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

$\langle x = 2; y = 5 \rangle$

```
y = x++;
```

??

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?

Forward and backward reasoning

$\langle\ x = 2;\ y = 5\ \rangle$

```
y = x++;
```

$\langle\ x = 3;\ y = 2\ \rangle$

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?

Forward and backward reasoning

$\langle\, x = 2; y = 5 \,\rangle$

```
y = x++;
```

$\langle\, x = 3; y = 2 \,\rangle$

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?

??

```
y = x++;
```

$\langle\, x = 3; y = 2 \,\rangle$

Forward and backward reasoning

$\langle\ x = 2;\ y = 5\ \rangle$

```
y = x++;
```

$\langle\ x = 3;\ y = 2\ \rangle$

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?

$\langle\ x = 2;\ y = \text{anything}\ \rangle$

```
y = x++;
```

$\langle\ x = 3;\ y = 2\ \rangle$

Forward and backward reasoning

$\langle\ x = 2;\ y = 5\ \rangle$

```
y = x++;
```

$\langle\ x = 3;\ y = 2\ \rangle$

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?

$\langle\ x = 2;\ y = \text{anything}\ \rangle$

```
y = x++;
```

$\langle\ x = 3;\ y = 2\ \rangle$

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
- ⇒ is also true when the loop exits

Pre-conditions and post-conditions

```
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;
}
```

Entry point

Exit point

What are the pre-conditions of this procedure?

What are the post-conditions?

Pre-conditions and post-conditions

```
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;
}
```

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` = $\texttt{nums}_{\texttt{pre}}$ (frame condition)
- …

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;
}
```

Loop head

What is the loop invariant?

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;
}
```

Loop head

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;
}
```

Loop head

What is the loop invariant?

$0 <= i <= n$

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

Does this loop terminate?

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;
}
```

Loop head

What is the loop invariant?

$0 <= i <= n$

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

Does this loop terminate?

Axiom: If an integer decreases on each operation, it eventually reaches zero or less.

- The loop terminates when $n - i$ is zero or less
- $n - i$ decreases each iteration

∴ the loop terminates

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;
}
```

Determining invariants is an open problem.

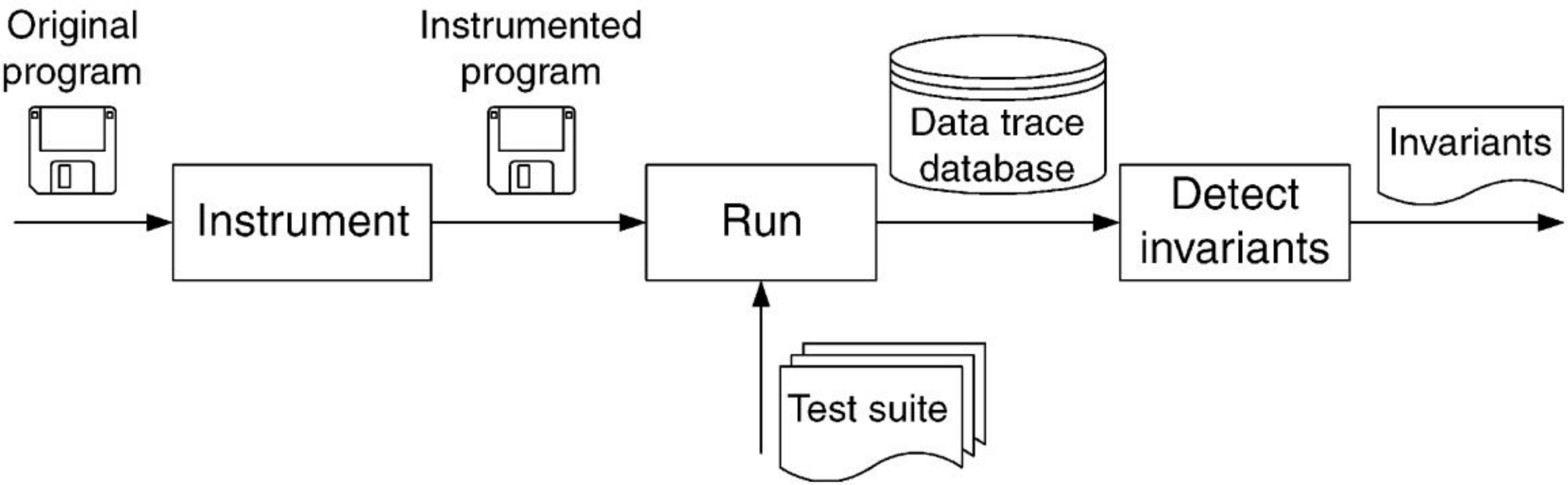
It can be easier to prove a theorem than to posit the theorem.

Can we automatically propose theorems?

Daikon live example

[(https://plse.cs.washington.edu/daikon/download/doc/daikon/Example-usage.html#Detecting-invariants-in-Java-programs)](https://plse.cs.washington.edu/daikon/download/doc/daikon/Example-usage.html#Detecting-invariants-in-Java-programs)

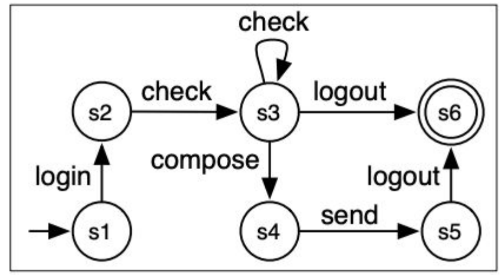
Daikon: general workflow



Log-based model inference

trace 1:	trace 2:
login	login
check	check
check	compose
logout	send
	logout

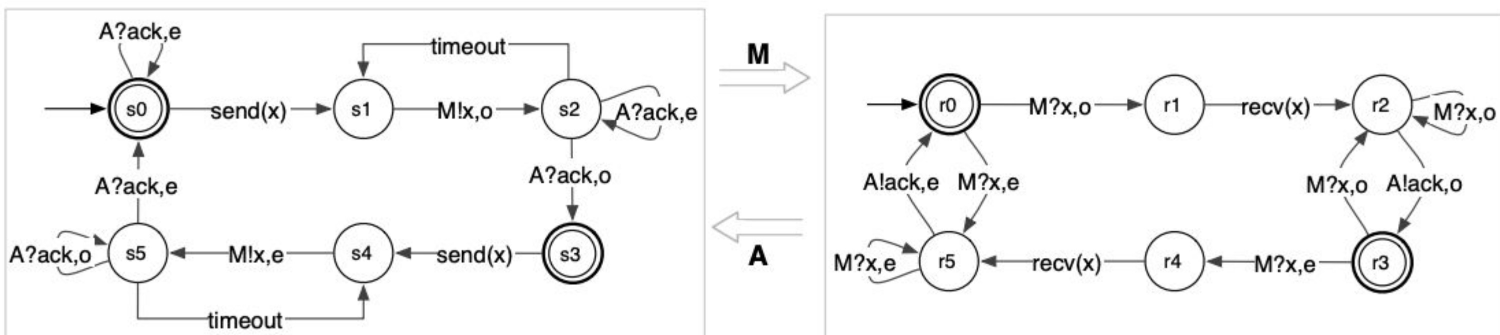
(a) Input log



(b) Output model

Sender	Receiver
1,0 send(x)	2,1 M?x,o
2,0 M!x,o	2,2 recv(x)
3,3 A?ack,o	2,3 A!ack,o
4,3 send(x)	5,4 M?x,e
5,3 M!x,e	5,5 recv(x)
6,6 A?ack,e	5,6 A!ack,e
7,6 send(x)	8,7 M?x,o
8,6 M!x,o	8,8 recv(x)
9,9 A?ack,o	8,9 A!ack,o
10,9 send(x)	11,10 M?x,e
11,9 M!x,e	11,11 recv(x)
12,12 A?ack,e	11,12 A!ack,e

(a) Input log



(b.1) Output model (Sender) **(b.2) Output model (Receiver)**

Beschastnikh et al., Synoptic: Studying Logged Behavior with Inferred Models; Inferring Models of Concurrent Systems from Logs of their Behavior with CSight

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: abs(x) >= 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: 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

- 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

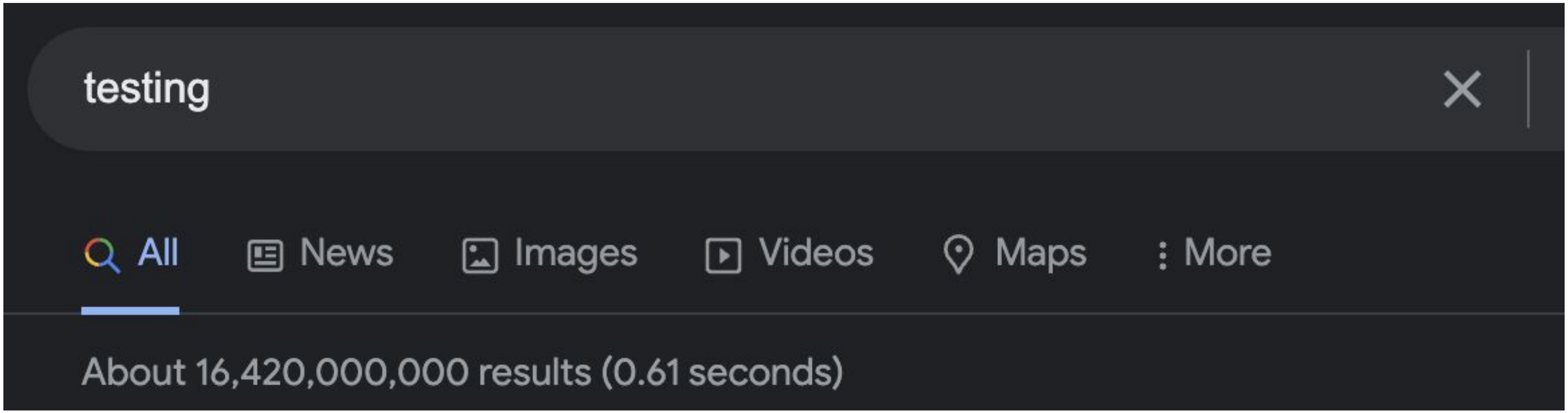
Simple case: related inputs with identical outcomes

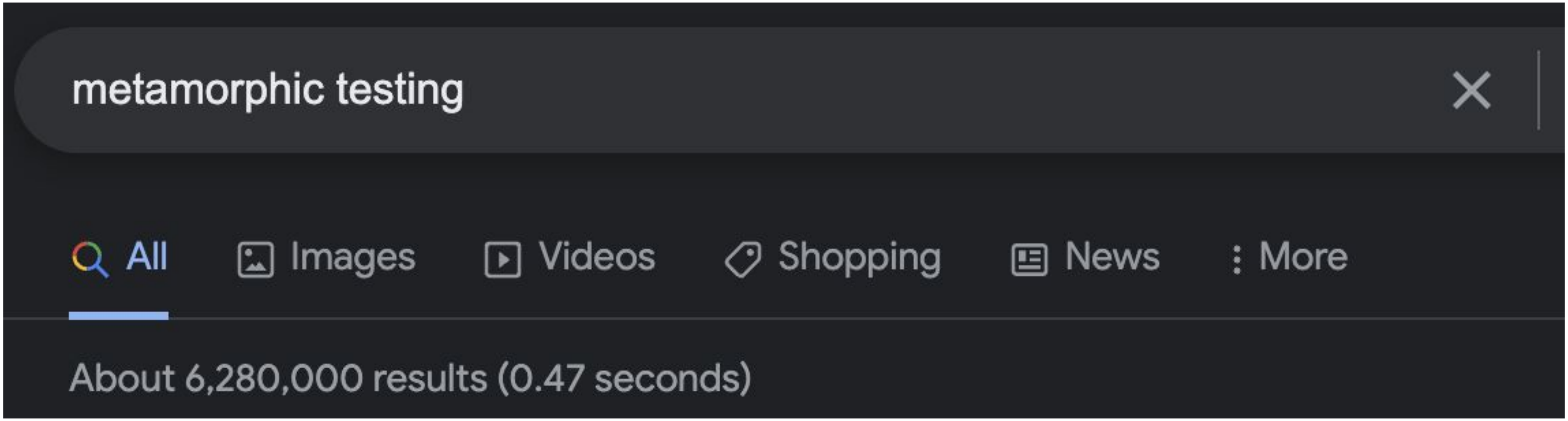
- Expected output for a given input is unknown
- Two related inputs must result in the same output
- Example: abs(x) == abs(-x)

Generalization: related inputs and related outputs

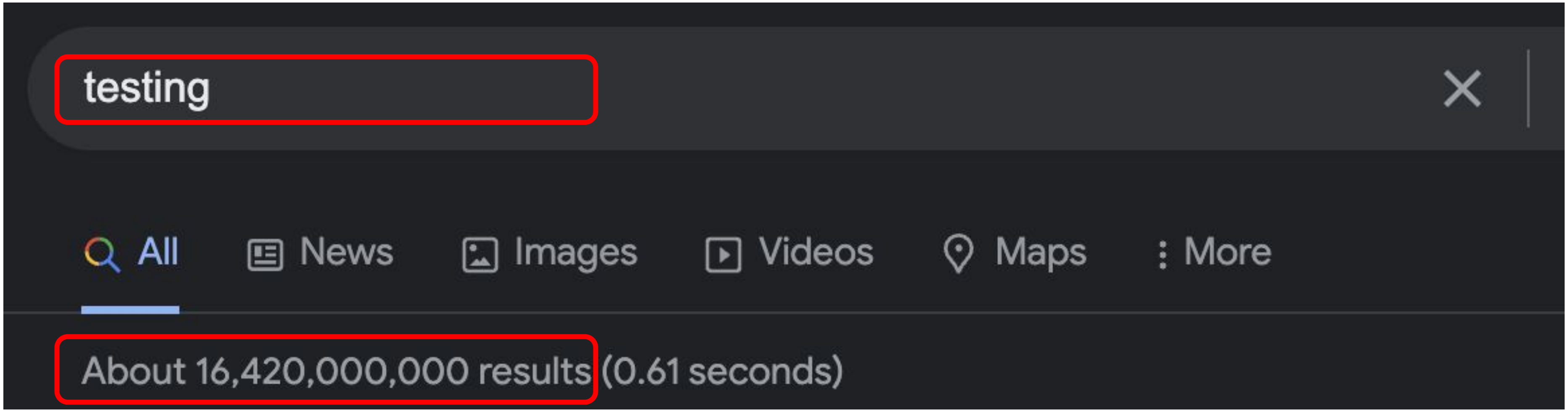
- Input i_1 yields (unknown) o_1 (initial input)
- R_i: $i_1 \Longrightarrow i_2$ (follow-up input)
- R_o: $o_1 \Longrightarrow o_2$ (necessary condition)

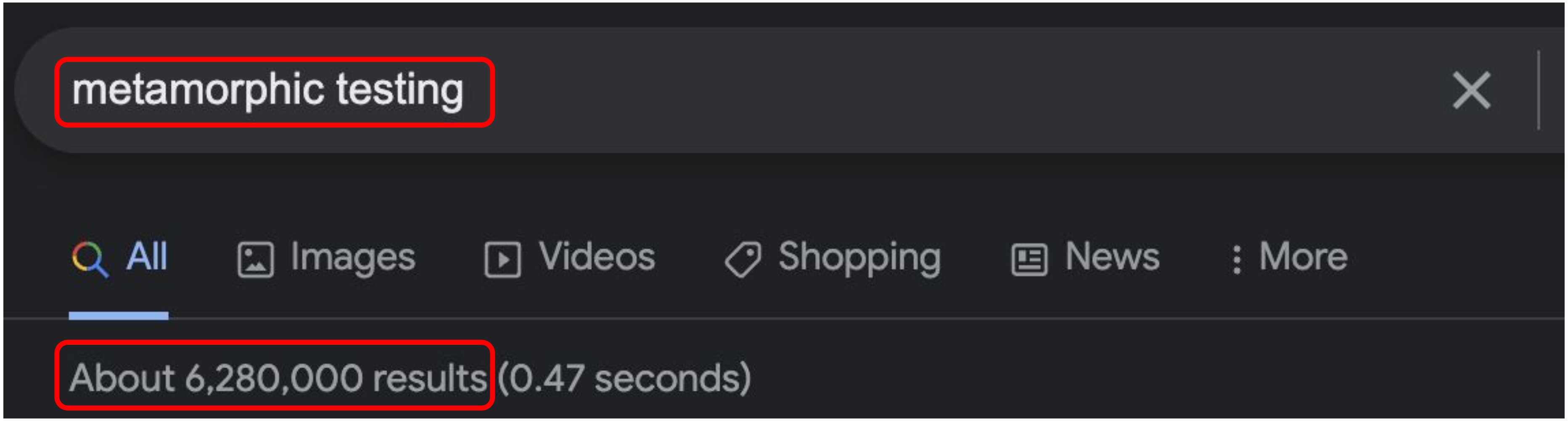
Metamorphic testing: impact





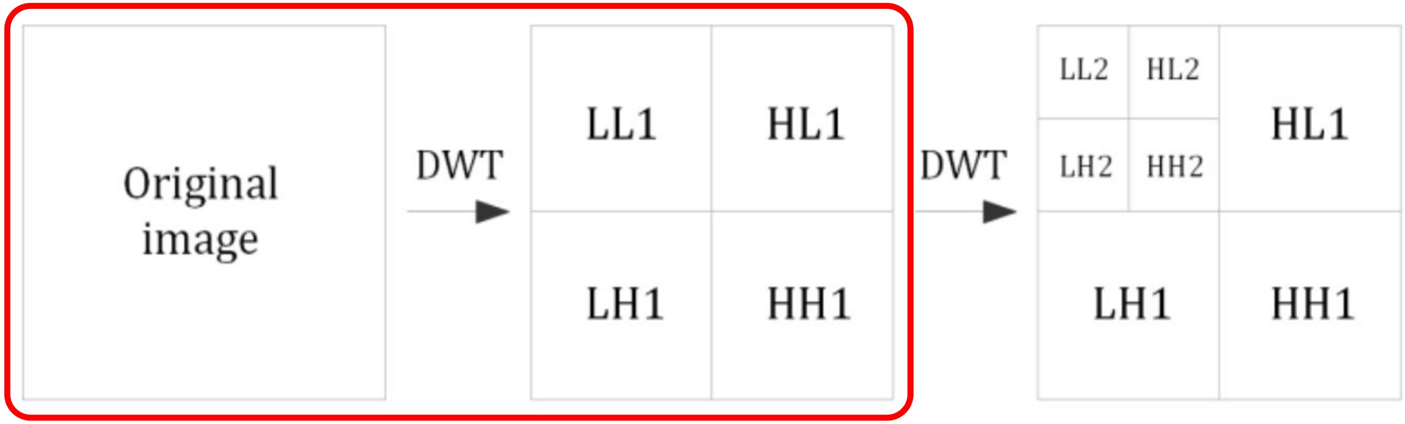
Metamorphic testing: impact



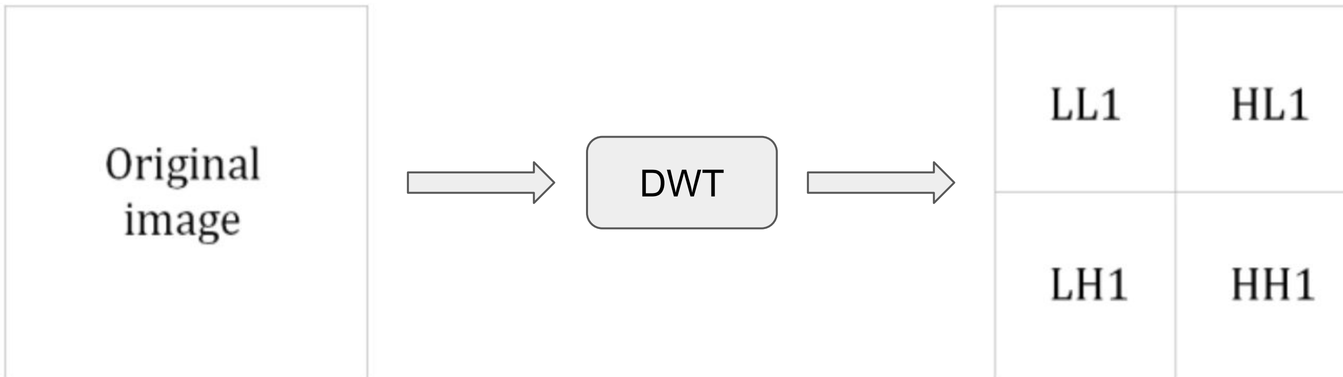


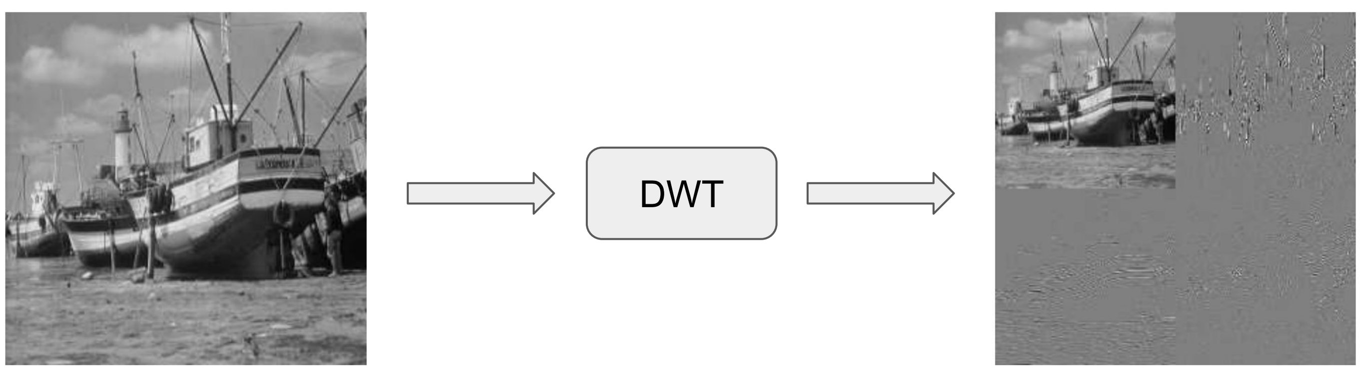
Discrete wavelet transformation

Discrete wavelet transformation

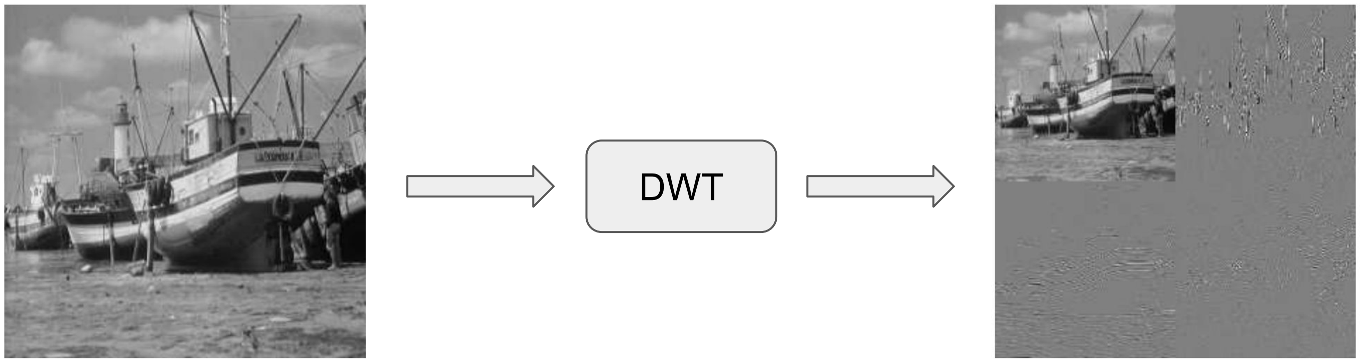


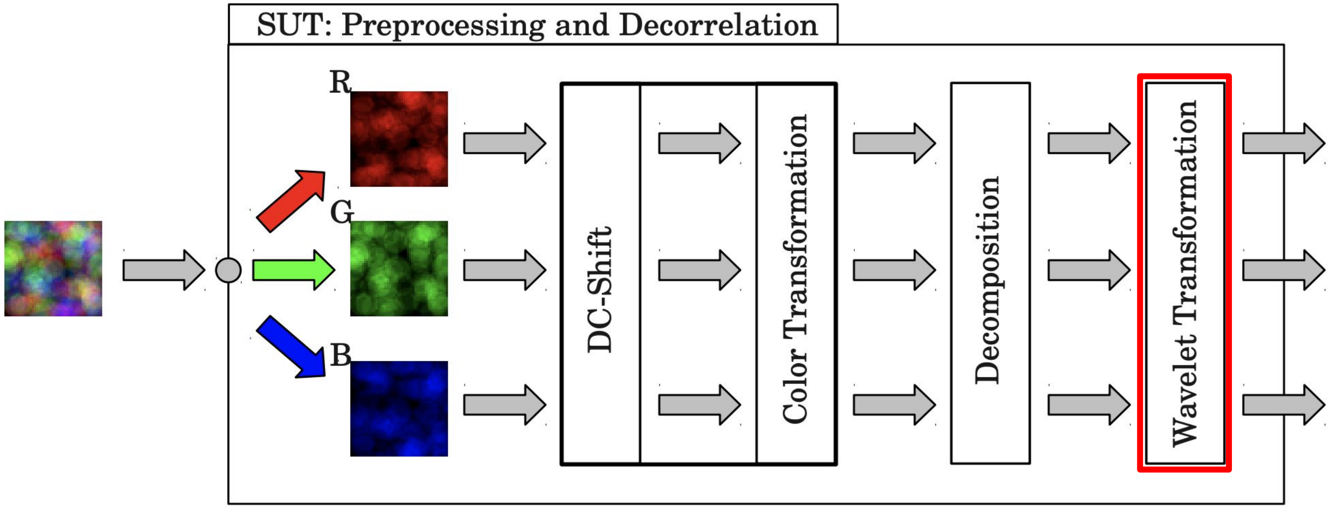
Discrete wavelet transformation



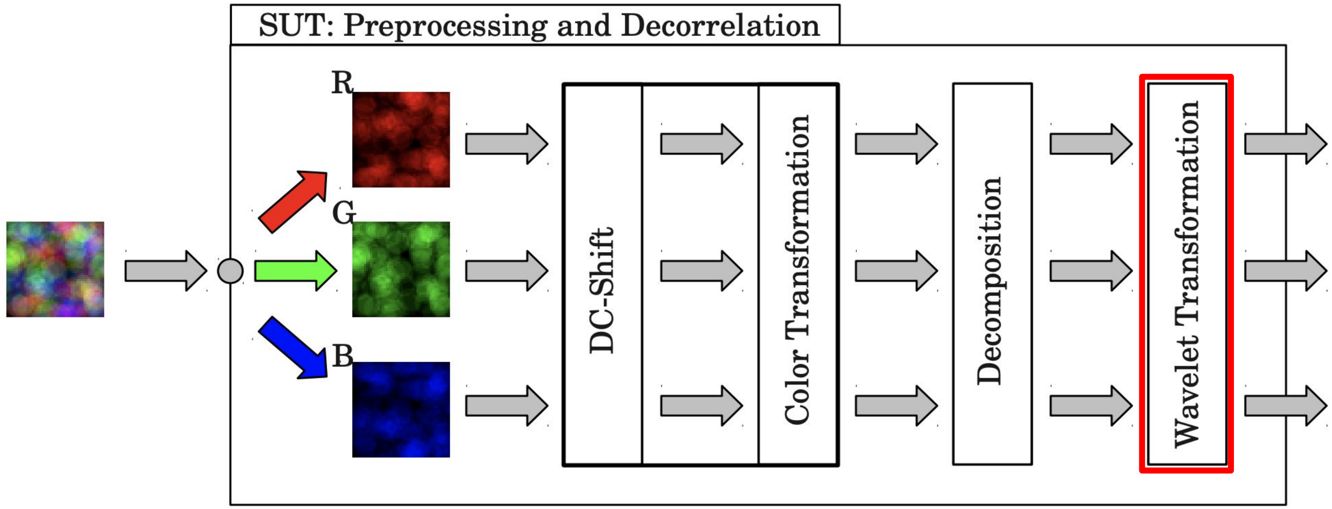


A concrete SUT: jpeg2000 encoder





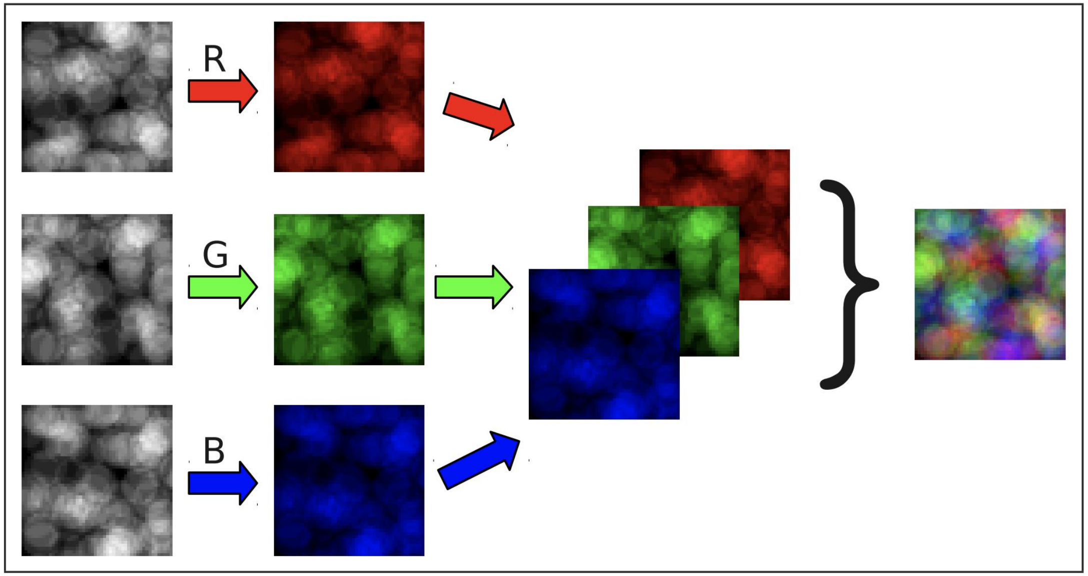
Metamorphic testing: three requirements



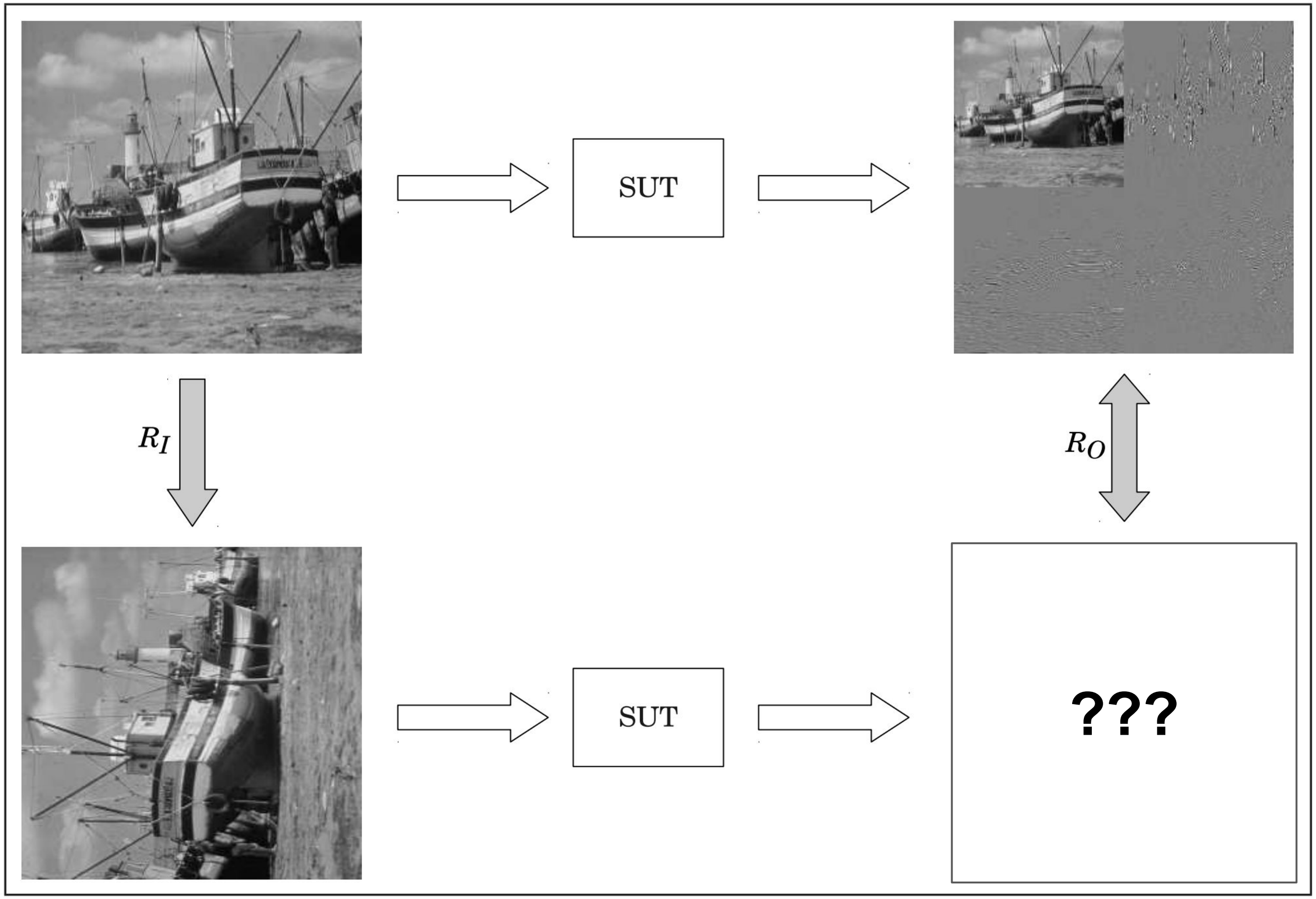
MT requires

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

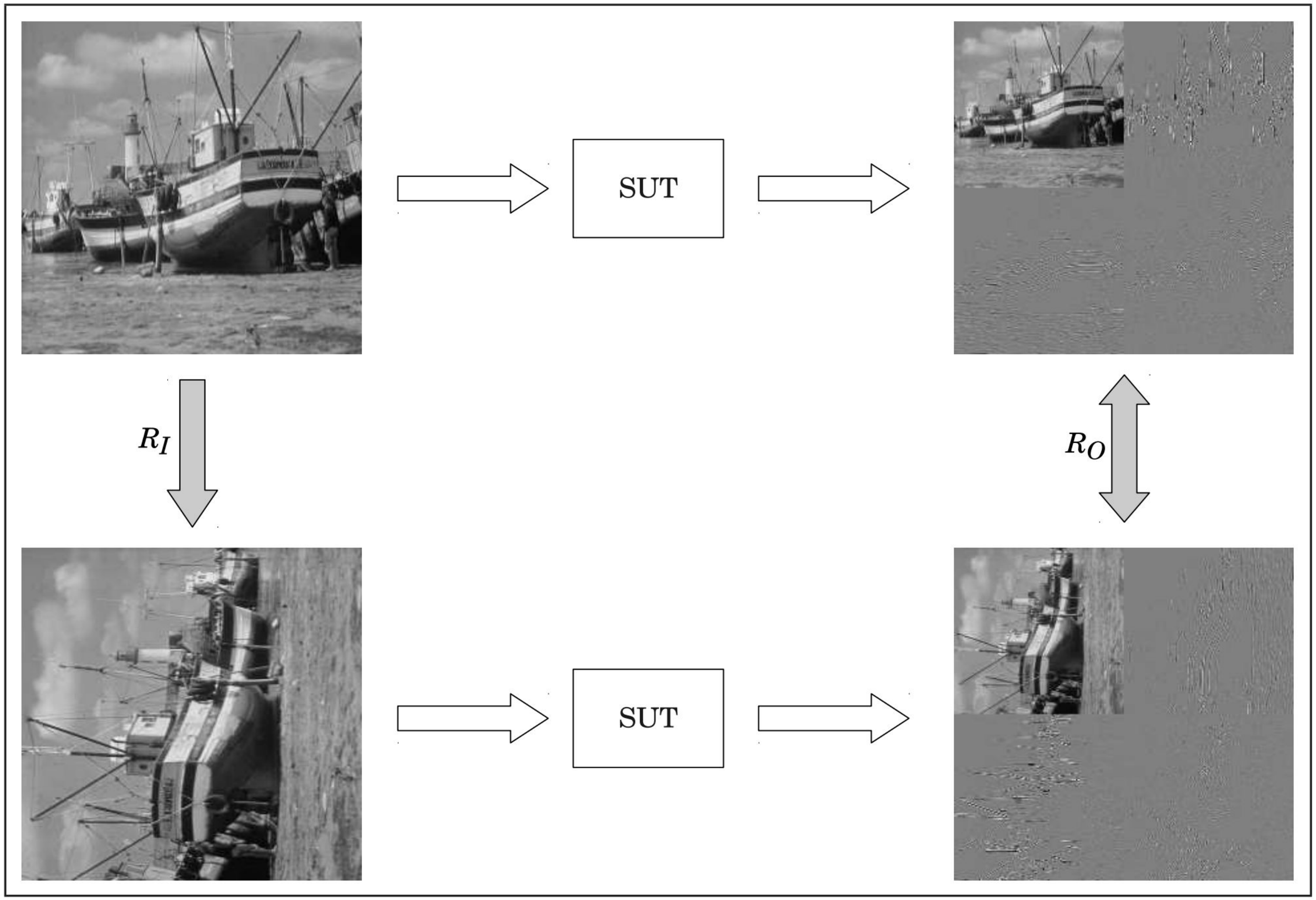
Metamorphic testing: Input generation



Metamorphic testing: relations R_i and R_o



Metamorphic testing: relations R_i and R_o

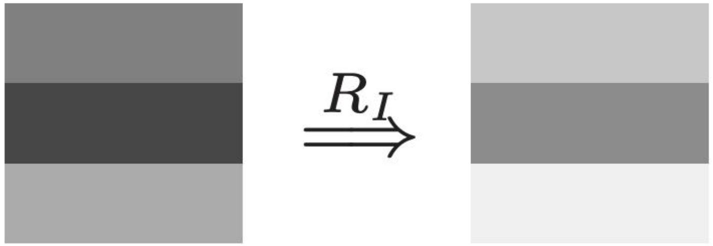


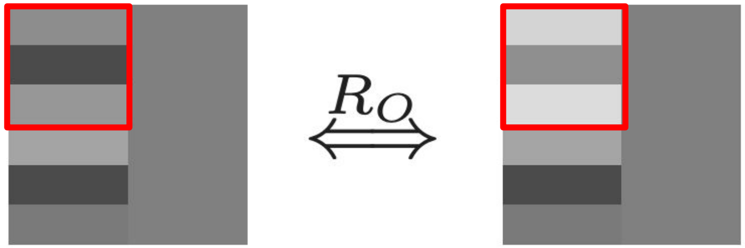
Metamorphic testing: Relations

1. R_i: Add a constant offset to all color values
   $R_{o:}$???

Metamorphic testing: Relations

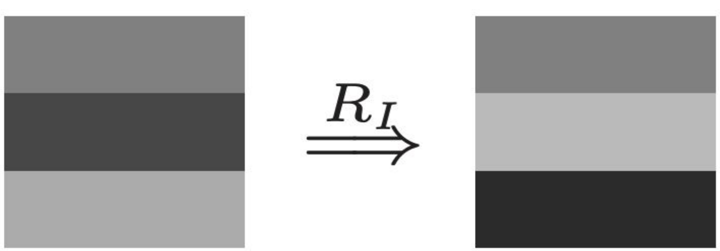
1. R_i: Add a constant offset to all color values
   $R_{o:}$ Only the DC component must change

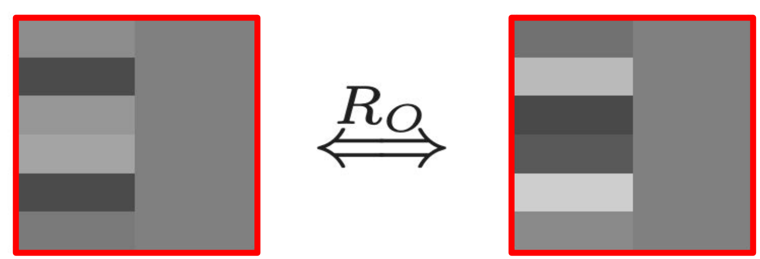




Metamorphic testing: Relations

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



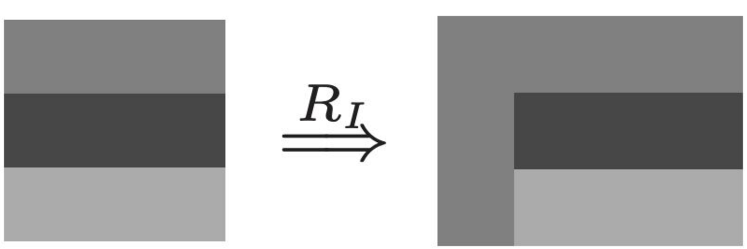


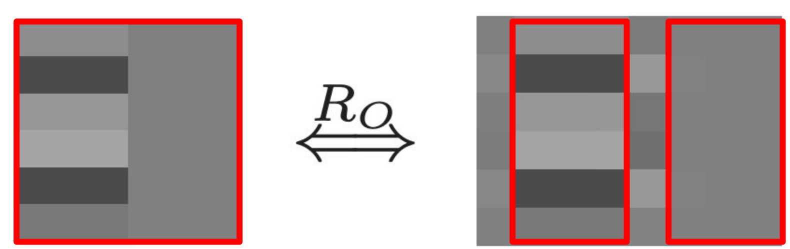
Metamorphic testing: Relations

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

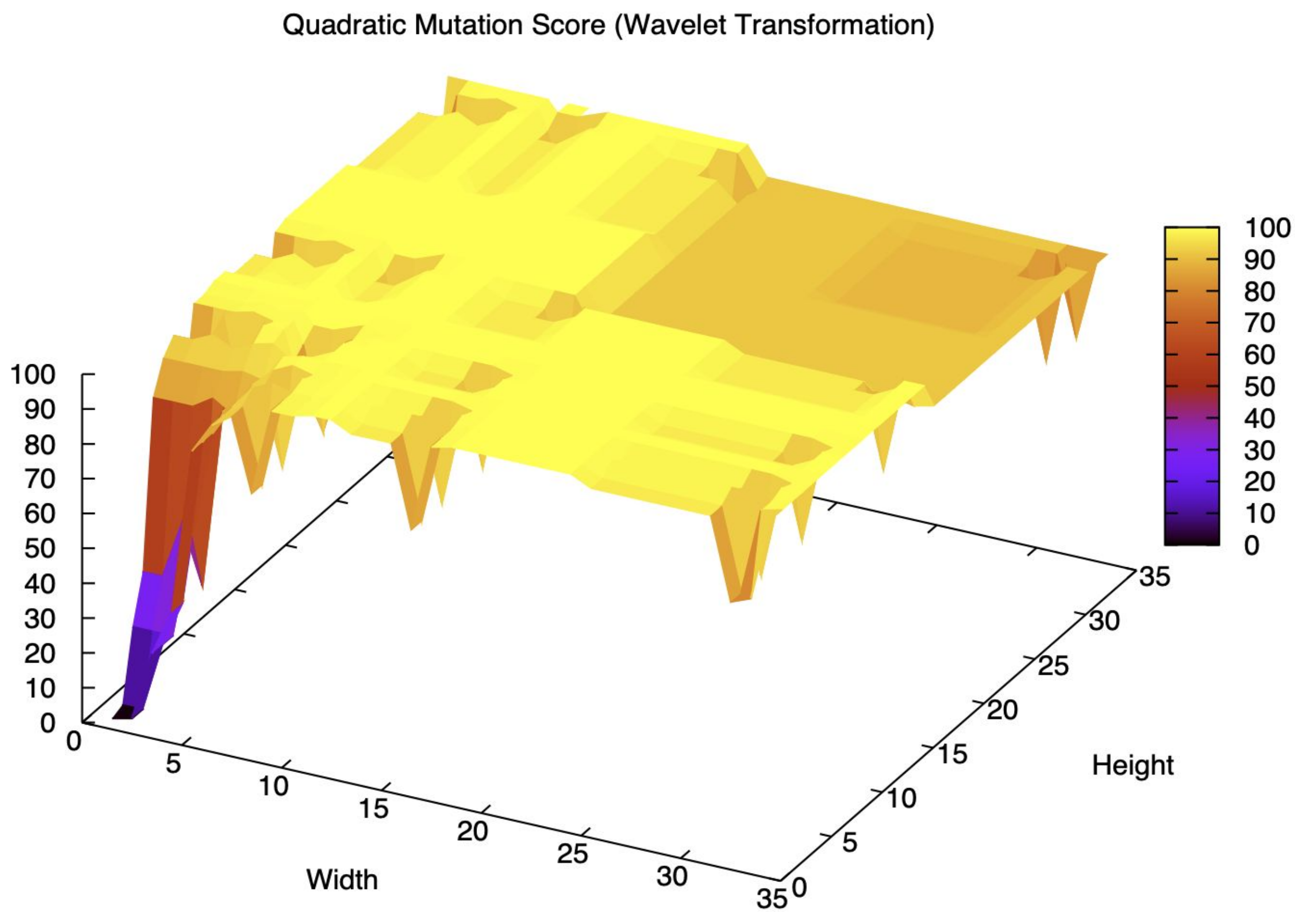
1. R_i: Add a constant offset to all color values
   $R_{o:}$ Only the DC component must change
2. R_i: Invert the color values
   $R_{o:}$ The color values of the output must be inverted
3. R_i: Transpose the input image
   $R_{o:}$ The output components must be transposed
4. R_i: Enlarge the input image (“zero-padding”)
   $R_{o:}$ The output components must be shifted

Commutative

Time-invariant

Can compose MR

Metamorphic testing: effectiveness



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