CSE P 590
Beyond Coverage: Modern Testing and Debugging
Spring 2019

Formal reasoning & constraint-based testing

May 14, 2019
Today

- Mutation-based testing discussion
- Formal reasoning and Z3
- In-class 5
Mutation-based testing: recap and discussion
Recap: mutation testing

Program

Mutants

Tests

lhs < rhs  ➡️  lhs <= rhs

lhs < rhs  ➡️  lhs != rhs

stmt  ➡️  no-op
Recap: not all mutants are effective

A mutant is not effective if:
- it cannot be detected (semantically **equivalent**)
- it fails for any given test (**trivial**)
- it is dominated by other mutants (**subsumed**)

Desirable tests
Recap: killable vs. productive mutants

Status quo
- Killable mutants are **good** $\implies$ tests
- Equivalent mutants are **bad** $\implies$ no tests

A more nuanced view
- Killable vs. equivalent is too simplistic
- **Productive mutants** elicit effective tests, but
  - Killable mutants can be unproductive
  - Equivalent mutants can be productive

The notion of productive mutants is fuzzy!

A mutant is **productive** if it is
1. killable and **elicits an effective test** or
2. equivalent and advances code quality or knowledge
In-class discussion: mutation-based testing

Effectiveness of mutation-based testing
- Do you see value in mutation-based testing?
- What mutants did you consider unproductive?
- When and how would you apply mutation-based testing?

Reasoning about mutants
- How did you reason about equivalence?
- What was challenging about this task?
Formal reasoning and Z3
Reasoning about programs

Use cases
● Verification: ensure code is correct
● Testing: constraint-based test generation
● Debugging: understand why code is incorrect

Prove facts to be true about a program, e.g.:
● $x$ is never null
● $y$ is always greater than 0
● input array $a$ is sorted
● $x + y > z$
Forward vs. backward reasoning

Forward reasoning

- Knowing a fact that is true before execution.
- Reasoning about what must be true after execution.
- Given a precondition, what postcondition(s) are true?

Backward reasoning

- Knowing a fact that is true after execution.
- Reasoning about what must be true before execution.
- Given a postcondition, what precondition(s) must hold?
What is Z3?

- An SMT (Satisfiability Modulo Theories) solver.
- Uses a standard language (SMT-LIB).
  - Print to the screen.
  - Declare variables and functions.
  - Define constraints.
  - Check satisfiability and obtain a model.
  - ...

```lisp
(echo "Running Z3...")
(declare-const a Int)
(assert (> a 0))
(check-sat)
(get-model)
```
What is Z3?

- An SMT (Satisfiability Modulo Theories) solver.
- Uses a standard language (SMT-LIB).
  - Print to the screen.
  - Declare variables and functions.
  - Define constraints.
  - Check satisfiability and obtain a model.
  - ...

```lisp
(echo "Running Z3...")
(declare-const a Int)
(assert (> a 0))
(check-sat)
(get-model)
```

This is equivalent to asking the question: Does an integer greater than 0 exist?
A first example

```cpp
int simpleMath(int a, int b) {
    assert(b>0);
    if(a + b < a) {
        return 1;
    }
    return 0;
}
```

Does this method ever return 1?
A first example

```c
int simpleMath(int a, int b) {
    assert(b>0);
    if(a + b < a) {
        return 1;
    }
    return 0;
}
```

(declare-const a Int)
(declare-const b Int)
(assert (> b 0))
(assert (< (+ a b) a))
(check-sat)

Does this method ever return 1? Let’s ask Z3...
A first example: correctly modeling data types

```c
int simpleMath(int a, int b) {
    assert(b>0);
    if(a + b < a) {
        return 1;
    }
    return 0;
}
```

Z3 supports Bitvectors of arbitrary size. Let's model Java ints (32 bits) and ask the same question...

```z3
(declare-const a (_ BitVec 32))
(declare-const b (_ BitVec 32))
(assert (bvsgt b #x00000000))
(assert (bvslt (bvadd a b) a))
(check-sat)
(get-model)
```
A more complex example: modeling control flow

```c
int doesStuff(int a, int b, int c) {
    if (c==0) return 0;
    if (c==4) return 0;
    if (a + b < c) return 1;
    if (a + b > c) return 2;
    if (a * b == c) return 3;
    return 4;
}
```

Does this method ever return 3?
A more complex example: modeling control flow

```c
int doesStuff(int a, int b, int c) {
    if (c==0) return 0;
    if (c==4) return 0;
    if (a + b <  c) return 1;
    if (a + b >  c) return 2;
    if (a * b == c) return 3;
    return 4;
}
```

All of the following must be true (why?):
- !(c == 0)
- !(c == 4)
- !(a + b < c)
- !(a + b > c)
- a * b == c

Does this method ever return 3?
A more complex example: modeling control flow

```c
int doesStuff(int a, int b, int c) {
    if (c==0) return 0;
    if (c==4) return 0;
    if (a + b < c) return 1;
    if (a + b > c) return 2;
    if (a * b == c) return 3;
    return 4;
}
```

```lisp
(define-sort JInt () (_ BitVec 32))
(declare-const a JInt)
(declare-const b JInt)
(declare-const c JInt)
(assert (not (= c #x00000000)))
(assert (not (= c #x00000004)))
(assert (not (bvslt (bvadd a b) c)))
(assert (not (bvsigt (bvadd a b) c)))
(assert (= (bvmul a b) c))
(check-sat)
(get-model)
```
Reasoning about program equivalence

```c
int originalSum(int a, int b) {
    return a + b;
}

int mutatedSum(int a, int b) {
    return a * b;
}
```

Are these two methods equivalent?
Reasoning about program equivalence

Are these two methods equivalent?
Reasoning about program equivalence

```java
int originalSum(int a, int b) {
    return a + b;
}

int mutatedSum(int a, int b) {
    return a * b;
}
```

```prolog
(declare-const a Int)
(declare-const b Int)
(declare-const rOrig Int)
(declare-const rMut Int)
(assert (= rOrig (+ a b)))
(assert (= rMut (* a b)))
(assert (= rOrig rMut))
(check-sat)
(get-model)
```

Yes, for a=0 and b=0.
What have we actually proven here?
Reasoning about program equivalence

```c
int originalSum(int a, int b) {
    return a + b;
}

int mutatedSum(int a, int b) {
    return a * b;
}
```

Our goal is to prove that no model for equivalence exists (i.e., the defined constraints are unsatisfiable)!