

CS 520

In-Class 4

Formal Reasoning

Due: **Thursday, November 21, 2019, 9:00 AM EST** via [Moodle](#). This in-class exercise is a group submission. This means that **each group only needs to submit their solution once** and also that every student in a group will get the same grade. You will work with students within your group, but not with students from other groups. Multiple groups' submissions may not be created jointly. Late assignments will not be accepted without **prior** permission.

Overview and goal

The high-level goal of this exercise is to learn to use theorem provers to reason about code and prove code properties. The exercise will also teach how to generate test cases via formal reasoning with theorem provers.

Forming groups

1. Team up in groups of size 4. (If you cannot find a 4th member, raise your hand and ask the instructor.)
2. Create a new group on Moodle (see “In-class exercise 4: group selection”), and add all group members.

Background

Do you recall the Triangle program? You wrote tests for it the in-class 2 exercise and then used coverage and mutation testing to evaluate your test suites. It was especially hard to write tests for several of the mutants, so in this assignment, we will use formal reasoning to prove whether such tests could be written.

Specifically, we'll look at two mutants (but first, one simpler program to get us started). We'll write a **logical formula** that describes the behavior of the relevant part of the original (not mutated) program, and another logical formula that describes the behavior of the relevant part of the mutant. We'll then use the [Z3](#) theorem prover to ask the question “Does an input exist such that the mutant and the original program exhibit different behavior?” If it does, we'll use what Z3 tells us to create a test case. The important thing to remember is that, as we learned in in-class 2, writing tests that detect mutants can be difficult. Our goal today is to use the Z3 theorem prover to do this difficult work for us.

The key to this assignment is using [Z3](#). You can always refer to the [Z3 tutorial](#): <https://rise4fun.com/Z3/tutorial/guide> if you need help using it. But we'll start out here with information on how to use Z3.

What is Z3?

Here's an example Boolean formula:

$$(x_1 \vee x_2) \wedge (\neg x_1 \vee x_2)$$

It says “ $(x_1 \text{ or } x_2)$ and $((\text{not } x_1) \text{ or } x_2)$ ”. Let's call that formula ϕ . Given a Boolean formula ϕ without quantifiers (just *ands* (\wedge), *ors* (\vee), and *nots* (\neg)), we can assign its variables to Boolean values and ϕ will evaluate to `true` or `false`. (We will write 1 for `true` and 0 for `false`.) For example, if $x_1 = 0$ and $x_2 = 0$, then $\phi = (0 \vee 0) \wedge (\neg 0 \vee 0) = (0) \wedge (1) = 0$. What Z3 does is take a Boolean formula in a particular format and compute whether there exists some assignment to the variables that makes the entire formula `true`. If

such an assignment exists, then we say the formula is **satisfiable**. If all assignments make our formula `true`, we say that the formula is a **tautology**. If *no* assignment satisfies our formula then we say that our formula is **unsatisfiable**. Note that if ϕ is a tautology, then we know that $\neg\phi$ is unsatisfiable. We'll use this fact later.

The question “Is formula ϕ satisfiable?” turns out to be hard in general¹ but Z3 uses *heuristics* to solve exactly this problem for us. The challenge in front of us is to encode the questions we care about into this formulation Z3 can help us with.

Z3 allows users to not only ask questions about Boolean formulae but also about certain math equations by introducing things like *existential* and *universal quantification*, *numbers*, and *linear arithmetic* (addition, subtraction, multiplication, division, and ordering).

Z3 for Program Verification

So how can we use Z3 for program verification? Well, say we have a program P and a question about P . We can encode the question and the relevant parts of the program into **constraints**. If we do it just right, Z3 can tell us if the set of constraints is satisfiable and answer our question. To tell Z3 about a constraint we use an `assert` statement.

Take, for example, a program

```
1      int P(int a, int b){
2          return a + b;
3      }
```

and let's say the question we want answered is “Are there inputs such that P returns 0?”

We would write Z3 constraints as follows: First, we declare a and b . We then assert that the return value is 0 (to do this we can assert that $a + b == 0$). Then, we ask Z3 if this is satisfiable by running (`check-sat`). To see what Z3 came up with, we use the command (`get-model`). Let's try that. Go to <https://rise4fun.com/Z3> and type in (without the line numbers):

```
1 (declare-const a Int)
2 (declare-const b Int)
3 (assert (= (+ a b) 0))      ; We want a + b to be 0
4 (check-sat)                 ; Find out if this is satisfiable
5 (get-model)                 ; It is, so let's get a satisfying model
```

Z3 then returns

```
sat
(model
  (define-fun b () Int
    0)
  (define-fun a () Int
    0)
)
```

which tells us that when $a=0$ and $b=0$, the constraints are satisfied. Great, we just used Z3 to generate a test!

¹Satisfiability is an NP-complete problem, meaning, among other things, that we only know how to solve it in time exponential in the number of variables. If you haven't yet, you'll encounter NP-completeness in other classes, but you don't have to worry about it here.

Modeling Control Flow

Let's deal with a more complicated program. Say we have the following Java method and we want to know if it can ever return 3.

```
1
2 int doesStuff(int a, int b, int c){
3     if (c == 0) return 0;
4     if (c == 4) return 0;
5     if (a + b < c) return 1;
6     if (a + b > c) return 2;
7     if (a * b == c) return 3; // Does this ever happen??
8     return 4;
9 }
```

Z3 can do this for us, no problem! Here are the constraints² (we'll explain what they do right after):

```
1 (define-sort JInt () (_ BitVec 32)) ; For convenience we alias the 32 bit vector
2 (declare-const a JInt)
3 (declare-const b JInt)
4 (declare-const c JInt)
5
6 (assert (not (= c #x00000000)))
7 (assert (not (= c #x00000004)))
8 (assert (not (bvslt (bvadd a b) c)))
9 (assert (not (bvsgt (bvadd a b) c)))
10 (assert (= (bvmul a b) c))
11
12 (check-sat)
13 (get-model)
```

So why does this ask Z3 if our method ever returns 3? Well returning 3 is equivalent to saying that none of the other if statement conditionals were true but that the if-statement conditional $a * b == c$ was true. Thus we assert that $c \neq 0$, $c \neq 4$, that $a + b > c$, that $a + b < c$, and finally that $a * b == c$.

Cool, let's see Z3's answer:

```
sat
(model
  (define-fun c () (_ BitVec 32)
    #x2da77000)
  (define-fun a () (_ BitVec 32)
    #x252cc8c0)
  (define-fun b () (_ BitVec 32)
    #x087aa740)
)
```

While this is *technically* a valid answer with Java ints, it is actually an overflow error. To verify this, try the same Z3 script with all instances of JInts replaced with Ints (and replace the operators as well). This will return unknown which means that Z3 can't find an answer.

²These constraints use something called a bitvector (BitVec 32), which is an accurate way to define Java ints, as opposed to the abstract concept of an integer. Don't worry about what that means for now, it's just another way to represent integers. If you'd like, you can read about bitvectors here: <https://rise4fun.com/Z3/tutorialcontent/guide#h25>. But do note that with bitvectors, operators such as + are replaced with bvadd, * with bvmul, < with bvslt, > with bvsgt, etc.

Z3 for Mutant Detection

OK, now let's finally talk about mutation testing. What if we want to find out if two programs are equivalent? Consider the following two programs.

```
1  int normal_sum(int a, int b){
2      return a + b;
3  }
4
5  int mutant_sum(int a, int b){
6      return a * b;
7  }
```

We want to know if these are equivalent; that means we want to know if they *always* produce the same output on the same inputs. Let's try to ask Z3 if they are the same:

```
1 (declare-const a Int)
2 (declare-const b Int)
3 (assert (= (+ a b) (* a b)))
4 (check-sat)
5 (get-model)
```

Running Z3 we get

```
sat
(model
  (define-fun b () Int
    0)
  (define-fun a () Int
    0)
)
```

Z3 said sat??? But these are totally not the same programs! On inputs $a = 1$, $b = 1$ they don't produce the same results!

Remember that Z3 tells us if there exists an assignment that satisfies the constraints. Here, it told us that when $a = 0$, $b = 0$, these two programs produce the same answer. That's all. It doesn't tell us if the constraints are **always** true.

So how do we ask Z3 that question? Well, recall that *something is a tautology if and only if its negation is unsatisfiable*. We can use this fact. We can say that `normal_sum` is equivalent to `mutant_sum` if and only if the following statement is true: "there does not exist an input on which their outputs differ." And we can check this with Z3 by asserting that `normal_sum` is **not** equivalent to `mutant_sum`, and then asking Z3 if this is satisfiable. If it is, then we can *separate* these two programs and they are not equivalent; otherwise, if this assertion is unsatisfiable, we can conclude that there is no input on which the two programs differ. Let's do that:

```
1 (declare-const a Int)
2 (declare-const b Int)
3 (assert (not (= (+ a b) (* a b))))
4 (check-sat)
5 (get-model)
```

which outputs

```

sat
(model
  (define-fun b () Int
    (- 5))
  (define-fun a () Int
    (- 4))
)

```

Thus Z3 has found a test to show that the two programs are not equivalent.

As a final example, let's see what happens when two mutants are the same. Consider the following two methods

```

1      int normal(int a){
2          if (a <= 0) return -1;
3          if (a > 0) return 1;
4      }
5      int mutant(int a){
6          if (a <= 0) return -1;
7          if (a >= 0) return 1;
8      }

```

How can we model this with Z3 code? Well, we want to track the different possible return values for each of the methods. We declare `normal-returns-neg-1`, `normal-returns-pos-1`, `mutant-returns-neg-1`, and `mutant-returns-pos-1`. Then we can use assertions to add constraints to the system to model our methods: `(assert (= normal-returns-neg-1 (<= a 0)))` tells Z3 that normal only return -1 if $a \leq 0$.

Here is our complete Z3 script:

```

1 (declare-const a Int)      ; The input to be passed to both normal and mutant
2
3 ;; Declare all possible returns
4 (declare-const normal-returns-neg-1 bool)
5 (declare-const normal-returns-pos-1 bool)
6 (declare-const mutant-returns-neg-1 bool)
7 (declare-const mutant-returns-pos-1 bool)
8
9 ;; Use asserts to constrain how each return can be reached
10 (assert (= normal-returns-neg-1 (<= a 0)))
11 (assert (= mutant-returns-neg-1 (<= a 0)))
12
13 (assert (= normal-returns-pos-1
14           (and (not normal-returns-neg-1)
15                (> a 0))))
16
17 (assert (= mutant-returns-pos-1
18           (and (not mutant-returns-neg-1)
19                (>= a 0))))
20
21 (assert (not (= normal-returns-neg-1 mutant-returns-neg-1)))
22 (assert (not (= normal-returns-pos-1 mutant-returns-pos-1)))
23 (check-sat)
24 ;; This yields 'unsat'

```

Questions

You will be given three pairs of programs. Download them here:

<http://people.cs.umass.edu/~brun/class/2019Fall/CS520/in-class4.programs.zip>

For each pair, you'll find three files:

1. A program (`Sum.java` or `Triangle.java`).
2. A mutant (`Mutant.java`). Note that `javac` won't compile this class cause of the filename.
3. A starter script with helpful Z3 commands encoding constraints you'll need (`Z3startercode.pairN.smt2`).
Look for the

```
;;;;;;;;;;;;; START STUDENT CODE ;;;;;;;;;;;;;;  
;;;;;;;;;;;;; END STUDENT CODE ;;;;;;;;;;;;;;
```

tags for where to write your code. You won't have to edit anything outside of these tags, except possibly uncommenting the penultimate `;; (get-model)` line.

For each pair, your job is to determine if there exists a test that produces different outputs on the two programs in the pair (and, if so, what that test is), or if the programs are equivalent (and thus no such test exists). Start with `Z3startercode.pairN.smt2` and fill in the space that's labeled for you to fill in.

Deliverables

Your submission, via [Moodle](#), must be a single (one per group) archive (`.zip` or `.tar.gz`) file with name `<group name>-inclass4.<zip/tar.gz>`, containing:

1. `Z3startercode.pair1.smt2`
2. `Z3startercode.pair2.smt2`
3. `Z3startercode.pair3.smt2`
4. `writeup.txt`

Each of the `.smt2` files should contain your Z3 code solution. Verify that pasting the **entire file** into the [Z3](#) interface produces the output you expect.

The `writeup.txt` file should contain, for each of the three pairs, an explanation of either how you know the mutants are equivalent, or a test case that demonstrates that they are not equivalent.