CS 520
Theory and Practice of Software Engineering
Spring 2022
Reasoning about programs
April 26, 2022
What is Software Engineering?

More than just writing code
The complete process of specifying, designing, developing, analyzing, deploying, and maintaining a software system.

Common Software Engineering tasks include:

- Requirements engineering
- Software architecture and design
- Programming
- Verification & Validation
- Debugging
What is Software Engineering?

More than just writing code
The complete process of specifying, designing, developing, analyzing, deploying, and maintaining a software system.

Common Software Engineering tasks include:

- Requirements engineering
- Software architecture and design
- Programming
- Verification & Validation
- Debugging
Another way to verify your code

• Prove that the system does what you want
  – Representation (rep) invariants are preserved
  – Implementation satisfies specification

• Proof can be informal or formal (e.g., theorem prover)

• Complementary to manual and automated reasoning (e.g., code review, testing, model checking)
Reasoning about code

Determine what facts are true during system execution, e.g.,

- $x > 0$
- for all nodes $n$: $n\text{.next}\text{.previous} == n$
- array $a$ is sorted
- $x + y == z$
- if $x \neq \text{null}$, then $x\text{.a} > x\text{.b}$
Possible uses of such facts

• **Verification & Validation:** Ensure code is correct (via reasoning or testing)

• **Debugging:** Understand why code is incorrect (e.g., assertion violations)
int y = 100;
for (int x = 0; x < 100; x++)
    y--;
int y = 100;
for (int x = 0; x < 100; x++)
    y--;

What is true about the above?
• Before loop: $x = 0$, $y = 100$
• During loop: Each iteration $x$ increases by 1 and $y$ decreases by 1
• After loop: $x = 100$, $y = 0$
Forward reasoning

• **Key idea:**
  – You know what is true before running the code. *What is true after running the code?*
  – Given a precondition, what is the postcondition?

• **Possible uses:**
  Rep invariant holds before running code
  Does it still hold after running code?

• **Example:**
  // precondition: x is even
  x = x + 3;
  y = 2x;
  x = 5;
  // postcondition: ??
Forward reasoning example

// precondition: x is even
x = x + 3;
// ??

y = 2x;
// ??

x = 5;
// postcondition: ??
Forward reasoning example

// precondition: x is even
x = x + 3;
// x is odd
y = 2x;
// ??
x = 5;
// postcondition: ??
Forward reasoning example

// precondition: x is even
x = x + 3;

// x is odd
y = 2x;

// y is even and thus divisible by 2 (but not by 4)
x = 5;

// postcondition: ??
Forward reasoning example

// precondition: x is even
x = x + 3;

// x is odd
y = 2x;

// y is even and thus divisible by 2 (but not by 4)
x = 5;

// postcondition: x = 5, y is divisible by 2 (but not by 4)
Advantages of forward reasoning

• More intuitive for most people
  – Helps understand what will happen (simulates the code)
  – Introduces facts that may be irrelevant to goal
    Set of current facts may get large
  – Takes longer to realize that the task is hopeless
Backward reasoning

• **Key idea:**
  – You know what you want to be true after running the code. What must be true beforehand in order to ensure that?
  – Given a postcondition, what is the corresponding precondition?

• **Possible uses:**
  (Re-)establish rep invariant at method exit: what’s required?
  Reproduce a bug: what must the input have been?

• **Example:**
  ```
  // precondition: ??
  x = x + 3;
  y = 2x;
  x = 5;
  // postcondition: y > x
  ```
Backward reasoning example

// precondition: ??
x = x + 3;
// ??
y = 2x;
// ??
x = 5;
// postcondition: y > x
Backward reasoning example

// precondition: ??
x = x + 3;
// ??
y = 2x;
// y > 5
x = 5;
// postcondition: y > x
Backward reasoning example

// precondition: ??
x = x + 3;
// x >= 3
y = 2x;
// y > 5
x = 5;
// postcondition: y > x
Backward reasoning example

// precondition: x >= 0
x = x + 3;
// x >= 3
y = 2x;
// y > 5
x = 5;
// postcondition: y > x
Advantages of backward reasoning

• Usually more helpful
  – Helps you understand what should happen
  – Given a specific goal, indicates how to achieve it
  – Given an error, gives a test case that exposes it
Common formal reasoning techniques

• Weakest preconditions, loop invariants

• Mathematical proofs
  – e.g., lemmas, induction

• Constraint satisfaction problems (CSP)
Common formal reasoning techniques

• Weakest preconditions, loop invariants

• Mathematical proofs
  – e.g., lemmas, induction

• Constraint satisfaction problems (CSP)
Overview of CSP-based theorem provers

Take as input:

- a **program** modeled in first-order logic (i.e. a set of boolean formulae)
- a **question** about that program also modeled in first-order logic (i.e. additional boolean formulae)
Overview of CSP-based theorem provers

Use **formal reasoning (e.g., decision procedures)** to produce as output one of the following:

- **satisfiable**: For some input/output pairs (i.e. variable assignments), the program does satisfy the question

- **unsatisfiable**: For all input/output pairs (i.e. variable assignment), the program does not satisfy the question
Z3

• Online interfaces:
  • https://rise4fun.com/z3
  • https://compsys-tools.ens-lyon.fr/z3/index.php

• Download: https://github.com/Z3Prover/z3
Theorem prover architecture: Z3

Program [Constraints]  Question [Constraints]

Z3 theorem prover [SAT constraint solver]

SAT (+ positive example) [Variable assignments]
-OR-
UNSAT

https://github.com/Z3Prover/z3
Theorem prover architecture: Z3

Program [Constraints] → Z3 theorem prover [SAT constraint solver + Heuristics] → Question [Constraints]

SAT (+ positive example) [Variable assignments] -OR- UNSAT -OR- UNK
Programming language: SMT (Satisfiability Modulo Theories)

Supports the following:

• Variables, e.g., `(declare-const a Int)`
• Assertions, e.g., `(assert (> a 0))`
• Print statements, e.g., `(echo “Printing…”)`
• Comments, e.g., `;; This is a comment.`
• Functions, e.g.,
  `(declare-fun compareTo (Int Int) Bool)`
• ...

[http://smtlib.cs.uiowa.edu/](http://smtlib.cs.uiowa.edu/)
Example: Simple program

Java:

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

Z3 input:
Example: Simple program

Java:

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

Z3 input:

```
(declare-const a Int)
(declare-const b Int)
```
Example: Simple program

Java:

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

Z3 input:

```
(declare-const a Int)
(declare-const b Int)
(declare-const r1 Int)
```
Example: Simple program

Java:

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

Z3 input:

```z3
(declare-const a Int)
(declare-const b Int)
(declare-const r1 Int)
(assert (= (+ a b) r1))
```
Z3’s question types

- Basic boolean equations
- More complex boolean equations involving existential and universal quantification
- Certain math equations involving numbers, and linear arithmetic (addition, subtraction, multiplication, division, and ordering)
Z3’s questions and possible answers

• Can ask “Is this possible (i.e. satisfiable)?”
  \((\text{check-sat})\)

• If satisfiable, can ask “What is an example (i.e. a satisfying variable assignment)?”
  \((\text{get-model})\)

• If unsatisfiable (or unknown), cannot ask “What is an example?”
Example: Simple program

Question: Can sum ever return 0?

(assert (= r1 0)) ;; We want r1 = a + b to be 0
(check-sat) ;; Ask if this is possible
(get-model) ;; It is, so let’s get an example
SAT constraint solving

- **Satisfiability** is about finding a solution to a set of constraints (in our case formulae).

- A formula $F$ is **satisfiable** if there is some assignment of appropriate values to its uninterpreted function and constant symbols under which $F$ evaluates to true.

SAT constraint solving

• **Validity** is about finding a proof of a statement (in our case a formula $F$).

• A formula $F$ is **valid** if $F$ **always** evaluates to true for any assignment of appropriate values to its uninterpreted function and constant symbols.
SAT constraint solving

- F is **satisfiable** if and only if not F is not **valid** (is **invalid**).
  - Report that there exists a satisfying assignment

- F is **valid** precisely when not F is not **satisfiable** (is **unsatisfiable**).
  - Report that none of the assignments are satisfying
Example: Simple program

**Z3 run:**

```
z3 Z3code.simple.smt2
```

**Z3 output:**

```
sat
(model
  (define-fun a () Int 0)
  (define-fun b () Int 0)
  (define-fun r1 () Int 0)
)
```
Example: Simple program

**Z3 run:**

z3 Z3code.simple.smt2

**Z3 output:**

sat
(model
  (define-fun a () Int 0)
  (define-fun b () Int 0)
  (define-fun r1 () Int 0)
)

Here is the expected result.
The sum is 0 when both a and b are 0.
Detect mutants using Z3

1. Given an original program and a mutant, use Z3 to show that mutant is either detectable or undetectable

2. If the mutant is detectable, use Z3’s output to create a JUnit test to kill it
Show a mutant is either detectable or undetectable

- If two functions are behaviorally equivalent (i.e. undetectable mutants), for all inputs, they act the same (in our case produce the same outputs)

- We can ask if two functions are **NOT** behaviorally equivalent (i.e. detectable mutants), does there exist an input for which they act differently (in our case produce different outputs)
Example: Pair 0

Java:

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

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

Z3 input:
Example: Pair 0

**Java:**

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

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

**Z3 input:**

```
(declare-const a Int)
(declare-const b Int)
```
Example: Pair 0

Java:

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

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

Z3 input:

```z3
(declare-const a Int)
(declare-const b Int)
```
Example: Pair 0

Java:

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

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

Z3 input:

```
(declare-const a Int)
(declare-const b Int)
(declare-const r1 Int)
(declare-const mutated_r1 Int)
```
Example: Pair 0

Java:

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

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

Z3 input:

```z3
(declare-const a Int)
(declare-const b Int)
(declare-const r1 Int)
(declare-const mutated_r1 Int)
```
Example: Pair 0

Java:

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

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

Z3 input:

```z3
(declare-const a Int)
(declare-const b Int)
(declare-const r1 Int)
(declare-const mutated_r1 Int)
(assert (= (+ a b) r1))
(assert (= (* a b) mutated_r1))
```
Example: Pair 0

Java:

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

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

Z3 input:

```z3
(declare-const a Int)
(declare-const b Int)
(declare-const r1 Int)
(declare-const mutated_r1 Int)

(assert (= (+ a b) r1))
(assert (= (* a b) mutated_r1))
```
Example: Pair 0

Java:

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

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

Z3 input:

```z3
(declare-const a Int)
(declare-const b Int)
(declare-const r1 Int)
(declare-const mutated_r1 Int)

(assert (= (+ a b) r1))
(assert (= (* a b) mutated_r1))

(assert (not (= r1 mutated_r1)))
```
Example: Pair 0

Java:

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

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

Z3 input:

```z3
(declare-const a Int)
(declare-const b Int)
(declare-const r1 Int)
(declare-const mutated_r1 Int)

(assert (= (+ a b) r1))
(assert (= (* a b) mutated_r1))
(assert (not (= r1 mutated_r1)))
(check-sat)
;;(get-model)
```
Example: Pair 0 (cont.)

Z3 output:

sat

(get-model)
(model
  (define-fun mutated_r1 () Int (- 8))
  (define-fun r1 () Int 2)
  (define-fun b () Int 4)
  (define-fun a () Int (- 2))
)
Example: Pair 0 (cont.)

Z3 output:

sat

(model
  (define-fun mutated_r1 () Int (- 8))
  (define-fun r1 () Int 2)
  (define-fun b () Int 4)
  (define-fun a () Int (- 2))
)

JUnit test case:

@Test
public killSimpleMutant {

}
Example: Pair 0 (cont.)

Z3 output:

sat

(model
  (define-fun mutated_r1 () Int (- 8))
  (define-fun r1 () Int 2)
  (define-fun b () Int 4)
  (define-fun a () Int (- 2))
)

JUnit test case:

@Test
public killSimpleMutant {
  int a = -2;
  int b = 4;
  assertEquals(2,
               sum(a,b));
}

Example: Pair 0 (cont.)

Z3 output:

sat

(model
  (define-fun mutated_r1 () Int (- 8))
  (define-fun r1 () Int 2)
  (define-fun b () Int 4)
  (define-fun a () Int (- 2))
)

JUnit test case:

@Test
public killSimpleMutant {
  int a = -2;
  int b = 4;
  assertEquals(2, // Expected: 2
               sum(a,b)); // Actual: -8
}
Example: Pair 2

Z3 input:

;;; START STUDENT CODE ;;;

(assert (= a-eq-b (= a b))) ;It is fine if these three lines are missing.
(assert (= a-eq-c (= a c))) ;Adding asserts to an unsat problem cannot make it sat
(assert (= b-eq-c (= b c))) ;so just the lines below are sufficient to prove unsat
(assert (= initial-condition (= trian 0)))
(assert (= mutated-condition (= trian 0)))

;;; END STUDENT CODE ;;;

Z3 output: UNSAT
Example: Pair 2

Z3 input:

;;;;;;;;;;;;;;;;;; START STUDENT CODE;;;;;;;;;;;;;;;;
(assert (= a-eq-b (= a b))) ;It is fine if these three lines are missing.
(assert (= a-eq-c (= a c))) ;Adding asserts to an unsat problem cannot make it sat
(assert (= b-eq-c (= b c))) ;so just the lines below are sufficient to prove unsat
(assert (= initial-condition (= trian 0)))
(assert (= mutated-condition (<= trian 0)))
;;;;;;;;;;;;;;;; END STUDENT CODE;;;;;;;;;;;;;;;;

Z3 output: UNSAT // This is an undetectable mutant.
Advantages and disadvantages of theorem proving

- **Automates reasoning about all program behaviors**
- Considers all possible input/output pairs
- **Suffers from the state space explosion problem**
  - Incorporates heuristics that may lead to returning unknown
- Requires expertise with modeling in first-order logic
- May be hard to update the proof after program changes
Advantages and disadvantages of theorem proving

• **Automates reasoning about all program behaviors**
  • Considers all possible input/output pairs

• **Suffers from the state space explosion problem**
  • Incorporates heuristics that may lead to returning unknown

• **Requires expertise with modeling in first-order logic**

• **May be hard to update the proof after program changes**
Repair/Synthesis of proofs

- Repair of proofs -
  [https://github.com/uwplse/coq-change-analytics](https://github.com/uwplse/coq-change-analytics) (Talia Ringer, Alex Sanchez-Stern, et al.)

- Synthesis of proofs -
  [https://github.com/LASER-UMASS/TacTok](https://github.com/LASER-UMASS/TacTok) (Emily First, Yuriy Brun, Arjun Guha)