Invariants and partial test oracles

UW CSE P 504
Reasoning about programs
Reasoning about programs

Use cases

- Testing: increase confidence in correctness
- Verification: prove facts to be true, e.g.:
  - x is never null
  - y is always greater than 0
  - a happens before b
- Debugging: understand why code is incorrect
Reasoning about programs

Use cases
- Testing: increase confidence in correctness
- Verification: prove facts to be true, e.g.:
  - x is never null
  - y is always greater than 0
  - a happens before b
- Debugging: understand why code is incorrect

Approaches
- Testing
- Abstract interpretation
- Theorem proving
- Delta debugging
- Slicing
- ...
Forward and 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?

\[
\{ x = 2; y = 5 \}
\]

\[
y = x++; \\
??
\]
Forward and 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?

```plaintext
\{ x = 2; y = 5 \}
y = x++;
\{ x = 3; y = 2 \}
```
Forward and 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?

\[
\begin{align*}
\langle x = 2; y = 5 \rangle \\
y &= x++; \\
\langle x = 3; y = 2 \rangle \\
\end{align*}
\]

??
\[
\begin{align*}
y &= x++; \\
\langle x = 3; y = 2 \rangle \\
\end{align*}
\]
Forward and 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?

```plaintext
\{ x = 2; y = 5 \}
y = x++;  
\{ x = 3; y = 2 \}

\{ x = 2; y = anything \}
y = x++;  
\{ x = 3; y = 2 \}
```
Forward and 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 are the pros and cons of each approach?
Forward and backward reasoning

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

Backward reasoning
- Usually more helpful
- Helps understand what should happen
- Given a specific goal, indicates how to achieve it
- Given an error, gives a test case that exposes it
Pre- and post-conditions and invariants
Terminology

**Pre-condition** (of a procedure)
- A condition that should be true when entering
- Includes expectations about the arguments

**Post-condition** (of a procedure)
- A condition that should be true when exiting

**Specification** or **contract**: the pre-condition and post-condition
- A procedure is *correct* if it satisfies its specification

**Loop invariant**
- A condition that should be true at beginning of each loop iteration
- \[ \Rightarrow \] is also true when the loop exits
Pre-conditions and post-conditions

```java
double avgAbs(double[] nums) {
    int n = nums.length;
    double sum = 0;
    int i = 0;
    while (i != n) {
        if (nums[i] > 0) {
            sum = sum + nums[i];
        } else {
            sum = sum - nums[i];
        }
        i = i + 1;
    }
    return sum / n;
}
```

What are the pre-conditions of this procedure?

What are the post-conditions?
Pre-conditions and post-conditions

```
1 double avgAbs(double[] nums) {
2    int n = nums.length;
3    double sum = 0;
4
5    int i = 0;
6    while (i != n) {
7        if (nums[i] > 0) {
8            sum = sum + nums[i];
9        } else {
10           sum = sum - nums[i];
11        }
12        i = i + 1;
13    }
14
15    return sum / n;
16 }
```

**Pre-conditions**
- `nums` is not null
- `nums.length > 0`

**Post-conditions**
- `n > 0`
- `n = nums.length`
- `i = n`
- `sum >= 0`
- return value >= 0
- return value = avg of absolute values of `nums`
- `nums = nums_{pre (frame condition)}`
- ...
double avgAbs(double[] nums) {
    int n = nums.length;
    double sum = 0;

    int i = 0;
    while (i < n) {
        if (nums[i] > 0) {
            sum = sum + nums[i];
        } else {
            sum = sum - nums[i];
        }
        i = i + 1;
    }

    return sum / n;
}
double `avgAbs(double[] nums)` {
    int n = nums.length;
    double sum = 0;
    int i = 0;
    while (i < n) {
        if (nums[i] > 0) {
            sum = sum + nums[i];
        } else {
            sum = sum - nums[i];
        }
        i = i + 1;
    }
    return sum / n;
}

What is the loop invariant?

0 <= i <= n

\[
\text{sum} = \sum_{j=0}^{j<i} |nums[j]|
\]
Loop invariants

double avgAbs(double[] nums) {
    int n = nums.length;
    double sum = 0;

    int i = 0;
    while (i < n) {
        if (nums[i] > 0) {
            sum = sum + nums[i];
        } else {
            sum = sum - nums[i];
        }
        i = i + 1;
    }

    return sum / n;
}

What is the loop invariant?
0 <= i <= n

sum = ∑_{j=0}^{i-1} |nums[j]|

Does this loop terminate?
Loop invariants

```java
1 double avgAbs(double[] nums) {
2     int n = nums.length;
3     double sum = 0;
4     int i = 0;
5     while (i < n) {
6         if (nums[i] > 0) {
7             sum = sum + nums[i];
8         } else {
9             sum = sum - nums[i];
10        }
11         i = i + 1;
12     }
13     return sum / n;
14 }
```

What is the loop invariant?

$0 <= i <= n$

$sum = \sum_{j=0}^{j<i} |nums[j]|$

Does this loop terminate?

- The loop terminates when $n - i$ is zero or less
- $n - i$ decreases each iteration
  \[ \therefore \text{the loop terminates} \]

Axiom: If an integer decreases on each operation, it eventually reaches zero or less.
Determining invariants is an open problem.

It can be easier to prove a theorem than to posit the theorem.

Can we automatically propose theorems?

```java
1 double avgAbs(double[] nums) {
2     int n = nums.length;
3     double sum = 0;
4     
5     int i = 0;
6     while (i < n) {
7         if (nums[i] > 0) {
8             sum = sum + nums[i];
9         } else {
10             sum = sum - nums[i];
11         }
12         i = i + 1;
13     }
14     return sum / n;
15 }
```
Daikon live example

Daikon: general workflow
Log-based model inference

(a) Input log

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>send(x)</td>
</tr>
<tr>
<td>2.0</td>
<td>M</td>
</tr>
<tr>
<td>3.3</td>
<td>A?ack,o</td>
</tr>
<tr>
<td>3.3</td>
<td>M</td>
</tr>
<tr>
<td>4.3</td>
<td>send(x)</td>
</tr>
<tr>
<td>5.3</td>
<td>M</td>
</tr>
<tr>
<td>6.6</td>
<td>A?ack,e</td>
</tr>
<tr>
<td>7.6</td>
<td>send(x)</td>
</tr>
<tr>
<td>8.8</td>
<td>M</td>
</tr>
<tr>
<td>9.9</td>
<td>A?ack,o</td>
</tr>
<tr>
<td>10.9</td>
<td>send(x)</td>
</tr>
<tr>
<td>11.9</td>
<td>M</td>
</tr>
<tr>
<td>12.12</td>
<td>A?ack,e</td>
</tr>
</tbody>
</table>

(b) Output model

(a) Input log

(b.1) Output model (Sender)

(b.2) Output model (Receiver)
Discussion about Daikon
Discussion about Daikon

- What was so hard about this?
- Choice of templates
- Scalability: what are the most important factors?
- Biggest problem: too much output, not too little
- Dereferencing pointers
Partial test oracles
Property-based testing
Metamorphic testing*

*Chen et al. coined the term metamorphic testing in 1998, but the key idea was first described by Ammann and Knight as data diversity in 1988.
Partial test oracles

Partial test oracle

- Necessary (but not sufficient) conditions
- Example: \( \text{abs}(x) \geq 0 \)
Property-based testing

Partial test oracle
- Necessary (but not sufficient) conditions
- Example: abs(x) >= 0

Property-based testing
- Check property that holds for every input, which requires knowledge about the system; contrast to “assert(x == 22)”
- Commonly used with random input generation

How is property-based testing different from testing with input-output pairs and how is it different from fuzzing?
Property-based testing

Partial test oracle
- Necessary (but not sufficient) conditions
- Example: $\text{abs}(x) \geq 0$

Property-based testing
- Check property that holds for every input, which requires knowledge about the system; contrast to “$\text{assert}(x == 22)$”
- Commonly used with random input generation
- Contrast: testing with input-output pairs usually checks for sufficient conditions for a (small) subset of all possible inputs
- Contrast: fuzzing is usually a black-box approach that checks for a simple property (“should not crash”)
Data diversity and metamorphic testing

Simple case: related inputs with identical outcomes
- Expected output for a given input is unknown
- Two related inputs must result in the same output
- Example: abs(x) == abs(-x)
Data diversity and metamorphic testing

Simple case: related inputs with identical outcomes
- Expected output for a given input is unknown
- Two related inputs must result in the same output
- Example: \( \text{abs}(x) == \text{abs}(-x) \)

Generalization: related inputs and related outputs
- Input \( i_1 \) yields (unknown) \( o_1 \) (initial input)
- \( R_i : i_1 \leftrightarrow i_2 \) (follow-up input)
- \( R_o : o_1 \leftrightarrow o_2 \) (necessary condition)
Metamorphic testing: impact
Metamorphic testing: impact
Discrete wavelet transformation
Discrete wavelet transformation
Discrete wavelet transformation

Original image → DWT → LL1, HL1

Original image → DWT → LH1, HH1
A concrete SUT: jpeg2000 encoder
Metamorphic testing: three requirements

MT requires
1. A set of initial inputs (or a generator)
2. A relation $R_i$: generates follow-up inputs
3. A relation $R_o$: necessary correctness condition
Metamorphic testing: Input generation
Metamorphic testing: relations $R_i$ and $R_o$
Metamorphic testing: relations $R_i$ and $R_o$
Metamorphic testing: Relations

1. $R_i$: Add a constant offset to all color values
   $R_o$: ???

\[ \begin{array}{ccc}
& R_I & \\
\text{??？} & & \text{??？}
\end{array} \]
Metamorphic testing: Relations

1. $R_i$: Add a constant offset to all color values

   $R_o$: Only the DC component must change
Metamorphic testing: Relations

1. \( R_i \): Add a constant offset to all color values
   \( R_o \): Only the DC component must change

2. \( R_i \): Invert the color values
   \( R_o \): The color values of the output must be inverted
Metamorphic testing: Relations

1. \( R_i \): Add a constant offset to all color values
   \( R_o \): Only the DC component must change

2. \( R_i \): Invert the color values
   \( R_o \): The color values of the output must be inverted

3. \( R_i \): Transpose the input image
   \( R_o \): The output components must be transposed
Metamorphic testing: Relations

1. \( R_i \): Add a constant offset to all color values
   \( R_o \): Only the DC component must change

2. \( R_i \): Invert the color values
   \( R_o \): The color values of the output must be inverted

3. \( R_i \): Transpose the input image
   \( R_o \): The output components must be transposed

4. \( R_i \): Enlarge the input image ("zero-padding")
   \( R_o \): The output components must be shifted
Metamorphic testing: Relations

1. \( R_i \): Add a constant offset to all color values
   \( R_o \): Only the DC component must change

2. \( R_i \): Invert the color values
   \( R_o \): The color values of the output must be inverted

3. \( R_i \): Transpose the input image
   \( R_o \): The output components must be transposed

4. \( R_i \): Enlarge the input image ("zero-padding")
   \( R_o \): The output components must be shifted

Can compose MR
Metamorphic testing: effectiveness

Quadratic Mutation Score (Wavelet Transformation)
Putting it all together

1. (Random) input generation
2. Metamorphic testing: follow-up inputs and partial oracles
3. Delta debugging: Minimize bug-exposing inputs
4. Mutation analysis: assess the effectiveness of relations

Examples:
- GraphicsFuzz
- Testing ML-based systems