CS 520
Theory and Practice of Software Engineering
Spring 2021

Automated theorem proving

April 22, 2020
Upcoming assignments

• Week 12 Participation Questionnaire will be about Automated Theorem Proving

• Final project deliverables are due Tuesday May 11, 11:59 PM (just before midnight)
Programs are known to be error-prone

• Capture complex aspects such as:
  • Threads and synchronization (e.g., Java locks)
  • Dynamically heap allocated structured data types (e.g., Java classes)
  • Dynamically stack allocated procedures (e.g., Java methods)
  • Non-determinism (e.g., Java HashSet)
  • Many input/output pairs

• Challenging to reason about all possible behaviors of these programs
Programs are known to be error-prone

• Capture complex aspects such as:
  • Threads and synchronization (e.g., Java locks)
  • Dynamically heap allocated structured data types (e.g., Java classes)
  • Dynamically stack allocated procedures (e.g., Java methods)
  • Non-determinism (e.g., Java HashSet)
  • Many input/output pairs

• Challenging to reason about all possible behaviors of these programs
Overview of theorem provers

Key idea: Constraint satisfaction problem

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)
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
Possible uses of theorem provers

• Testing, e.g., detecting mutants
• Analysis
• Verification
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] → Z3 theorem prover [SAT constraint solver] → Question [Constraints]

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/
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:

```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)?” *(check-sat)*

• If satisfiable, can ask “What is an example (i.e. a satisfying variable assignment)?” *(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)
)

Here is the expected result.
The sum is 0 when both a and b are 0.
Advantages and disadvantages of theorem proving

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

- Requires expertise with modeling in first-order logic

- Suffers from the state space explosion problem
  - Incorporates heuristics that may lead to returning unknown
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)
Z3

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

• Download: https://github.com/Z3Prover/z3

• Examples:
  https://people.cs.umass.edu/~hconboy/class/2021Spring/CS520/lectures/20210422theoremProving-programs.zip
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:

```
(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:

```plaintext
(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:

```
(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

(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, sum(a,b)); // Expected: 2
    // Actual: -8
}
Example: Pair 1
Example: Pair 1

Z3 input:

```
```

```
( assert (= (+ x y) a1))
( assert (= (+ a1 z) a2))
( assert (= (- a1 z) mutated_a2))
( assert (not (= a2 mutated_a2)))
```

```
```

Z3 output: SAT
Example: Pair 2
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 3