Understanding Programming Bugs in Java Programs Using Counterexamples
-Optimizing minimal counterexamples

Zhaoyu Zhang

- Identify and locate programming errors
- Produce counterexample of current problem
- Ensure correctness of software systems by understanding the identified error

Usage

Input: *.class* file or *.jar* (Java ARchive) file
Output: Errors, Transitions, Snapshots, Results and Data Statistics

Problem:

BMC tools often produce counterexamples that are either too large or difficult to be understood mainly because of the software size and the values chosen by the respective solver.

Objectives:

• Develop a method to automatically collect and manipulate counterexamples produced by a BMC tool to generate new instantiated code to reproduce the identified error.

• Employ a Bounded Model Checker for Java Bytecode called JBMC to show the effectiveness of the proposed method over publicly available benchmarks from the Software Verification Competition (SV-COMP)

Existing BMC for Java Bytecode	logic/Method Used	Holds for	SMT/SAT Solvers	Counter example generation
Java PathFinder (JPF)	performs an optimized explicit-state model check for program properties DFS algorithm[3] to store states and backtrack	Invariants,Deadlocks, Race conditions, Static and Runtime	Unknown	Yes
JayHorn	Verifying annotated Java programs with assertions expressing safety conditions by using Horn clauses[4]	Single Thread, Static Classes, Synchronization	Soot, SPACER, Eldarica	Yes
Extended Static Checking for Java (ESC/Java)	examines inconsistencies between the design decisions and the actual code, also warns of potential runtime errors[5]	Static,compile-time, Deadlocks, Race conditions,	PREfix[1]	Yes
Java Bounded Model Checker(JBMC)	processes Java bytecode together with a model of the standard Java libraries and checks a set of desired properties[6]	Synchronization, Runtime, Overflow, violations	CPROVER, CBMC	Yes
Bandera	Enables the aotumatic extraction of safe, compact finite-state transition models[7]	Synchronization, Runtime	Soot/Jimple[2]	Yes

References:
[1]W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. SP&E, 30(7):775–802, June 2000.
[2]R. V alle-Rai, L. Hendren, V. Sundaresan, P . Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. InProceedings of CASCON'99, Nov. 1999.
[3]Brat, Guillaume & Havelund, Klaus & Visser, Willem. (2000). Java PathFinder - Second Generation of a Java Model Checker.
[4]Rümmer, Philipp. (2019). JayHorn: a Java model checker. FTfJP '19: Proceedings of the 21st Workshop on Formal Techniques for Java-like Programs. 1-1. 10.1145/3340672.3341113.
[5]Nelson, Greg. (2004). Extended Static Checking for Java. Sigplan Notices - SIGPLAN. 37. 1-1. 10.1007/978-3-540-27764-4_1.
[6]Cordeiro, Lucas & Kroening, Daniel & Schrammel, Peter. (2019). JBMC: Bounded Model Checking for Java Bytecode: (Competition Contribution). 10.1007/978-3-030-17502-3_17.
[7]Corbett, James & Dwyer, Matthew & Hatcliff, John & Laubach, Shawn & Pasareanu, C.S. & Apriadi, Robby & Zheng, Hongjun. (2000). Bandera: Extracting finite-state models from Java source code. Proceedings - International Conference on Software Engineering. 439 - 448. 10.1109/ICSE.2000.870434.

```java
public class Racer implements Runnable {

    int d = 42;

    @Override
    public void run () {
        doSomething( n: 1001);                    // (1)
        d = 0;                                    // (2)
    }


    public static void main (String[] args){
        Racer racer = new Racer();
        Thread t = new Thread(racer);
        t.start();

        doSomething( n: 1000);                    // (3)
        int c = 420 / racer.d;         // (4)
        System.out.println(c);
    }


    static void doSomething (int n) {
        // not very interesting..
        try { Thread.sleep(n); } catch (InterruptedException ix) {}
    }
}
```

JPF JVM states:

- **State backtracking**: explore the executions for all values in the range. JPF implements this by storing and restoring the entire JVM state of the program being checked
- **State comparison**: performs a stateful search and stops an execution path if it encounters a previously seen state
- **State representation**: encodes each object simply as a Java integer array (int[ ]) and based on type information, interprets a field as either a primitive value or a pointer to another object. JPF encodes the entire heap effectively as an array of integer arrays (int[ ][ ] in Java).
- **Bytecode execution**: provides classes that implement the semantics of Java bytecodes by manipulating the special state representation, whose goal is to make the overall exploration fast even if it makes one straight-line execution path slow
- **Model Java Interface (MJI)**: provides a mechanism for executing parts of application code on the host JVM. MJI allows the host JVM to manipulate the JPF state representation, e.g., to read or write field values or to create new objects.

Every path in the abstracted program where all assignments are deterministic has a corresponding path in the concrete (unabstracted) program[1]

if an abstract system is deterministic, then it is equivalent to the concrete system
 (i.e., there is a simulation equivalence between the concrete and abstract systems)

every deterministic abstract path has a corresponding path in the concrete program

Reference:
[1](Theorem 5)Sa¨ıdi, H.: Model checking guided abstraction and analysis. In: Palsberg, J. (ed.), Proc. 7th International Static Analy_x0002_sis Symposium (SAS'00), Lecture Notes in Computer Science,
vol. 1824. Springer, Berlin Heidelberg New York, 2000, pp.377–396

```java
public class Signs {
  public static final int NEG =0;
  public static final int ZERO=1;
  public static final int POS =2;
  public static int abs(int n) {
    if (n < 0) return NEG;
    if (n == 0) return ZERO;
    if (n > 0) return POS;
  }
}
```

```java
public static int add(int a, int b) {
  int r;
  Verify.beginAtomic();
  if       (a==NEG  && b==NEG)  r=NEG;
  else if (a==NEG   && b==ZERO) r=NEG;
  else if (a==ZERO && b==NEG)   r=NEG;
  else if (a==ZERO && b==ZERO)  r=ZERO;
  else if (a==ZERO && b==POS)   r=POS;
  else if (a==POS  && b==ZERO)  r=POS;
  else if (a==POS  && b==POS)   r=POS;
  else r=Verify.choose(7);
  Verify.endAtomic();
  return r;
}
```

Java representation of the signs AI(Abstract Functions)

Abstract tokens are implemented as integer values, and the abstraction function and operations have straightforward implementations as Java methods

```
class App {              |    class App {
                              in(...) {
                              ();
    }                    |        }
}                        |    }
class AThread extends Thread {   class AThread extends Thread {
  public void run() {            public void run() {
    ...                            ...
[6]    Global.done=true;    [6]    Global.done=true;
    }                              }
}                        |    }
```

The counter-example does not contain nondeterministic choices since the value returned by Signs.lt(i,Signs.POS), when i is zero, is uniquely true.

nondeterminism is ...d when ...ts abstract ...on for integer <

At First it is true

i becomes POS,
so Signs.lt(i,Signs.POS)
can return true or false

Steps in the path contain additional information, such as the class name and line number of the executing thread, that can be used to determine the execution context. When executing the concrete system using the error-path generated from analysis of the abstract system we check whether the class name and line number expected by the error path is matched by the execution in the system. A mismatch indicates that the abstract path is not feasible in the concrete system.

- sel...
- typ...
- the... ...ed from
  ab...
- fin... ...with calls
  to...
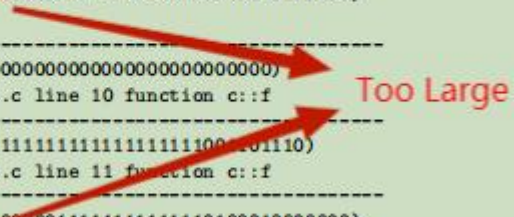
- A model checking counterexample is intended to be read by a person and used for debugging.

- Ideally, such a counterexample would be the most succinct and easily comprehensible witness to the existence of an error. The utility of small (in various senses, including length) counterexamples is widely recognized.

- Previous work on minimization of counterexamples has concentrated either on producing counterexamples of minimal length or on removing irrelevant information from a counterexample.

- Proposed a technique that can be used to minimize counterexample length, but focuses on a semantic minimization, with respect to the type system of the language
- => minimizes the values of variables in the counterexample

```
Counterexample:
Initial State
------------------------------------------------
State 1
------------------------------------------------
  a=-8193 (11111111111111111101111111111111)
State 2
------------------------------------------------
  b=-402 (11111111111111111111111001101110)
State 3
------------------------------------------------
  c=-2080380800 (10000011111111111110100010000000)
State 4
------------------------------------------------
  temp=0 (00000000000000000000000000000000)
State 10 file sort.c line 10 function c::f          Too Large
------------------------------------------------
  temp=-402 (11111111111111111111111001101110)
State 11 file sort.c line 11 function c::f
------------------------------------------------
  b=-2080380800 (10000011111111111110100010000000)
State 12 file sort.c line 12 function c::f
------------------------------------------------
  c=-402 (11111111111111111111111001101110)
Failed assertion: assertion file sort.c line 19 function c::f
```

A very unfortunate choice for the initial value

Reason of Large Value Variables in Counter Examples:

The SAT solvers used to check these formulas for satisfiability typically return the first satisfying assignment produced. The counterexample values, therefore, are highly dependent on the decision heuristics used by the SAT solver. These choices may result in needlessly large values for the actual program variables

Abstract counter-example makes it easier to collect counter-example,

then we want to minimize the counter-example produced to generate new instantiated code to reproduce the identified error from a smaller value of detected error-related variables.

- Minimization of counterexample values can be considered as a 0-1 ILP(Integer Linear Programming) problem
- A pseudo-Boolean constraint solver(PBS) which, given a SAT instance in CNF and a set of integer coefficients for SAT variables, will solve optimization problems over the constraints
- The length of the counterexample is minimized before values are minimized. Each guard bit is given a weight equal to the number of program statements guarded by that condition.
- The psuedo-Boolean optimization problem is to minimize the weighted sum (the number of executed program statements).
- Counterexample length minimization is completed and guard values are locked before value minimization begins.

```java
public static void function(int a, int b, int c){
    int temp;
    if (a > b) {
        temp = a;
        a = b;
        b = temp;
    }
    if (b > c) {
        temp = b;
        b = c;
        c = temp;
    }
    if (a < b) {
        temp = a;
        a = b;
        b = temp;
    }
    assertTrue( condition: (a <= b) && (b <= c));}
```

TODO:
Minimize the sum of |a| + |b| + |c| + |temp|

Theory:
each of these variables may take on different values during execution of the program. Therefore, the sum that is minimized is over all program variables after loop unrolling and static single assignment(SSA)

Actual:
Minimize the sum of |a#0| + |a#1| + |a#2| + |a#3| + |a#4| + |b#0| + ... + |b#6| + |c#0| + |c#1| + |c#2| + |temp#0| + ... + |temp#6|

```
public static void function(int a, int b, int c){
    int temp;
    if (a > b) {
        temp = a;
        a = b;
        b = temp;
    }
    if (b > c) {
        temp = b;
        b = c;
        c = temp;
    }
    if (a < b) {
        temp = a;
        a = b;
        b = temp;
    }
    assertTrue( condition: (a <= b) && (b <= c));}
```

```
Counterexample:
Initial State
-------------------------------
```
Original: has large values
```
Counterexample:
Initial State
-------------------------------

temp=-1 (11111111111111111111111111111111)
a=0 (00000000000000000000000000000000)
b=0 (00000000000000000000000000000000)
```
Minimized:
```
c=-1 (11111111111111111111111111111111)
State 6 file test.java line 10
```
no large value
```
-------------------------------
temp=0 (00000000000000000000000000000000)
State 7 file test.java line 11
-------------------------------
b=-1 (11111111111111111111111111111111)
State 8 file test.java line 12
-------------------------------
c=0 (00000000000000000000000000000000)
Failed assertion: assertion file test.java line 19
-------------------------------
State 12 file test.java line 12
-------------------------------
c=-402 (11111111111111111111111111001101110)
Failed assertion: assertion file test.java line 19
```

Experiment ID	Normal Result			Minimized Result		
	Time(s)	Sum of Variables	length in steps	Time(s)	Sum of Variables	length in steps
test1.java (sorting)	0.70	4.161×10^9	7	25.72	2	7
test2.java (Race Condition)	1.30	111,905	13	13.35	9,734	13
test3.java (Deadlock)	1.20	747,623	25	14.55	9,524	25

- Sum of variables significantly reduces
- length in steps of the counterexample does not change
- Running time(JPF build time) increases, depends on code logic

```
arg0i=1 (00000000 00000000 00000000 00000001)

State 113 file Main.java function Main.function(int, int, int) line 17 thread 0
----------------------------------------------------------
  arg1i=0 (00000000 00000000 00000000 00000000)

State 121 file Main.java function Main.function(int, int, int) line 19 thread 0
----------------------------------------------------------
  dynamic_object13={ .@class_identifier="java::java.lang.Class" } ({ ? })

State 122 file Main.java function Main.function(int, int, int) line 19 thread 0
----------------------------------------------------------
  dynamic_object13.@class_identifi

State 124 file Main.java function
----------------------------------------------------------
  dynamic_object13.@class_identifi

State 127 file Main.java function
----------------------------------------------------------
  this=&dynamic_object13 (00000000 00010000 00000000 00000000 00000000 00000000 00000000 00000000)

Violated property:
  file Main.java function Main.function(int, int, int) line 19 thread 0
  assertion at file Main.java line 19 function java::Main.function:(III)V bytecode-index 38
  false


** 1 of 14 failed (2 iterations)
VERIFICATION FAILED
```

```
arg0i=1 (00000000 00000000 00000000 00000001)

State 94 file Main.java function Main.function(int, int, int) line 17 thread 0
----------------------------------------------------------
  arg1i=0 (00000000 00000000 00000000 00000000)

State 102 file Main.java function Main.function(int, int, int) line 19 thread 0
----------------------------------------------------------
  dynamic_object13={ .@class_identifier="java::java.lang.Class" } ({ ? })

State 103 file Main.java function Main.function(int, int, int) line 19 thread 0
----------------------------------------------------------

  this=&dynamic_object13 (00000000 00001111 00000000 00000000 00000000 00000000 00000000 00000000)

Violated property:
  file Main.java function Main.function(int, int, int) line 19 thread 0
  assertion at file Main.java line 19 function java::Main.function:(III)V bytecode-index 38
  false


** 1 of 13 failed (2 iterations)
VERIFICATION FAILED
```

- All benchmarks were checked by employing JBMC using proposed method
- Number of states changes from 127 to 108
- Sum of variables changes from 164 to 23
- Elapsed Time changes from 0.0109139s to 0.0101328s

- minimizing algorithm are considerably slower than plain BMC without minimization

- investigate SAT solvers with decision heuristics that are aware of a metric for counterexample simplicity: the idea being to make favoring simple counterexamples a part of the search algorithm, instead of a post-processing step.

- For programs does not include linear algebra logics

THANKS FOR WATCHING !