Oracles

Martin Kellogg

Reading Quiz: oracles

Q1: **TRUE** or **FALSE**: CSmith uses both static analysis and run-time checks to avoid undefined or unspecified behaviors in the C programs that it generates

Q2: The main example in section 2 of the Daikon paper was:

A. a hash table

B. a trie

C. a stack

D. a C program

Reading Quiz: oracles

Q1: **TRUE** or **FALSE**: CSmith uses both static analysis and run-time checks to avoid undefined or unspecified behaviors in the C programs that it generates

Q2: The main example in section 2 of the Daikon paper was:

A. a hash table
B. a trie
C. a stack
D. a C program

Reading Quiz: oracles

Q1: **TRUE** or **FALSE**: CSmith uses both static analysis and run-time checks to avoid undefined or unspecified behaviors in the C programs that it generates

Q2: The main example in section 2 of the Daikon paper was:

A. a hash table

B. a trie

C. a stack

D. a C program

Oracle generation

- Generating input is of limited value if **we don’t know what the program is supposed to do** with that input

Oracle generation

- Generating input is of limited value if **we don’t know what the program is supposed to do** with that input
- **Key question**: if we generate an input for a given path, **how do we tell** if the program behaved correctly?

Oracle generation: difficulty

- Oracles are **tricky**.

Oracle generation: difficulty

- Oracles are **tricky**.
  - Many believe that formally writing down what a program should do is **as hard** as coding it.

Oracle generation: difficulty

- Oracles are **tricky**.
  - Many believe that formally writing down what a program should do is **as hard** as coding it.
- The ***Oracle Problem*** is the difficulty and cost of determining the correct test oracle (i.e., output) for a given input.

Oracle generation: difficulty

- Oracles are **tricky**.
  - Many believe that formally writing down what a program should do is **as hard** as coding it.
- The ***Oracle Problem*** is the difficulty and cost of determining the correct test oracle (i.e., output) for a given input.
  - “What should the program do?”

Oracle generation: difficulty

- Oracles are **tricky**.
  - Many believe that formally writing down what a program should do is **as hard** as coding it.
- The ***Oracle Problem*** is the difficulty and cost of determining the correct test oracle (i.e., output) for a given input.
  - “What should the program do?”
  - It is **expensive** both for humans and for machines.

Oracle generation: difficulty

- Oracles are **tricky**.
  - Many believe that formally writing down what a program should do is **as hard** as coding it.
- The ***Oracle Problem*** is the difficulty and cost of determining the correct test oracle (i.e., output) for a given input.
  - "What should the program do?"
  - It is **expensive** both for humans and for machines.
    - and, for machines, sometimes impossible!

Oracle generation: implicit oracles

Observation: there are some things programs definitely shouldn’t do given **any** input

Oracle generation: implicit oracles

Observation: there are some things programs definitely shouldn't do given **any** input

- crash, segfault, loop forever, exfiltrate user data, etc.

Oracle generation: implicit oracles

Observation: there are some things programs definitely shouldn't do given **any** input

- crash, segfault, loop forever, exfiltrate user data, etc.
- **key idea**: run the program and check if it does any of these **definitely bad** things

Oracle generation: implicit oracles

Observation: there are some things programs definitely shouldn’t do given **any** input

- crash, segfault, loop forever, exfiltrate user data, etc.
- **key idea**: run the program and check if it does any of these **definitely bad** things

Definition: an ***implicit oracle*** is one associated with the language or architecture, rather than program-specific semantics (e.g., “don't segfault”, “don't loop forever”).

Oracle generation: implicit oracles

Observation: there are some things program given **any** input

- crash, segfault, loop forever, exfiltrate us
- **key idea**: run the program and check if it **definitely bad** things

Implicit oracles like these are used by **many test generation tools** (e.g., most fuzzers) in the real world.

Definition: an ***implicit oracle*** is one associated with the language or architecture, rather than program-specific semantics (e.g., “don't segfault”, “don't loop forever”).

Implicit oracles: a key weakness

Implicit oracles: a key weakness

- limited to facts that are true about **all programs**

Implicit oracles: a key weakness

- limited to facts that are true about **all programs**
    - most bugs in most programs don’t manifest as crashes
        - an implicit oracle cannot detect such bugs!

Implicit oracles: a key weakness

- limited to facts that are true about **all programs**
  - most bugs in most programs don’t manifest as crashes
    - an implicit oracle cannot detect such bugs!
- compare to the way that humans write tests:
  - select an input
  - select an oracle
  - compare the two

Implicit oracles: a key weakness

- limited to facts that are true about **all programs**
    - most bugs in most programs don’t manifest as crashes
        - an implicit oracle cannot detect such bugs!
- compare to the way that humans write tests:
    - select ***an*** input
    - select ***an*** oracle
    - compare the two

Implicit oracles: a key weakness

- limited to facts that are true about **all programs**
  - most bugs in most programs don’t manifest as crashes
    - an implicit oracle cannot detect such bugs!
- compare to the way that humans write tests:
  - select ***an*** input
  - select ***an*** oracle
  - compare the two
- that is, human testing usually **samples** the concrete behaviors of a program

Can we do better than sampling?

Can we do better than sampling?

- the key idea behind all of the techniques for producing better oracles that we'll discuss today is each gives us a **more general way** to describe what the program should do

Can we do better than sampling?

- the key idea behind all of the techniques for producing better oracles that we'll discuss today is each gives us a **more general way** to describe what the program should do
  - we call these ***partial oracles***, because they are **less specific** about what exactly the program should do than traditional, human-written oracles

Can we do better than sampling?

- the key idea behind all of the techniques for producing better oracles that we'll discuss today is each gives us a **more general way** to describe what the program should do
  - we call these ***partial oracles***, because they are **less specific** about what exactly the program should do than traditional, human-written oracles
  - you can view a partial oracle as an **abstraction** of testing:

Can we do better than sampling?

- the key idea behind all of the techniques for producing better oracles that we'll discuss today is each gives us a more general way to describe what the program should do
    - we call these ***partial oracles***, because they are less specific about what exactly the program should do than traditional, human-written oracles
    - you can view a partial oracle as an **abstraction** of testing:
        - concrete (traditional) oracle: **x = 5**

Can we do better than sampling?

- the key idea behind all of the techniques for producing better oracles that we'll discuss today is each gives us a **more general way** to describe what the program should do
  - we call these ***partial oracles***, because they are **less specific** about what exactly the program should do than traditional, human-written oracles
  - you can view a partial oracle as an **abstraction** of testing:
    - concrete (traditional) oracle: **$x = 5$**
    - abstract (partial) oracle: **$\forall x : x > 0$**

Can we do better than sampling?

- the key idea behind all of the techniques for producing better oracles that we'll discuss today is each gives us a **more general way** to describ
  - we cal re **less specific** about than traditional, humar
  - you can view a partial oracle as an **abstraction** of testing:
    - concrete (traditional) oracle: $x = 5$
    - abstract (partial) oracle: $\forall x : x > 0$

Today's key theme: combine test input generation (e.g., fuzzing) with **abstract, partial oracles**

Sources of partial oracles

Sources of partial oracles

- Option 1: **ask a human** to write a partial oracle instead of a concrete oracle. Humans turn out to be pretty good at this.
    - leads to ***property-based testing***

Sources of partial oracles

- Option 1: **ask a human** to write a partial oracle instead of a concrete oracle. Humans turn out to be pretty good at this.
  - leads to ***property-based testing***
- Option 2: exploit **known relationships** between different inputs or programs (humans provide the relationships)
  - leads to ***metamorphic testing***

Sources of partial oracles

- Option 1: **ask a human** to write a partial oracle instead of a concrete oracle. Humans turn out to be pretty good at this.
  - leads to ***property-based testing***
- Option 2: exploit **known relationships** between different inputs or programs (humans provide the relationships)
  - leads to ***metamorphic testing***
- Option 3: run the program and **automatically observe invariants** that happen to be true on human-written tests
  - leads to ***dynamic invariant detection***

Agenda: remainder of today’s lecture

- **Property-based testing**
- Metamorphic testing
- Dynamic invariant detection

Property-based testing

Definition: ***property-based testing*** (PBT) is a testing technique in which a human writes a partial oracle that is specific to the system under test

Property-based testing

Definition: ***property-based testing*** (PBT) is a testing technique in which a human writes a partial oracle that is specific to the system under test

- almost always paired with random input generation
    - can be viewed as “fuzzing, but using a human-written, program-specific oracle instead of an implicit oracle”

Property-based testing

Definition: ***property-based testing*** (PBT) is a testing technique in which a human writes a partial oracle that is specific to the system under test

- almost always paired with random input generation
  - can be viewed as "fuzzing, but using a human-written, program-specific oracle instead of an implicit oracle"
- note that PBT **requires** knowledge about the system being tested
  - if you can apply a partial oracle to *any* SUT, it's probably an implicit oracle instead

Property-based testing: benefits

Property-based testing: benefits

- Tests can have a **clear, mathematical** presentation
  - makes it easier for future developers to understand what is and is not being tested

Property-based testing: benefits

- Tests can have a **clear, mathematical** presentation
  - makes it easier for future developers to understand what is and is not being tested
- Can avoid finding and writing every case for each property
  - allows tester to focus on **the what not the how**

Property-based testing: benefits

- Tests can have a **clear, mathematical** presentation
  - makes it easier for future developers to understand what is and is not being tested
- Can avoid finding and writing every case for each property
  - allows tester to focus on **the what not the how**
- Can **decrease maintenance costs** with the same (or sometimes even greater!) coverage

Property-based testing in practice

- Historically, PBT was developed first for **functional languages**
  - Originated with QuickCheck for Haskell in 2000
  - PBT has the same kind of **mathematical vibe** as FP

Property-based testing in practice

- Historically, PBT was developed first for **functional languages**
  - Originated with QuickCheck for Haskell in 2000
  - PBT has the same kind of **mathematical vibe** as FP
- Now there are PBT frameworks available for mainstream programming languages
  - Hypothesis for Python and Java (https://hypothesis.works/)
  - DeepState for C/C++ (https://github.com/trailofbits/deepstate)
  - etc.

Agenda: remainder of today’s lecture

- Property-based testing
- **Metamorphic testing**
- Dynamic invariant detection

Metamorphic testing*

Definition: ***metamorphic testing*** is a property-based testing technique in which oracles are defined by ***metamorphic relations*** (MRs) between related inputs or programs

** 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.*

Metamorphic testing*

Definition: ***metamorphic testing*** is a property-based testing technique in which oracles are defined by ***metamorphic relations*** (MRs) between related inputs or programs

- we're using **relation** here in the mathematical sense:

** 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.*

Metamorphic testing*

Definition: ***metamorphic testing*** is a property-based testing technique in which oracles are defined by ***metamorphic relations*** (MRs) between related inputs or programs

- we're using **relation** here in the mathematical sense:
  - formally, a relation *R* over a set *X* can be seen as a set of ordered pairs (x,y) of members of *X*. The relation *R* holds between x and y if (x,y) is a member of *R*. [Wikipedia]

** 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.*

Metamorphic testing: programs vs inputs

- We’re going to cover two kinds of metamorphic testing today

Metamorphic testing: programs vs inputs

- We’re going to cover two kinds of metamorphic testing today
    - metamorphic testing where the outputs of **two related programs on the same input** have a metamorphic relationship
        - traditionally called ***differential testing***
        - today’s reading on CSmith is an example of this

Metamorphic testing: programs vs inputs

- We’re going to cover two kinds of metamorphic testing today
  - metamorphic testing where the outputs of **two related programs on the same input** have a metamorphic relationship
    - traditionally called ***differential testing***
    - today’s reading on CSmith is an example of this
  - metamorphic testing where the output of **the same program on two related inputs** have a metamorphic relationship
    - this is usually what’s meant by “metamorphic testing” in the literature

Metamorphic testing: differential testing

Metamorphic testing: differential testing

Observation: there are many programs with **similar or identical specifications**

Metamorphic testing: differential testing

Observation: there are many programs with **similar or identical specifications**

- if we are building such a program, we can use **another implementation** as an oracle

Metamorphic testing: differential testing

Observation: there are many programs with **similar or identical specifications**

- if we are building such a program, we can use **another implementation** as an oracle
- e.g., if we're writing a C compiler, we can compare our output to gcc

Metamorphic testing: differential testing

Observation: there are many programs with **similar or identical specifications**

- if we are building such a program, we can use **another implementation** as an oracle
- e.g., if we're writing a C compiler, we can compare our output to gcc

Definition: ***differential testing*** is a technique for testing two related programs by comparing their output on generated test inputs. Any difference indicates non-conformance in one of the two.

Metamorphic testing: differential testing

Advantages and disadvantages of differential testing:

Metamorphic testing: differential testing

Advantages and disadvantages of differential testing:

- only applicable in **limited situations**: need another implementation

Metamorphic testing: differential testing

Advantages and disadvantages of differential testing:

- only applicable in **limited situations**: need another implementation
    - but **useful more often than you might think** - for example, when writing a “fast” version of a routine, you can compare its output to a “slow” but easy-to-implement version

Metamorphic testing: differential testing

Advantages and disadvantages of differential testing:

- only applicable in **limited situations**: need another implementation
  - but **useful more often than you might think** - for example, when writing a “fast” version of a routine, you can compare its output to a “slow” but easy-to-implement version
- a human needs to decide **which of the two is correct**

Metamorphic testing: differential testing

Advantages and disadvantages of differential testing:

- only applicable in **limited situations**: need another implementation
    - but **useful more often than you might think** - for example, when writing a "fast" version of a routine, you can compare its output to a "slow" but easy-to-implement version
- a human needs to decide **which of the two is correct**
    - and sometimes neither is!

Metamorphic testing: differential testing

Advantages and disadvantages of differential testing:

- only applicable in **limited situations**: need another implementation
  - but **useful more often than you might think** - for example, when writing a “fast” version of a routine, you can compare its output to a “slow” but easy-to-implement version
- a human needs to decide **which of the two is correct**
  - and sometimes neither is!
- but, differential testing provides a much stronger oracle than most other techniques (true of metamorphic testing generally!)

Metamorphic testing: differential testing MR

- What’s the **metamorphic relation** in differential testing?
    - hint: think about how the outputs of e.g., two C compilers are related

Metamorphic testing: differential testing MR

- What's the **metamorphic relation** in differential testing?
  - hint: think about how the outputs of e.g., two C compilers are related
  - it's the **identity function**: we're checking if the two programs have the same output!
    - this is the **most common** MR! But not the only one…

Metamorphic testing: differential testing MR

- What's the **metamorphic relation** in differential testing?
  - hint: think about how the outputs of e.g., two C compilers are related
  - it's the **identity function**: we're checking if the two programs have the same output!
    - this is the **most common** MR! But not the only one…
- What **other MRs** could we use for differential testing?

Metamorphic testing: differential testing MR

- What's the metamorphic relation in differential testing?
    - hint: think about how the outputs of e.g., two C compilers are related
    - it's the identity function: we're checking if the two programs have the same output!
        - this is the most common MR! But not the only one…
- What other MRs could we use for differential testing?
    - Inversion: forall X. unzip(zip(X)) = X
    - Convergence / Idempotency: forall X. sort(sort(X)) = sort(X)

Aside: designing for testing: tests for free

- Many programs transform data from one format to another (cf. adapter design pattern)

Aside: designing for testing: tests for free

- Many programs transform data from one format to another (cf. adapter design pattern)
- If the program is implementing a function with similar domain and range, you can often get high-coverage tests “for free” by **composing the program with itself**

Aside: designing for testing: tests for free

- Many programs transform data from one format to another (cf. adapter design pattern)
- If the program is implementing a function with similar domain and range, you can often get high-coverage tests “for free” by **composing the program with itself**
    - If possible, design your program so that this is possible

Metamorphic testing: related inputs

Metamorphic testing: related inputs

- **Simple case**: related inputs with identical outcomes
  - expected output for a given input is unknown

Metamorphic testing: related inputs

- **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: forall x, abs(x) == abs(-x)

Metamorphic testing: related inputs

- **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: forall x, abs(x) == abs(-x)
- **Generalization**: related inputs and related outputs

Metamorphic testing: related inputs

- **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: forall x, abs(x) == abs(-x)
- **Generalization**: related inputs and related outputs
  - Input i_1 yields (unknown) o_1 (***initial input***)

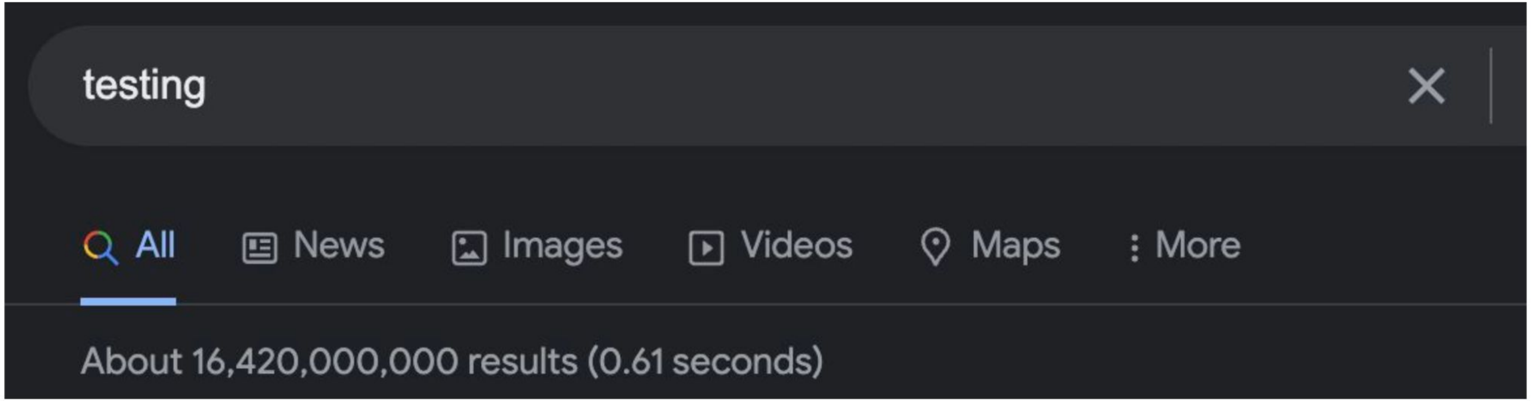
Metamorphic testing: related inputs

- **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: forall x, abs(x) == abs(-x)
- **Generalization**: related inputs and related outputs
  - Input i_1 yields (unknown) o_1 (***initial input***)
  - $R_i : i_1 \rightarrow i_2$ (***follow-up input***)

Metamorphic testing: related inputs

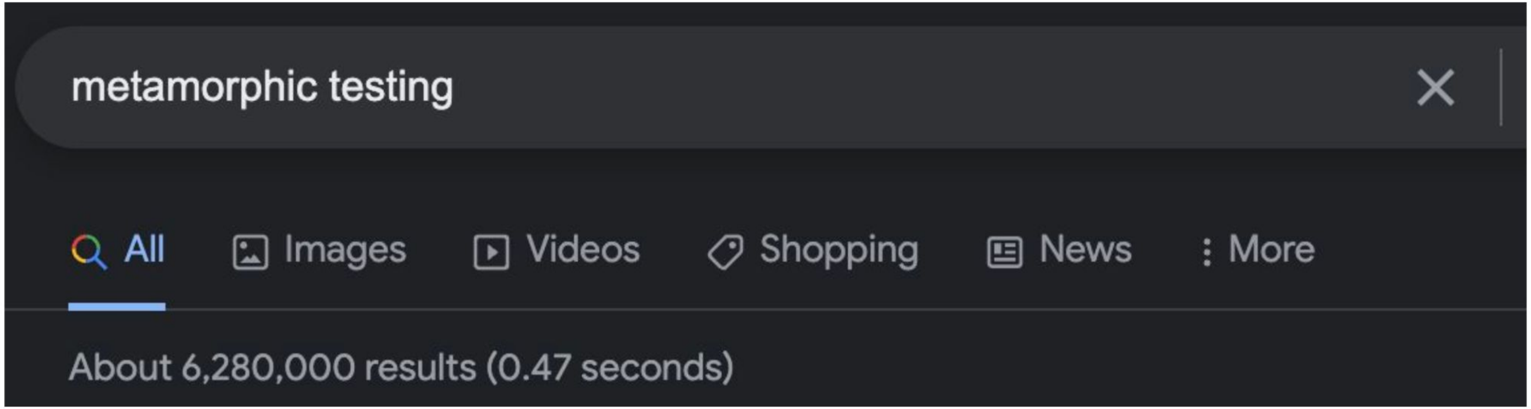
- **Simple case**: related inputs with identical outcomes
  - expected output for a given input is unknown
  - two related inputs must result in the same output
    - example: forall x, abs(x) == abs(-x)
- **Generalization**: related inputs and related outputs
  - Input i_1 yields (unknown) o_1 (***initial input***)
  - $R_i : i_1 \text{ -> } i_2$ (***follow-up input***)
  - $R_o : o_1 \text{ -> } o_2$ (***necessary condition***)

Metamorphic testing: online service example



testing

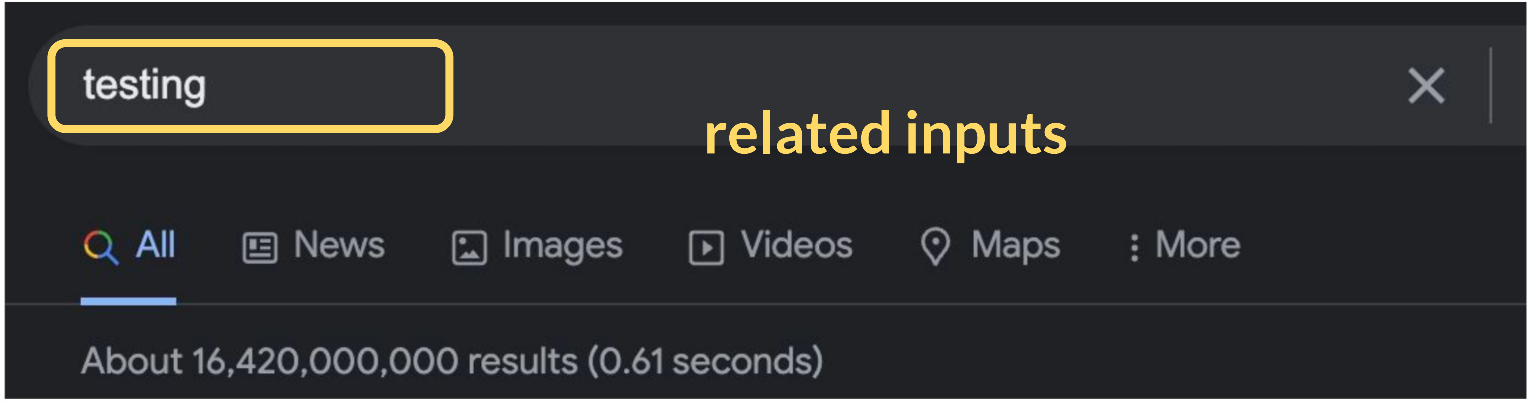
About 16,420,000,000 results (0.61 seconds)



metamorphic testing

About 6,280,000 results (0.47 seconds)

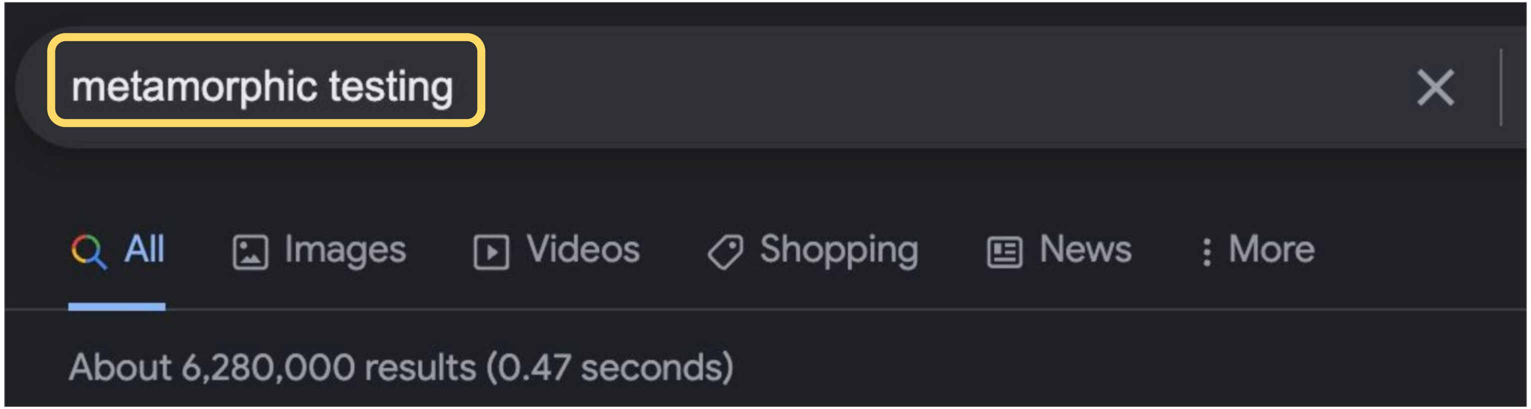
Metamorphic testing: online service example



testing

related inputs

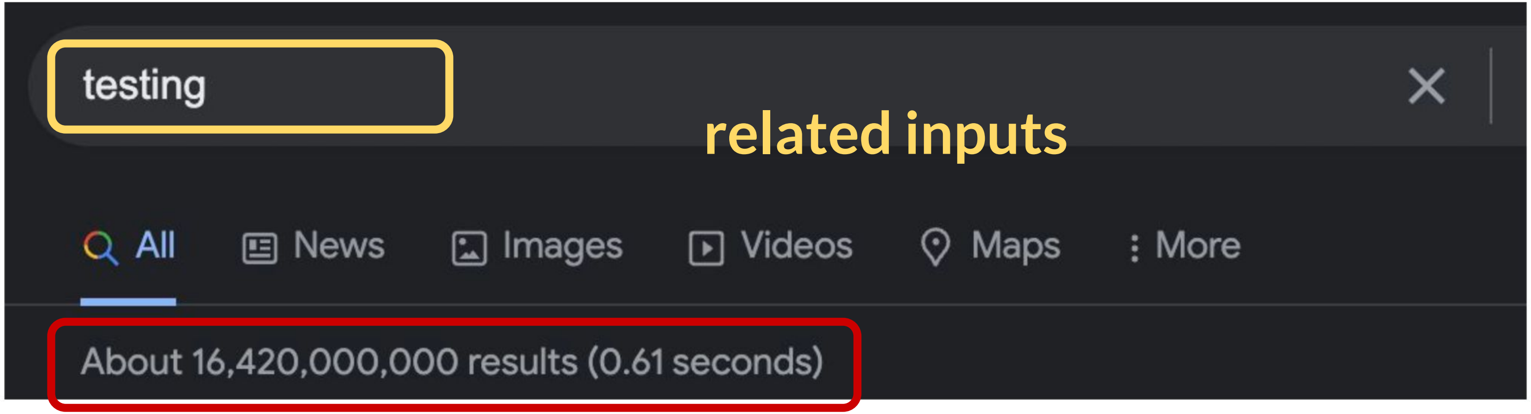
About 16,420,000,000 results (0.61 seconds)



metamorphic testing

About 6,280,000 results (0.47 seconds)

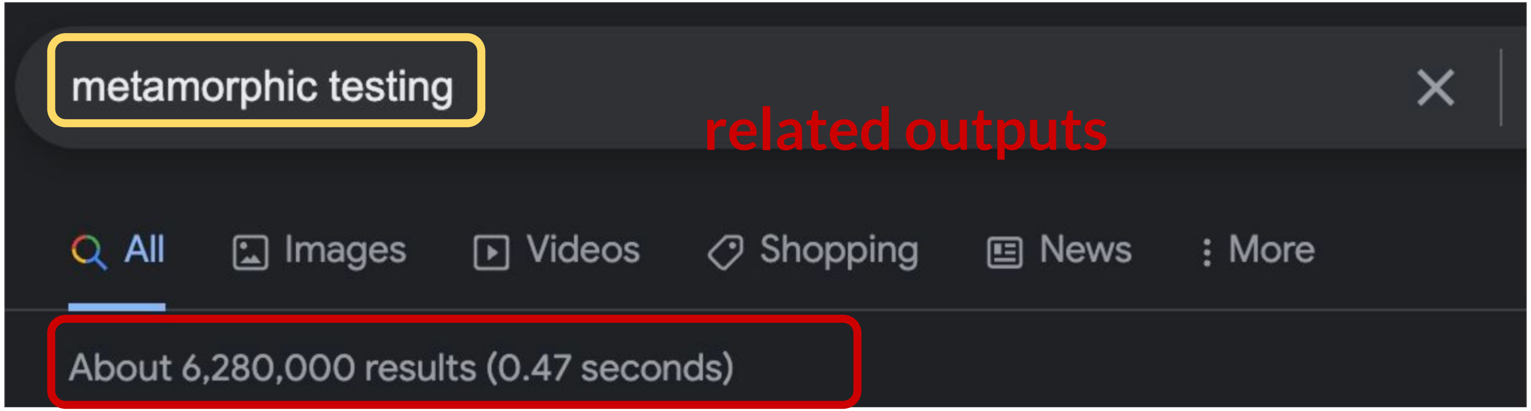
Metamorphic testing: online service example



testing

related inputs

About 16,420,000,000 results (0.61 seconds)

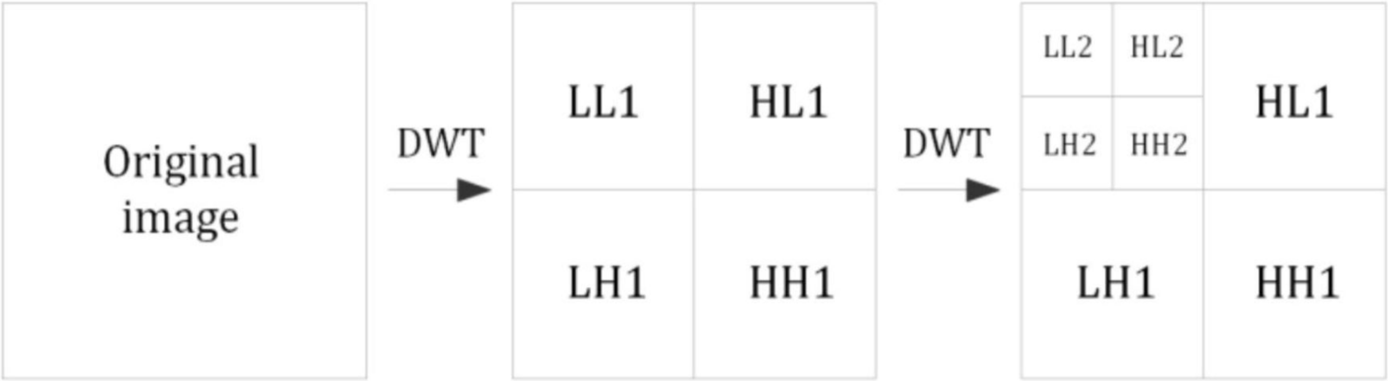


metamorphic testing

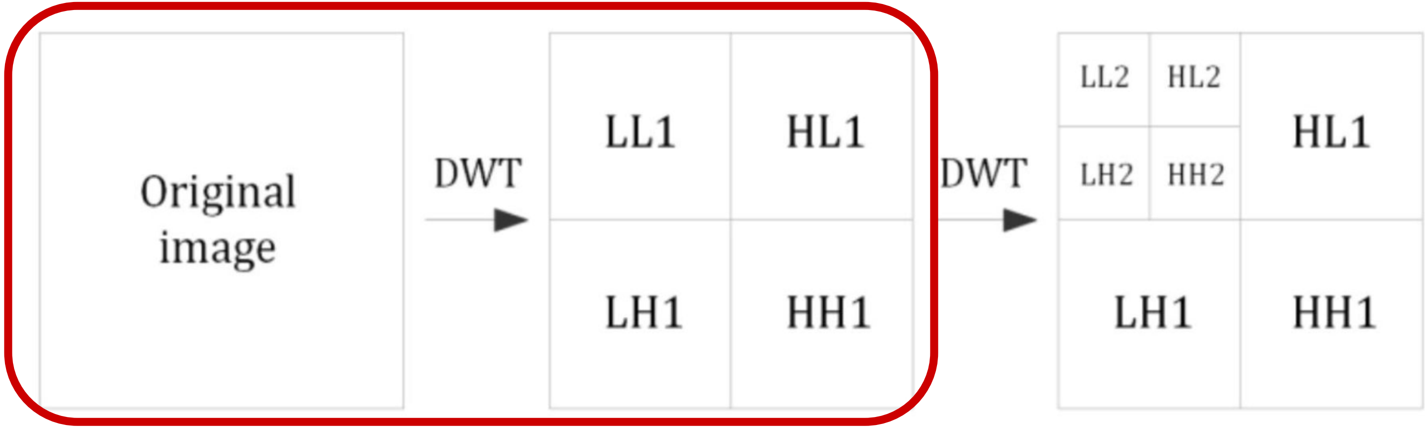
related outputs

About 6,280,000 results (0.47 seconds)

MT: discrete wavelet transform example



MT: discrete wavelet transform example



Original image

DWT

LL1 HL1

LH1 HH1

DWT

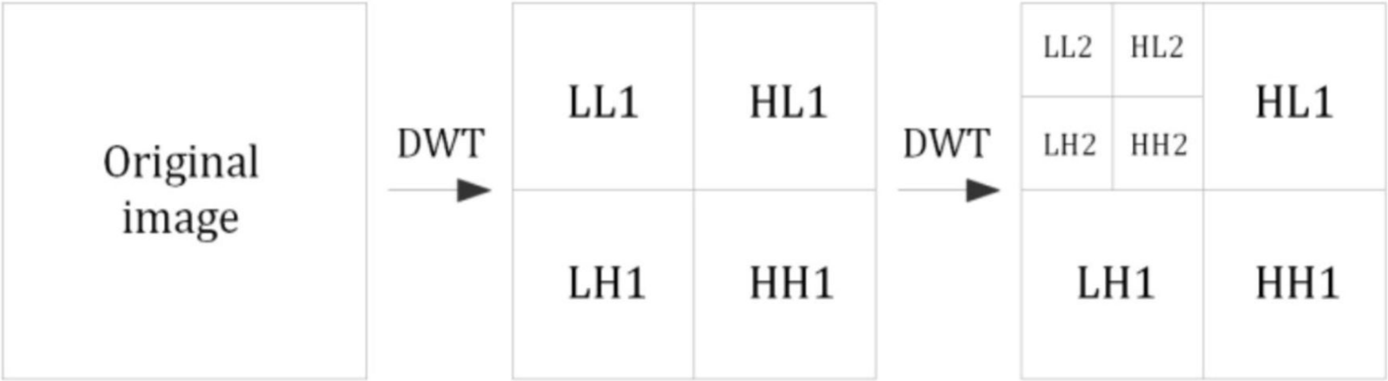
LL2 HL2

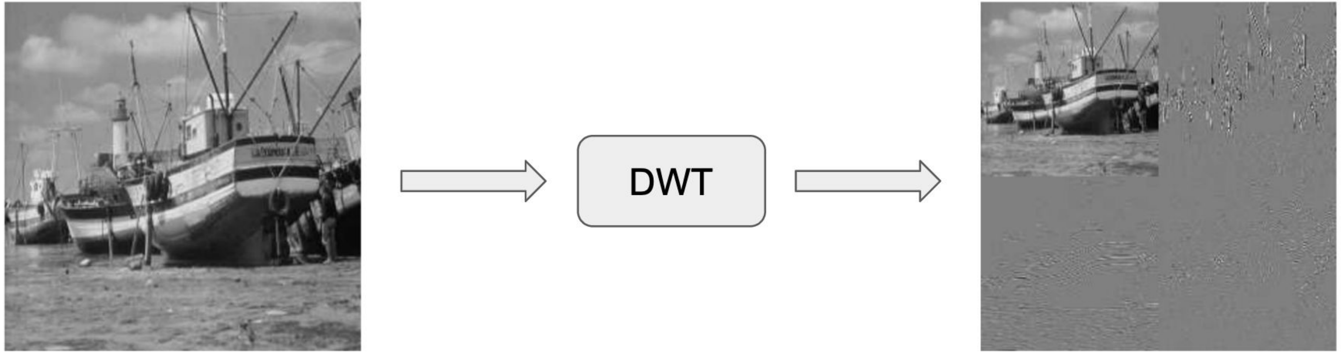
LH2 HH2

HL1

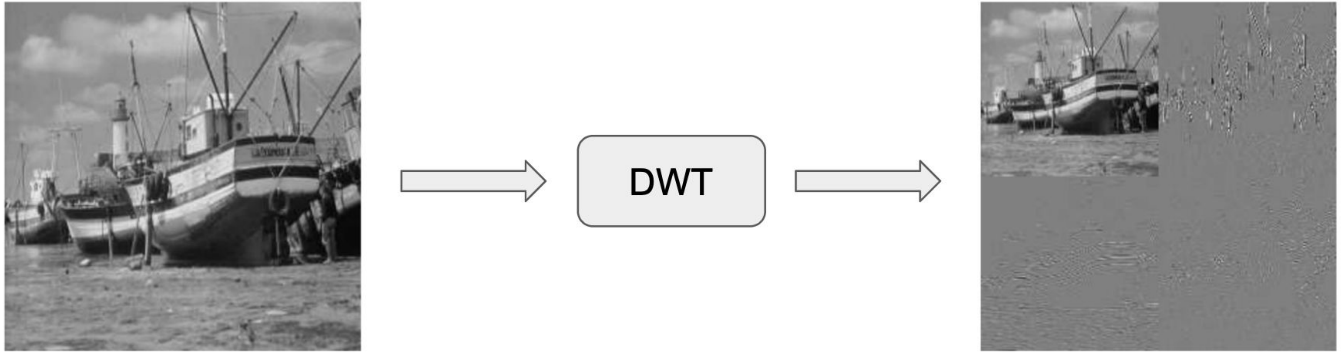
LH1 HH1

MT: discrete wavelet transform example

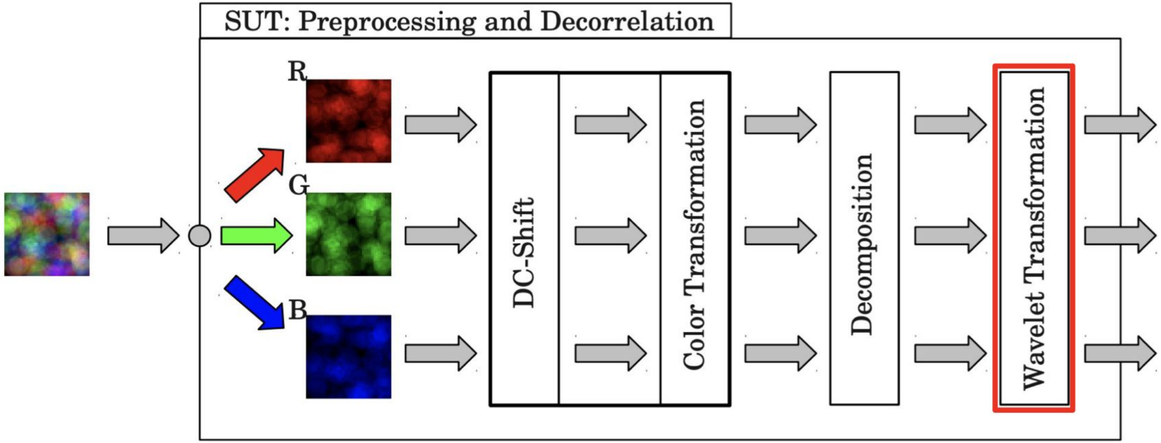




MT: DWT: concrete SUT: jpeg2000 encoder



DWT



SUT: Preprocessing and Decorrelation

R

G

B

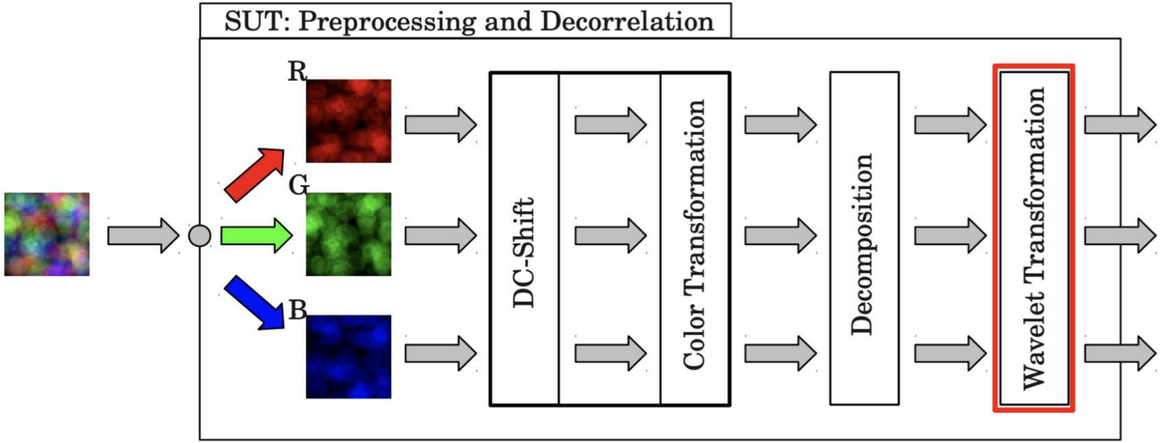
DC-Shift

Color Transformation

Decomposition

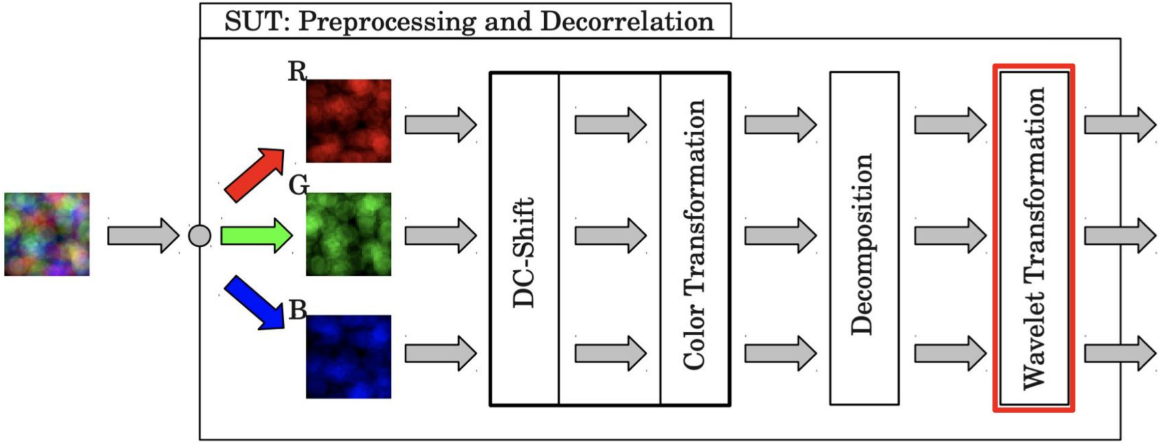
Wavelet Transformation

MT: DWT: concrete SUT: jpeg2000 encoder



Metamorphic testing has **three requirements**:

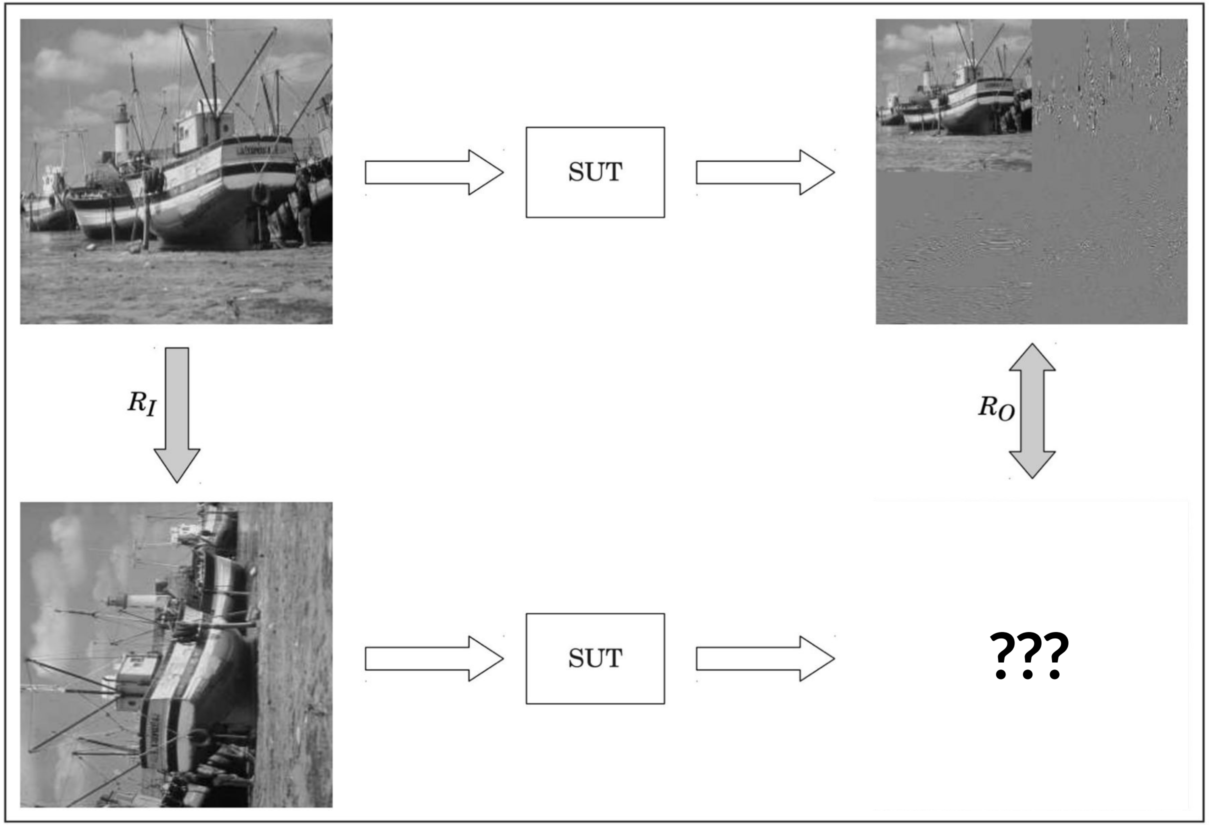
MT: DWT: concrete SUT: jpeg2000 encoder



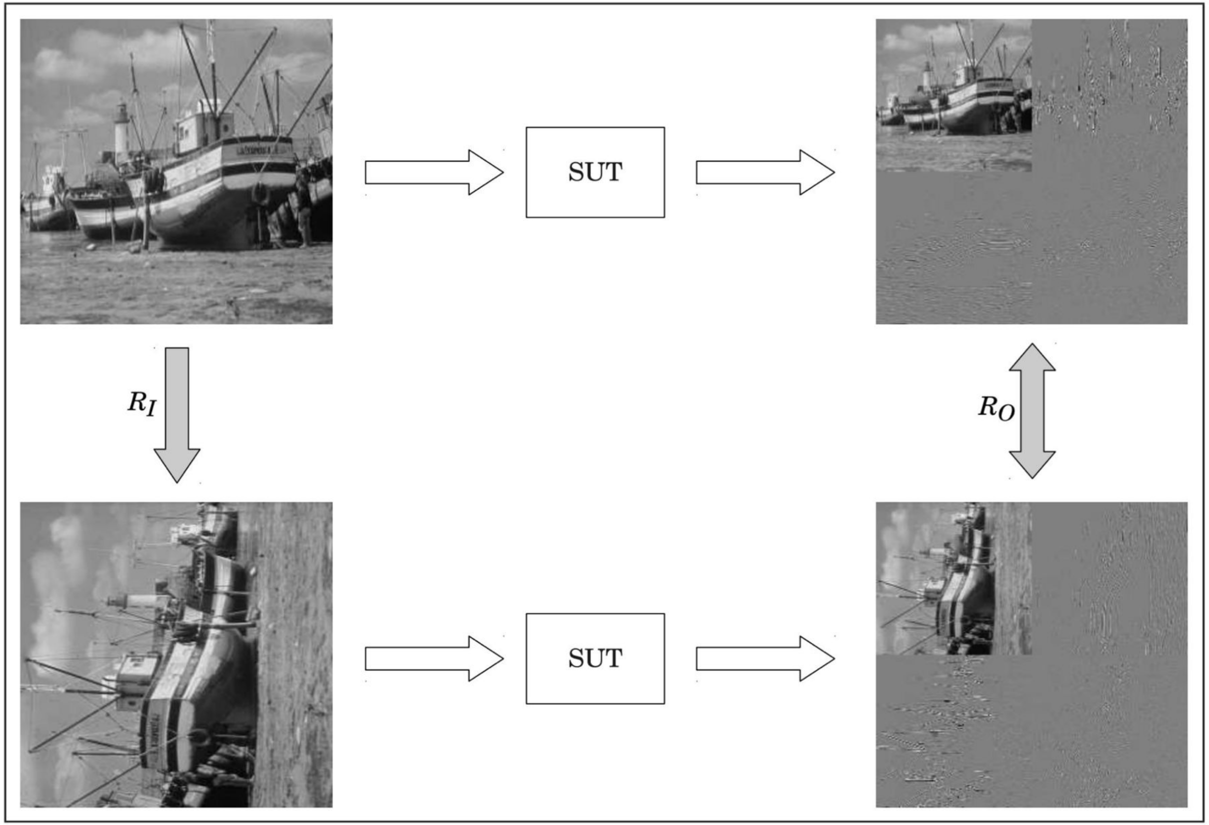
Metamorphic testing has **three requirements**:

- a set of initial inputs (or a generator)
- a relation R_i that can generate follow-up inputs
- a relation R_o that gives the necessary correctness condition

MT: DWT: relations R_i and R_o



MT: DWT: relations R_i and R_o



MT: DWT: concrete SUT: jpeg2000 encoder

1. R_i: Transpose the input image
   R_o: ???

MT: DWT: concrete SUT: jpeg2000 encoder

1. R_i: Transpose the input image
   R_o: The output components must also be transposed

MT: DWT: concrete SUT: jpeg2000 encoder

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

MT: DWT: concrete SUT: jpeg2000 encoder

1. R_i: Transpose the input image
   R_o: The output components m
2. R_i: Add a constant to all color
   R_o: Only the DC components
3. R_i: Invert the color values
   R_o: The color values of the ou
4. R_i: Enlarge the input image ("z
   R_o: The output components m

Some notes:

- these MRs are very **program-specific**
  - domain knowledge!
  - some MRs have interesting properties
    - e.g. MR 1 is commutative!
  - MR compositions are effective in practice

MT: DWT: concrete SUT: jpeg2000 encoder

1. R_i: Transpose the input image
   R_o: The output components m
2. R_i: Add a constant to all color
   R_o: Only the DC components
3. R_i: Invert the color values
   R_o: The color values of the ou
4. R_i: Enlarge the input image (“
   R_o: The output components m

Some notes:

- these MRs are very **program-specific**
  - domain knowledge!
- some MRs have interesting properties
  - e.g., MR 1 is **commutative**!
  - MR compositions are effective in practice

MT: DWT: concrete SUT: jpeg2000 encoder

1. R_i: Transpose the input image
   R_o: The output components m
2. R_i: Add a constant to all color
   R_o: Only the DC components
3. R_i: Invert the color values
   R_o: The color values of the ou
4. R_i: Enlarge the input image (“
   R_o: The output components m

Some notes:

- these MRs are very **program-specific**
  - domain knowledge!
- some MRs have interesting properties
  - e.g., MR 1 is **commutative**!
- **MR compositions** are effective in practice

Metamorphic testing: practicality

Metamorphic testing: practicality

- One of the **most effective** ways to test real systems
    - especially when combined with a fuzzer for random input generation

Metamorphic testing: practicality

- One of the **most effective** ways to test real systems
  - especially when combined with a fuzzer for random input generation
- Often **difficult to apply**
  - designing MRs requires **domain expertise**
  - but easier for some kinds of systems than others

Metamorphic testing: practicality

- One of the **most effective** ways to test real systems
  - especially when combined with a fuzzer for random input generation
- Often **difficult to apply**
  - designing MRs requires **domain expertise**
  - but easier for some kinds of systems than others
- My advice: **always** be on the lookout for opportunities to carry out metamorphic testing
  - **great value** in terms of increasing your confidence in a system's correctness vs effort you need to put in!

Agenda: remainder of today's lecture

- Property-based testing
- Metamorphic testing
- **Dynamic invariant detection**

Dynamic invariant detection: intuition

Observation: programs **usually** behave correctly

Dynamic invariant detection: intuition

Observation: programs **usually** behave correctly

- e.g., if I have a human-written test suite with ten tests, and we have `index == array_len - 1` in **every test**

Dynamic invariant detection: intuition

Observation: programs **usually** behave correctly

- e.g., if I have a human-written test suite with ten tests, and we have `index == array_len - 1` in **every test**
- then maybe the correct oracle is that on **every input** we should have `index == array_len - 1`

Dynamic invariant detection: intuition

Observation: programs **usually** behave correctly

- e.g., if I have a human-written test suite with ten tests, and we have `index == array_len - 1` in **every test**
- then maybe the correct oracle is that on **every input** we should have `index == array_len - 1`

Definition: an ***invariant*** is a predicate over program expressions that is true on every execution

Dynamic invariant detection: intuition

Observation: programs **usually** behave correctly

- e.g., if I have a human-written test suite with ten tests, and we have `index == array_len - 1` in **every test**
- then maybe the correct oracle is that on **every input** we should have `index == array_len - 1`

Definition: an ***invariant*** is a predicate over program expressions that is true on every execution

- high-quality invariants can serve as test oracles

Background: forward and backward reasoning

There are two ways to reason about what a program does:

Background: forward and backward reasoning

There are two ways to reason about what a program does:

- ***forward reasoning***:

Background: forward and backward reasoning

There are two ways to reason about what a program does:

- ***forward reasoning***: knowing a fact that is true **before** execution, …

Background: forward and backward reasoning

There are two ways to reason about what a program does:

- ***forward reasoning***: knowing a fact that is true **before** execution, and reasoning about what must be true **after** execution

Background: forward and backward reasoning

There are two ways to reason about what a program does:

- ***forward reasoning***: knowing a fact that is true **before** execution, and reasoning about what must be true **after** execution
    - given a **precondition**, what **postcondition**(s) are true?

Aside: pre- and postconditons

Aside: pre- and postconditons

Definition: a ***precondition*** (to a function) is a condition that must be true when entering (the function).

- it may (but does not have to) include expectations about the arguments

Aside: pre- and postconditons

Definition: a ***precondition*** (to a function) is a condition that must be true when entering (the function).

- it may (but does not have to) include expectations about the arguments

Definition: a ***postcondition*** (to a function) is a condition that must be true when leaving (the function)

- it may (but does not have to) include expectations about the return value (of the function) or about side-effects

Background: forward and backward reasoning

There are two ways to reason about what a program does:

- ***forward reasoning***: knowing a fact that is true **before** execution, and reasoning about what must be true **after** execution
    - given a **precondition**, what **postcondition**(s) are true?

Background: forward and backward reasoning

There are two ways to reason about what a program does:

- ***forward reasoning***: knowing a fact that is true **before** execution, and reasoning about what must be true **after** execution
    - given a **precondition**, what **postcondition**(s) are true?
- ***backward reasoning***:

Background: forward and backward reasoning

There are two ways to reason about what a program does:

- ***forward reasoning***: knowing a fact that is true **before** execution, and 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, …

Background: forward and backward reasoning

There are two ways to reason about what a program does:

- ***forward reasoning***: knowing a fact that is true **before** execution, and 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, and reasoning about what must be true **before** execution

Background: forward and backward reasoning

There are two ways to reason about what a program does:

- ***forward reasoning***: knowing a fact that is true **before** execution, and 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, and reasoning about what must be true **before** execution
  - given a **postcondition**, what **precondition**(s) are true?

Pros and cons: forward vs backward reasoning

Forward reasoning:

Backward reasoning:

Pros and cons: forward vs 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 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

Dynamic invariant detection: insight

- Given a program location, if we could **infer** an invariant for that location, we could have …

Dynamic invariant detection: insight

- Given a program location, if we could **infer** an invariant for that location, we could have …
  - Function **preconditions** (location = entry)
  - Function **postconditions** (location = exit)
  - Loop invariants (location = loop entry)

Dynamic invariant detection: insight

- Given a program lo
  location, we could l
  - Function **preco**
  - Function **postc**
  - Loop invariants (location = loop entry)

A ***loop invariant*** is an invariant that must hold at both the start and end of each iteration of the loop. We'll come back to this concept later in the semester, but for now don't worry too much about it.

Dynamic invariant detection: insight

- Given a program location, if we could **infer** an invariant for that location, we could have …
  - Function **preconditions** (location = entry)
  - Function **postconditions** (location = exit)
  - Loop invariants (location = loop entry)
- Can we do this **automatically**?

Dynamic invariant detection: insight

- Given a program location, if we could **infer** an invariant for that location, we could have …
  - Function **preconditions** (location = entry)
  - Function **postconditions** (location = exit)
  - Loop invariants (location = loop entry)
- Can we do this **automatically**?
- Two insights:

Dynamic invariant detection: insight

- Given a program location, if we could **infer** an invariant for that location, we could have …
  - Function **preconditions** (location = entry)
  - Function **postconditions** (location = exit)
  - Loop invariants (location = loop entry)
- Can we do this **automatically**?
- Two insights:
  - An invariant **always holds** on all executions

Dynamic invariant detection: insight

- Given a program location, if we could **infer** an invariant for that location, we could have …
    - Function **preconditions** (location = entry)
    - Function **postconditions** (location = exit)
    - Loop invariants (location = loop entry)
- Can we do this **automatically**?
- Two insights:
    - An invariant **always holds** on all executions
    - We can **detect** spurious false invariants

Dynamic invariant detection: high-level idea

- What if we require that the program come equipped with inputs?
  - An indicative workload
  - High-coverage test cases

Dynamic invariant detection: high-level idea

- What if we require that the program come equipped with inputs?
  - An indicative workload
  - High-coverage test cases
- Since an invariant holds on **every execution** (by definition), any candidate invariant that is false **even once** can be tossed out!

Dynamic invariant detection: high-level idea

- What if we require that the program come equipped with inputs?
  - An indicative workload
  - High-coverage test cases
- Since an invariant holds on **every execution** (by definition), any candidate invariant that is false **even once** can be tossed out!
- Plan:

Dynamic invariant detection: high-level idea

- What if we require that the program come equipped with inputs?
  - An indicative workload
  - High-coverage test cases
- Since an invariant holds on **every execution** (by definition), any candidate invariant that is false **even once** can be tossed out!
- Plan:
    1. **generate** many candidate invariants

Dynamic invariant detection: high-level idea

- What if we require that the program come equipped with inputs?
  - An indicative workload
  - High-coverage test cases
- Since an invariant holds on **every execution** (by definition), any candidate invariant that is false **even once** can be tossed out!
- Plan:
  1. **generate** many candidate invariants
  2. **filter out** the false ones by running the tests!

Dynamic invariant detection: naive approach

- Given:

```
    while b do c
```

Dynamic invariant detection: naive approach

- Given:

```
    while b do c
```

- Instrument:

```
    while b do print Inv1; print Inv2; ... ; c
```

Dynamic invariant detection: naive approach

- Given:

```
    while b do c
```

- Instrument:

```
    while b do print Inv1; print Inv2; ... ; c
```

- Then just run the tests and filter out those that are false

Dynamic invariant detection: naive approach

- Given:

```
    while b do c
```

- Instrument:

```
    while b do print Inv1; print Inv2; ... ; c
```

- Then just run the tests and filter out those that are false
- What’s **wrong** with this plan?

Dynamic invariant detection: naive approach

- Given:

```
    while b do c
```

- Instrument:

```
    while b do print Inv1; print Inv2; ... ; c
```

- Then just run the tests and filter out those that are false
- What’s **wrong** with this plan?
    - Hint: how many invariants are there?

Dynamic invariant detection: naive approach

- Given:

```
    while b do c
```

- Instrument:

```
    while b do print Inv1; print Inv2; ... ; c
```

- Then just run the tests and filter out those that are false
- What’s **wrong** with this plan?
  - Hint: how many invariants are there?
  - **infinitely many :(**

Dynamic invariant detection: templates

- **Key idea** to keep the set of invariants finite: use a set of ***template invariants*** that will likely be useful as oracles

Dynamic invariant detection: templates

- **Key idea** to keep the set of invariants finite: use a set of ***template invariants*** that will likely be useful as oracles
- For example, given program variables x, y, and z:

Dynamic invariant detection: templates

- **Key idea** to keep the set of invariants finite: use a set of ***template invariants*** that will likely be useful as oracles
- For example, given program variables x, y, and z:

$x = c$	*constants*	$x < y$	*ordering*
$x \neq 0$	*non-zero*	$(x + y) \% b = a$	*math*
$x >= c$	*bounds*	$z = ax + by + c$	*linear*
$y = ax + b$	*linear*		

[Ernst, M. D., Cockrell, J., Griswold, W. G., & Notkin, D. (2001). *Dynamically discovering likely program invariants to support program evolution*.]

Dynamic invariant detection: templates

- **Key idea** to keep the set of invariants finite: use a set of ***template invariants*** that will likely be useful as oracles
- For example, given program variables x, y, and z:

○ $x = c$	*constants*	$x < y$	*ordering*
○ $x != 0$	*non-zero*	$(x + y) \% b = a$	*math*
○ $x >= c$	*bounds*	$z = ax + by + c$	*linear*
○ $y = ax + b$	*linear*		

- At most three variables => finite number of invariants to check

[Ernst, M. D., Cockrell, J., Griswold, W. G., & Notkin, D. (2001). *Dynamically discovering likely program invariants to support program evolution.*]

Dynamic invariant detection: Daikon

The Daikon invariant detection algorithm:

[Ernst, M. D., Cockrell, J., Griswold, W. G., & Notkin, D. (2001). *Dynamically discovering likely program invariants to support program evolution.*]

Dynamic invariant detection: Daikon

The Daikon invariant detection algorithm:

- For every program location:
    - For **all triples** of in-scope variables:
        - Instantiate templates to obtain candidate invariants
        - Instrument program

[Ernst, M. D., Cockrell, J., Griswold, W. G., & Notkin, D. (2001). *Dynamically discovering likely program invariants to support program evolution*.]

Dynamic invariant detection: Daikon

The Daikon invariant detection algorithm:

- For every program location:
  - For **all triples** of in-scope variables:
    - Instantiate templates to obtain candidate invariants
    - Instrument program
- For every test case:
  - Run instrumented program
  - Filter out any falsified candidate invariant

[Ernst, M. D., Cockrell, J., Griswold, W. G., & Notkin, D. (2001). *Dynamically discovering likely program invariants to support program evolution.*]

Dynamic invariant detection: Daikon

The Daikon invariant detection algorithm:

- For every program location:
  - For **all triples** of in-scope variables:
    - Instantiate templates to obtain candidate invariants
    - Instrument program
- For every test case:
  - Run instrumented program
  - Filter out any falsified candidate invariant
- Report surviving invariants

[Ernst, M. D., Cockrell, J., Griswold, W. G., & Notkin, D. (2001). *Dynamically discovering likely program invariants to support program evolution.*]

Dynamic invariant detection: Daikon

The Daikon invariant detection algorithm:

- For every program location:
  - For **all triples** of in-scope variables:
    - Instantiate templates to obtain candidate invariants
    - Instrument program
- For every test case:
  - Run instrumented program
  - Filter out any falsified candid
- Report surviving invariants

What's the **running time** of the Daikon algorithm?

- cubic in in-scope variables
- linear in test suite size
- linear in program size

[Ernst, M. D., Cockrell, J., Griswold, W. G., & Notkin, D. (2001). *Dynamically discovering likely program invariants to support program evolution*.]

Dynamic invariant detection: Daikon

The Daikon invariant detection algorithm:

- For every program location:
  - For **all triples** of in-scope variables:
    - Instantiate templates to obtain candidate invariants
    - Instrument program
- For every test case:
  - Run instrumented program
  - Filter out any falsified candid
- Report surviving invariants

What's the **running time** of the Daikon algorithm?

- **cubic** in in-scope variables
- linear in test suite size,
- linear in program size

[Ernst, M. D., Cockrell, J., Griswold, W. G., & Notkin, D. (2001). *Dynamically discovering likely program invariants to support program evolution.*]

In-class exercise: infer likely invariants

In-class exercise: infer likely invariants

Program: (input= N >0)

```
i := 0
while i != N:
   i := i + 1
```

In-class exercise: infer likely invariants

Program: (input= N >0)

```
i := 0
while i != N:
    i := i + 1
```

Invariants to evaluate:

- i=0
- i<0
- i<=0
- i>0
- i>=0
- N=0
- N<0
- N<=0
- N>=0
- N>0
- i==N
- i<N
- i<=N
- i>N
- i>=N

In-class exercise: infer likely invariants

Program: (input= N >0)

```
i := 0
while i != N:
    i := i + 1
```

Evaluate invariants at program start, program end, and for the loop itself (i.e., loop invariants)

Invariants to evaluate:

- i=0
- i<0
- i<=0
- i>0
- i>=0
- N=0
- N<0
- N<=0
- N>=0
- N>0
- i==N
- i<N
- i<=N
- i>N
- i>=N

In-cla likely invariants

i is not in scope at the start of the program, so we don't need to evaluate invariants involving i

Progra

```
    i := 0
    while i != N:
        i := i + 1
```

Evaluate invariants at **program start**, program end, and for the loop itself (i.e., loop invariants)

Invariants to evaluate:

- i=0
- i<0
- i<=0
- i>0
- i>=0
- ~~N=0~~
- ~~N<0~~
- ~~N<=0~~
- **N>=0**
- **N>0**
- i==N
- i<N
- i<=N
- i>N
- i>=N

In-class exercise: infer likely invariants

Program: (input= N >0)

```
i := 0
while i != N:
    i := i + 1
```

Evaluate invariants at program start, **program end**, and for the loop itself (i.e., loop invariants)

Invariants to evaluate:

- ~~i=0~~
- ~~i<0~~
- ~~i<=0~~
- **i>0**
- **i>=0**
- ~~N=0~~
- ~~N<0~~
- ~~N<=0~~
- **N>=0**
- **N>0**
- **i==N**
- ~~i<N~~
- **i<=N**
- ~~i>N~~
- **i>=N**

In-cla likely invariants

in class we evaluated each invariant at both the start and end of the loop

Progra

```
i := 0
while i != N:
    i := i + 1
```

Evaluate invariants at program start, program end, and for the loop itself (i.e., **loop invariants**)

Invariants to evaluate:

- ~~i=0~~
- ~~i<0~~
- ~~i<=0~~
- ~~i>0~~
- **i>=0**
- ~~N=0~~
- ~~N<0~~
- ~~N<=0~~
- ~~N>=0~~
- ~~N>0~~
- ~~i==N~~
- ~~i<N~~
- **i<=N**
- ~~i>N~~
- ~~i>=N~~

Dynamic invariant detection: limitations

Dynamic invariant detection: limitations

- **False Negatives**!

Dynamic invariant detection: limitations

- **False Negatives**!
  - If your invariant does not fit a template, Daikon cannot find it
    - Example: *l + u – 1 <= 2p <= l + u* (binary search pivot)

Dynamic invariant detection: limitations

- **False Negatives**!
  - If your invariant does not fit a template, Daikon cannot find it
    - Example: $l + u - 1 <= 2p <= l + u$ (binary search pivot)
- Nothing prevents a Daikon-like algorithm from finding these

Dynamic invariant detection: limitations

- **False Negatives**!
    - If your invariant does not fit a template, Daikon cannot find it
        - Example: *l + u – 1 <= 2p <= l + u* (binary search pivot)
- Nothing prevents a Daikon-like algorithm from finding these
    - but **templates are absolutely necessary** to permit Daikon to scale
        - and each additional template **bloats the complexity** (especially if it involves more variables!)

Dynamic invariant detection: limitations

- False positives from **limited input**

Dynamic invariant detection: limitations

- False positives from **limited input**
  - if you only test your sorting program on the input [4, 2, 3,], Daikon will learn the invariant output[0] = 2

Dynamic invariant detection: limitations

- False positives from **limited input**
  - if you only test your sorting program on the input [4, 2, 3,], Daikon will learn the invariant output[0] = 2
  - but as we’ve learned, making high-coverage, high-adequacy test suites is easy, right? (haha, no)

Dynamic invariant detection: limitations

- False positives from **limited input**
    - if you only test your sorting program on the input [4, 2, 3,], Daikon will learn the invariant output[0] = 2
    - but as we've learned, making high-coverage, high-adequacy test suites is easy, right? (haha, no)
- False positives from **linguistic coincidence**

Dynamic invariant detection: limitations

- False positives from **limited input**
  - if you only test your sorting program on the input [4, 2, 3,], Daikon will learn the invariant output[0] = 2
  - but as we've learned, making high-coverage, high-adequacy test suites is easy, right? (haha, no)
- False positives from **linguistic coincidence**
  - e.g., `ptr % 4 == 0` or `x <= MAX_INT`

Dynamic invariant detection: limitations

- False positives from **limited input**
  - if you only test your sorting program on the input [4, 2, 3,], Daikon will learn the invariant output[0] = 2
  - but as we've learned, making high-coverage, high-adequacy test suites is easy, right? (haha, no)
- False positives from **linguistic coincidence**
  - e.g., `ptr % 4 == 0` or `x <= MAX_INT`
  - not false, but not related to correctness (or useful as an oracle)
    - these are true of any program!

HW5

- Two parts
  - run Daikon on a data structure of **your choice**
  - design some metamorphic relations for a real software system of **your choice**

HW5

- Two parts
  - run Daikon on a data structure of **your choice**
  - design some metamorphic relations for a real software system of **your choice**
- This homework expects you to make more decisions on your own than prior homeworks
  - that is, there are fewer guard rails
  - my advice: if you get stuck because of a difficulty with a system that you picked, remember that ***you can go back and choose a different system!*** (The course staff won't ever need to know!)