understandable to other developers, which is pretty necessary in modern team-oriented software development. Existing of unreachable code will cause developers to waste their maintenance effort on code segments that would never be executed.
There are various causes of unreachable code such as: [3] programming errors in complex conditional branches; obsolete code that a programmer forgot to delete; unused code that a programmer decided not to delete because it was intermingled with functional code; conditionally useful code that will never be reached, because current input data will never cause that code to be executed; debugging constructs and vestigial development code which have yet to be removed from a program.

The primary reason for these causes above is human error, either a legacy or programming defect. For example, programmers often write functions to help structuralize programs, which programs call these functions in the main function or other functions. However, when programmers change the function call statement (e.g comment, delete function calls), the function being called is no longer used or required even if they are once useful.

Complex component (e.g. library, module) is another source of unreachable code. When a complex component is used in a program, only a small part of the component is actually used. Compilers could not remove these unused codes when compiling the program without recompiling the runtime library. [3]

B) How Bixie works

Bixie uses Soot as a front-end to parse and pre-process Java (byte)code and Princess as a theorem prover. Bixie is composed of three components: Jar2Bpl, which translates the Jimple code generated by Soot into Boogie, GraVy, which detects and reports inconsistent code, and a spam filter to suppress false alarms [5]

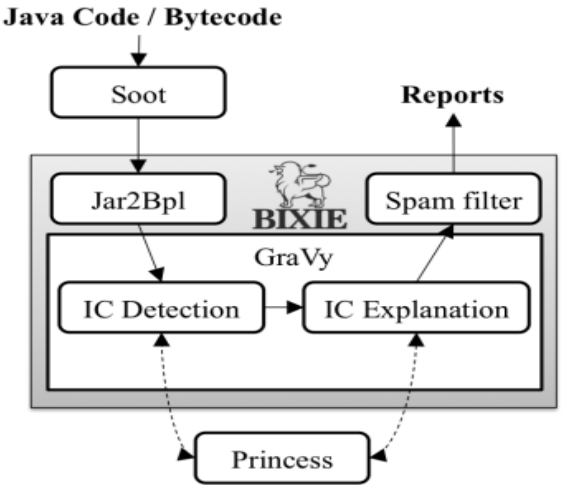

Figure. Bixie workflow diagram

1) *Bytecode Translation*

In this step, taking Java source code as input, we apply Java optimization framework Soot to translate the Java program into Jimple format, a 3-address intermediate representation of the program's bytecode. [8] This step simplifies the translation to Boogie, as it decreases the number of statement types need to be considered.

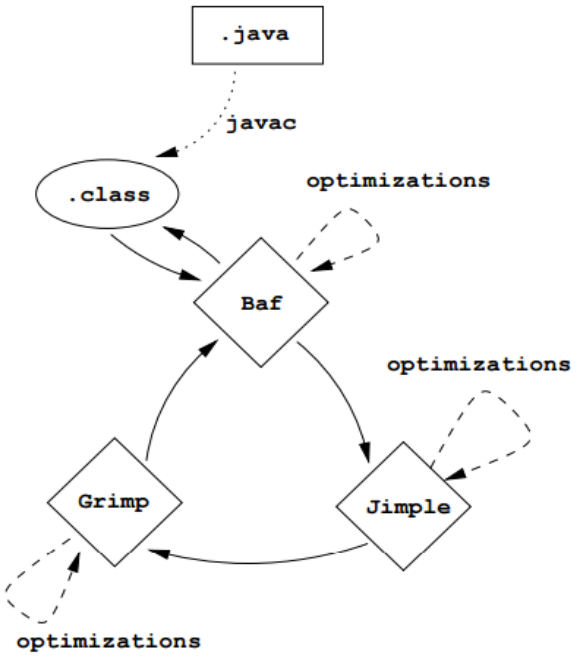

Figure. Soot workflow diagram [8]

Jimple

Jimple is a 3-address code representation of bytecode, which is typed and does not include the 3- address code equivalent of

a jsr (Jump to subroutine). In the process of transferring Baf to Jimple, additional local variables are used to replace the stack,while the natures of bytecode and stack area variables are reversed. It is an preferred form for performing optimizations and analyses, because both traditional optimizations such as copy propagation and more advanced optimizations such as virtual method resolution require this kind of bytecode form.[8]

```
public int stepPoly(int x)
{
    if(x < 0)
    {
        System.out.println("foo");
        return -1;
    }
    else if(x <= 5)
        return x * x;
    else
        return x * 5 + 16;
}
```

Figure stepPoly in Java source code form

```
public int 'stepPoly'(int)
{
    Test r0;
    int i0, $i1, $i2, $i3;
    java.io.PrintStream $r1;

    r0 := @this;
    i0 := @parameter0;
    if i0 >= 0 goto label0;

    $r1 = java.lang.System.out;
    $r1.println("foo");
    return -1;

 label0:
    if i0 > 5 goto label1;

    $i1 = i0 * i0;
    return $i1;

 label1:
    $i2 = i0 * 5;
    $i3 = $i2 + 16;
    return $i3;
}
```

Figure. stepPoly in Jimple form

2) **Boogie**

In this step, the Jimple format is then translated into the intermediate verification language with Boogie (a verification condition generation tool). The outputs of translation are the Control flow graph for the program. In this step, all implicit runtime assertions, such as array bounds check, are turned into assertion statements. [5]

Boogie is used as a layer on which to build program verifiers for other languages.[11]
The generation of verification conditions from bytecode is closely related to verifier design. Boogie architecture separates the concerns of deciding how to encode source language features from the concerns of how to reason about control flow in the program. Boogie provides assert statements that encode proof obligations stemming from the source program, to be checked by the program verifier, and assume statements that encode properties guaranteed by the source language and verification process, available for use in the proof by the program verifier. [10]

```
public static void M(int n) {
  Example e = new Example(100/n);
  int k = e.s.Length;
  for (int i = 0; i < n; i++) { e.x += i; }
  assert k == e.s.Length;
}
```

Figure. Example Java source code [10]

```
implementation Example.M(n : int)
{
  var e : ref where e = null ∨ typeof(e) <: Example;
  var k : int, i : int, tmp : int, PreLoopHeap : [ref, name]any;

  Start :
    assert n ≠ 0;
    tmp := 100/n;
    havoc e;
    assume e ≠ null ∧ typeof(e) = Example ∧ Heap[e, allocated] = false;
    Heap[e, allocated] := true;
    call Example..ctor(e, tmp);

    assert e ≠ null;  k := StringLength(cast(Heap[e, Example.s], ref));
    i := 0;
    PreLoopHeap := Heap;
    goto LoopHead;

  LoopHead :
    goto LoopBody, AfterLoop :

  LoopBody :
    assume i < n;
    assert e ≠ null;
    Heap[e, Example.x] := cast(Heap[e, Example.x], int) + i;
    i := i + 1;
    goto LoopHead;

  AfterLoop :
    assume ¬(i < n);
    assert e ≠ null;  assert k = StringLength(cast(Heap[e, Example.s], ref));
    return;
}
```

Figure. Boogie translation output [10]

III. EXAMPLE

We will illustrate some typical unreachable code examples and briefly explain how to apply our approach on these examples in this section.
Obviously unreachable code:

Figure.1 shows a piece of unreachable code create by us as demo. In method unreachable Code, there are nested If statements. The if statements in line 7 assumes that variable a equals 3. Then, the If statement in line 8 assumes that variable b equals 4. At line 9, the If statement checks whether a is not equal to 3 or b is not equal to 4 could be satisfied, which is obviously false. Therefore, the program will always go to the false branch at if statement in line 9, which lead to the method will only return false when it is called. This unreachable code will have negative impact on code using unreachable Code.

```
4
5   boolean unreachableCode(int a, int b){
6
7       if(a==3){
8           if(b==4){
9               if(a!=3 || b!=4){
10                  return true;
11              }else{
12                  return false;
13              }
14          }
15      }
16  }
17
```

Figure.1 unreachable code example

Confused Conjunction

Figure.2 shows a typical unreachable code example from Michael et.al "Hidden Truths in Dead Software Paths". The binary conjunction operators (e.g. && and ||) often cause logic confusion for programmers. Not only programming beginners have problem using binary conjunction operators, even experienced programmers sometime will make mistake when using them. [4] In Figure.2, a variable *maxBit* is checked in line 1842 whether it is great than 4 or less than 8. This If statement is always true because two expressions have overlap. Variable *maxBit* will always be assigned to 8, as a result, if statement in line 1845 will never go to the true branch, which becomes unreachable.

```
1842  if (maxBits > 4 || maxBits < 8) {
1843    maxBits = 8;
1844  }
1845  if (maxBits > 8)
1846    maxBits = 16;
1847  }
```

Figure.2 unreachable code example

To detect unreachable code in examples above, we take Java source code as input. We translate the source code into Java bytecode with Soot, and then further process the bytecode into intermediate verification language with Boogie, which all the runtime assertions are transferred into assertion statements. For each path in the Control flow graph, we generate a path assertion formula with DFS. Then passing the formula to SMT solver to check satisfactory, if satisfied, remove the line number at the path destination from line number set. Repeat check path formulas until all paths are covered. In the end, left line numbers in the line number set will be output as unreachable code segments.

IV. Related work

The goal of unreachable code detection is to find code segments that will never be executed; therefore, we propose the solution using static analysis in combination with Control flow graph analysis. However, there are some other existing tool and approach that detects and analyze unreachable code. Profiling is achieved by instrumenting either the program source code or its binary executable form using a tool called a profiler, which is a kind of dynamic program analysis. To detect unreachable code, combination of simple unreachable criteria and use of a profiler is an approach. [6]

Profiling detects unreachable code through dynamic program analysis instead of static analysis. Dynamic analysis is more precise since it knows the concrete program states by actually executing the program, however this approach still run the unreachable code, which means it still occupy unnecessary memory and decrease data locality. Those are two disadvantages of unreachable code we are trying to solve. The meaning of unreachable code detection would highly decrease if influences are already caused.

Moreover, Profiling could only find potential unreachable code. To prove the correctness of unreachable code, another analysis tool is required. [6] Compared to our approach, which detection is based on path verification, Profiling is not an automated approach.

V. Approach

Unreachable code detection

1) Set up unreachable statement candidates

Initially, we will regard all the statements as potential unreachable statements. For each procedure (one method), we put all the statements in the procedure to a set called unreachable statement candidate set.

2) Path exploration

Explore all possible paths using depth first search for each statement in a procedure and parse each path to be the corresponding SMT solver assertion formula. For example, the path for the statement in line 10 in figure below is a == 3 → b == 4 → (a != 3 || b != 4). The path formula for the statement is a == 3 && b ==4 && (a != 3 || b != 4).

```
4
5   boolean unreachableCode(int a, int b){
6
7       if(a==3){
8           if(b==4){
9               if(a!=3 || b!=4){
10                  return true;
11              }else{
12                  return false;
13              }
14          }
15      }
16  }
17
```

3) Check formulas with SMT solver

Send path formula to the SMT solver to check satisfiability. If the formula is satisfiable, remove the statement from the unreachable statement candidate. Otherwise, the statement stays in the set.

4) Report unreachable statements

After checking satisfactory for all the path formulas, all the statements remain in the unreachable statement candidate set are unreachable statements.

Unreachable code fault-localization

1) Construct prover expression paths for unreachable statements

After detecting all the unreachable statement, We use it as an input to find the fault that cause the unreachability. For every unreachable statement, do backward checking to collect all the prover expressions along the path. Consider the case given above, the unreachable statement is line 10. Therefore, a prover expression path is constructed as (a == 3), (b == 4), (a != 3), (b != 4).

2) Find the closest pair of conditions that cause unreachable

For every prover expression path collected in the above step, we will check from the end of the path. If we find any pair that is unsatifiable, we regard it as the closest pair of conditions that cause the unreachable. More information will be covered in the implementation.

VI. IMPLEMENTATION

Background:

Procedure is a representation of a method in Boogie. A procedure contains a list of basicblock, which contains a set of statements. Basicblock has a reference that point to other basicblock. Therefore, the control flow graph for a method is basically a graph of basicblock)

Code Structure:

1. Under the path of "bixie.checker.inconsistency_checker", a java file called "MyChecker.java" is created to detect unreachable code and localize the fault.
2. The java contains two classes; One is MyNode. The other is MyChecker.
3. Key components in class MyNode: list of "successors", list of "predecessors"
4. Key components in class MyChecker: set of "infeasibleCandidates", map of "contradictSet", method of "runAnalysis", method of "buildGraph", method of "getInfeasibleCodeCandidate", method of "removeCoveredBlock", method of "findContradict", method of "analyzeContradiction", method of "analyzeHelper".
5. Note: Infeasible code = unreachable code. Some variables or some methods' name use infeasible instead of unreachable.

Some key components explanation:

1. MyNode is used to collect all the all the successors and predecessors for each basicblock.
2. Set of "infeasibleCandidates" contains all the potential unreachable candidates initially.
3. "contradictSet" is a map that map unreachable code line number to its corresponding fault statement lines.
4. Method of "runAnalysis" is the backbond of the checker, it maintains the control flow.
5. Method of "buildgrpah" is a method that build a control flow graph in the representation of MyNode objects (Simply use depth first search to collect the information of predecessors and successors)
6. Method of "getInfeasibleCodeCandidate" is a method that add all the unreachableCode candidates to "infeasibleCandidates"
7. Method of "removeCoveredBlock" is a method that use depth first search to explore all the possible path in the control flow graph and push the prover expression for each statement on the path to the prover stack. Then, check whether the prover can give a satisfiable solution. If yes, regard the current statement as reachable, remove it form candidate set and explore more paths. Otherwise, leave the current statement as suspicious (keep it in candidate set) and then return (if current statement is unreachable for the current path, statements after the current statement are not reachable for for the current path).
8. Method of "findContradict" is a method that loop through all the unreachable statements and call "analyzeContradiction" to find fault.
9. Method of "analyzeContradiction" is a method that loop through all the paths that can reach the unreachable statement without consideration of satisfiability and call "analyzeHelper" to do analyze for a given path.
10. Method of "analyzeHelper" is a method that collect all the prover expressions in a path using backward checking (Depth first search in the direction of predecessors). As expected, the path will end at the root statement. Whenever we collect a prover expressions path, we check from the end of the prover expression (from parent of the unreachable statement to root statement) to find whether there is a prover expression ahead of the current's that can form a unsatisfiable pair of prover expressions. If there is, then regard it as the cause of fault. Let us run through a simple example give above. In the case, the only path that can reach the unreachable line (line 10 in the figure above)

is a == 3 → b == 4 → (a != 3 || b != 4). As a result, the prover expression is created in the form of (a == 3), (b == 4), (a != 3), (b != 4). Check it from the end. <1> Send (b != 4) && (a != 3) to the prover → Satifiable. <2> Send (b != 4) && (b == 4) to the prover → Not Satifiable. As mentioned above, whenever it find unsatifiable, regard it as the cause of falt to the current path and then return.

Control flow:

1. We first call "buildgraph" to construct a control flow graph that is in the representation of MyNode, then call getInfeasibleCodeCandidate to get all the unreachable code candidates. After that, "removeCoveredBlock" will be called to eliminate the reachable statements from the candidate set. At this stage, we find the unreachable code for the procedure.
2. Call "findContradict" along with "analyzeContradiction" and "analyzeHelper" to analysis the reason for unreachable and then localize it.

Complications and solution

How to determine whether a statement is unreachable if there are multiple paths reaching it?

A statement is declared to be unreachable only if all the path formulas of it are not satisfiable. Therefore, we could not simply declare the statement is unreachable if a path formula to it is not satisfiable. To solve this problem, we reverse the claim to be a reachable statement must have a path that could reach it, which means there is at least one path formula for it is satisfiable.

Evaluation

Research Questions

There are 3 research claims that we make in the Contribution part. They are

1. We improve unreachable code detection approach, which could detect some common unreachable code that could not be detected by Bixie.
2. We design a unreachable code fault-localization approach for unreachable code that could not be detected by Bixie.
3. We add the feature to show the unreachable code locations, which Bixie doesn't show.

"The research questions need to be proved from those claim is "How much more common unreachable code examples our approach could detect than Bixie?", "How good is fault-localization?" and "Does the unreachable code locations shown meets the actual unreachable code locations, which Bixie doesn't show?"

Hypotheses

If we could find a number of unreachable code examples to run with both Bixie and our approach, and then the detection result shows that our approach correctly detect the unreachable code while Bixie does not detect these unreachable code. The question "How much more common unreachable code examples our approach could detect than Bixie?"could be answered.

If we could find a number of unreachable code examples to run with our approach, and then the location of unreachable cause are accurate. The question "How good is fault-localization?" could be answered.

If we could find a number of unreachable code examples to run with both Bixie and our approach, and then our approach shows the accurate unreachable code location, while Bixie does not provide any unreachable code location information. The question "Does the unreachable code locations shown meets the actual unreachable code locations, which Bixie doesn't show?"could be answered.

Experiment results

To evaluation hypotheses above, we run our project on several example Java source code files. Figure below shows an example unreachable code. The unreachable code is inside the nested if statements. Line 16 is unreachable because Line 15 will always be false after go through the first two if statements.

```
11 public void unreachableCode(int a, int b) {
12     int c = 0;
13     if(a==3){
14         if(b==4){
15             if(a!=3||b!=4){
16                 c = 0;
17             }else{
18                 c = 1;
19             }
20         }
21     }
22 }
```

Figure. example 1 Java source code

The figure below shows the output of example 1 for Bixie, the bwd in last line shows the number of unreachable statements . If unreachable code is detected, it will be printed below the summary. As the figure shows, for example 1, Bixie doesn't find the unreachable code.

```
Soot finished on Fri Dec 18 03:00:49 EST 2015
Soot has run for 0 min. 1 sec.
[root] - Done parsing.
[root] - Checking
[root] - Total time: 2.268s
[root] - Total Timeouts after 30sec: 0 of 6
[root] - Summary: fwd=0 bwd=0
```

Figure. example 1 Unreachable code detection output for Bixie

The figure below shows the output of example 1 for our approach, Since we combine the original output of Bixie with our new approach, you could see the original output of original Bixie below output of our new detection approach. As the figure shows, our approach detects the unreachable code in Line 16, and finds out the causes in Line 13, 14, 15, which is accurate. This experimen result shows our approach find the unreachable code that Bixie doesn't with accurate fault-localization.

```
Soot finished on Fri Dec 18 04:09:56 EST 2015
Soot has run for 0 min. 0 sec.
[root] - Done parsing.
[root] - Checking
Union:
unreachable code:

------------------
Union:
unreachable code: 16
16: 13 14 15

------------------
[root] - Total time: 2.77s
[root] - Total Timeouts after 30sec: 0 of 6
[root] - Summary: fwd=0 bwd=0
```

Figure. example 1 Unreachable code detection output for our aproach

Another nested if statements example is shown below. This one is different from example 1 due to the extra if statement from Line 16 to Line 20. This makes the unreachable code in Line 24 have two paths to reach it. One goes through the true branch of the if statement in Line 16, another goes through the false branch.

```
12      public void unreachableCode(int i) {
13          int c = 0;
14          int a = 8;
15          int b = 9;
16          if(i>0){
17              c = 0;
18          }else{
19              c = 1;
20          }
21          if(a==8){
22              if(b==9){
23                  if(a!=8||b!=9){
24                      c = 0;
25                  }else{
26                      c = 1;
27                  }
28              }
29          }
30      }
```

Figure. example 2 Java source code

As the figure shows, for example 2, Bixie doesn't find the unreachable code.

```
Soot finished on Fri Dec 18 04:47:58 EST 2015
Soot has run for 0 min. 1 sec.
[root] - Done parsing.
[root] - Checking
[root] - Total time: 2.326s
[root] - Total Timeouts after 30sec: 0 of 6
[root] - Summary: fwd=0 bwd=0
```

Figure.example 2 Unreachable code detection output for Bixie

The figure below shows the output of example 2 for our approach, our approach detects the unreachable code in Line 23, and finds the causes in Line 20, 21, 22, which is accurate. This experiment result shows that our approach could still detect unreachable code with multiple reaching path.

```
Soot finished on Fri Dec 18 04:48:09 EST 2015
Soot has run for 0 min. 1 sec.
[root] - Done parsing.
[root] - Checking
Union:
unreachable code:

------------------
Union:
unreachable code: 23
23: 20 21 22

------------------
[root] - Total time: 2.632s
[root] - Total Timeouts after 30sec: 0 of 6
[root] - Summary: fwd=0 bwd=0
```

Figure. example 2 Unreachable code detection output for our approach

To prove our approach could find unreachable code locations, which Bixie doesn't show. We use the unreachable code example that could be detection by both Bixie and our approach. The statement in Line 5 is unreachable because the if statement in Line 13 will always be evaluated to true, which lead to variable maxBit is always equal to 8. Therefore, the if statement in Line 16 will always be evaluated to false.

```
11  public void unreachableCode(int maxBit) {
12
13          if(maxBit > 4 || maxBit < 8){
14              maxBit = 8;
15          }
16          if(maxBit > 8){
17              maxBit = 5;
18          }else{
19              maxBit = 3;
20          }
21      }
22  }
```

Figure. example 3 Java source code

Shown in figure below, the Line number for Bixie is just for unreachable code cause, there is no information indicating the

location of unreachable code. There are some error with the fault-localization of Bixie, which points the cause to the enpty line such as -1 and 21.

```
[root] - Total time: 2.213s
[root] - Total Timeouts after 30sec: 0 of 6
[root] - In File: Demo.java
 - Unreachable -
        Lines: -1, 16
        Lines: 13, 21
Summary: fwd=0  bwd=2
```

Figure. example 3 Unreachable code detection output for Bixie

For our approach shown below, the unreachable statement in Line 17 is accurately located with the correct cause on Line 13, 14, 16. After comparing two outputs, we could conclude that our approach shows the accurate unreachable code location, while Bixie does not provide any unreachable code location information.

```
Union:
unreachable code: 17
17: 16 13 14

------------------
[root] - Total time: 2.188s
[root] - Total Timeouts after 30sec: 0 of 6
[root] - In File: Demo.java
 - Unreachable -
        Lines: -1, 16
        Lines: 13, 21
Summary: fwd=0  bwd=2
```

Figure. example 3 Unreachable code detection output for our approach

Open source testing

We also test our project on the open source project SQLLine. SQLLine is a pure-Java console based utility for connecting to relational databases and executing SQL commands. It is similiar to other command-line database access utilities like sqlplus for Oracle,mysql for MySQL, and isql for Sybase/SQL Server. **The goal of this testing is to prove that false positive rate of our approach is not higher than Bixie.** If the false positive rate of our approach is higher than Bixie, then our approach may shown extra incorrect unreachable code other than result of Bixie when we running a open source project .

It takes some time (5 to 20 minutes depends on CPU) to run Bixie and our approach on Sqlline. Both Bixie and our project don't find the unreachable code in Sqliine.

```
$init$ra$$5906
[root] - Total time: 693.381s
[root] - Total Timeouts after 30sec: 11 of 628
[root] - Summary: fwd=0 bwd=0
```

Figure. Output of Bixie on SQLLine

```
------------------
Union:
unreachable code:

------------------
[root] - Total time: 1233.647s
[root] - Total Timeouts after 30sec: 20 of 628
[root] - Summary: fwd=0 bwd=0
```

Figure. Output of our project on SQLLine

This result shows that our approach doesn't regard the reachable code as unreachable code even in a large open source project, which means it has a relatively high accuracy and false positive rate is not higher than Bixie.

The experiment results in this section provide evidences to show that

- We improve unreachable code detection approach, which could detect some common unreachable code that could not be detected by Bixie.
- We design a unreachable code fault-localization approach for unreachable code that could not be detected by Bixie.
- We add the feature to show the unreachable code locations, which Bixie doesn't show

Conclusion

In this paper, we consider the tool that detects and fault-localizes unreachable code.
Unreachable code is the code segment in a program which will never be executed. We conduct the project to improve Bixie, an unreachable code detection and analysis tool to achieve following contributions.

- We improve unreachable code detection approach, which could detect some common unreachable code that could not be detected by Bixie.
- We design a unreachable code fault-localization approach for unreachable code that could not be detected by Bixie.
- We add the feature to show the unreachable code locations, which Bixie doesn't show

To demonstrate usefulness of our project, we conduct a series experiments on several poplar open source Java program and typical test codes. The goal of our evaluation is to demonstrate that unreachable code exists in real-world popular programs. However, there are some improvements could be done in the future. For future works, we may involve more test cases to test the reliability and accuracy of our project.

Reference

[1] Schäf, Martin, Daniel Schwartz-Narbonne, and Thomas Wies. "Explaining inconsistent code." Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, 2013.

[2] Debray, S. K.; Evans, W., Muth, R., and De Sutter, B. (March 2000). "Compiler techniques for code compaction." (PDF). Volume 22, issue 2. New York, USA: ACM Transactions on Programming Languages & Systems (TOPLAS).

[3]Wiki"Unreachable code" https://en.wikipedia.org/wiki/Unreachable_code#cite_note-1

[4] Eichberg, Michael, et al. "Hidden truths in dead software paths."Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ACM, 2015.

[5] McCarthy, Tim, Philipp Rümmer, and Martin Schäf. "Bixie: Finding and Understanding Inconsistent Code."

[6]Wiki"Profiling" https://en.wikipedia.org/wiki/Profiling_(computer_programming)

[7] Arlt, Stephan, and Martin Schäf. "Joogie: Infeasible code detection for java."*Computer Aided Verification*. Springer Berlin Heidelberg, 2012.

[8] Vallée-Rai, Raja, et al. "Soot-a Java bytecode optimization framework."*Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1999.

[9] Tim Lindholm and Frank Yellin. "*The Java Virtual Machine Specification*." Addison-Wesley, Reading, MA, USA, second edition, 1999