An Introduction to

Mutation Testing and Metamorphic Testing

Luigi Libero Lucio Starace

luigiliberolucio.starace@unina.it

Mutation Testing

What if I told you that you can start **killing mutants** to **test your tests**?

Testing your tests

How do you know if your test suite is **effective** or not?

- I wrote clean tests that run in isolation with Hamcrest and Mockito!
  - That's great, but it doesn't tell much on the effectiveness of the test suite
- I practiced TDD, so everything is tested!
  - Are you *really really really really really* sure of their effectiveness?
- My tests cover 100% of the branches!
  - Not very indicative. Coverage is good at telling us which parts are never covered, but tells very little on **how well** the covered parts are tested!

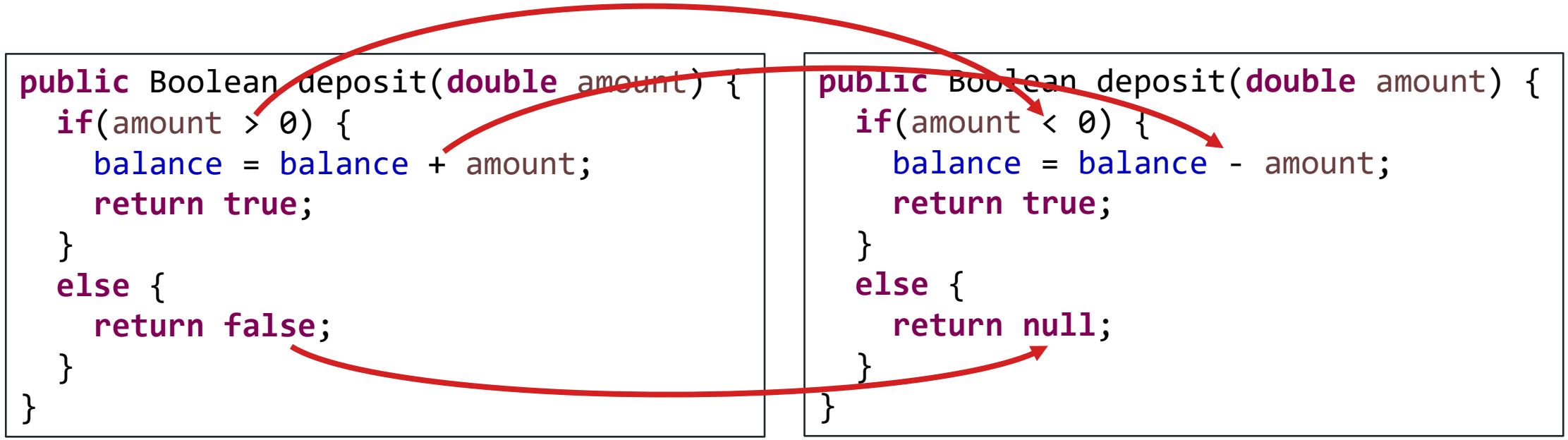
What is Mutation Testing?

- Mutation Testing is a technique to **evaluate** the **effectiveness** of test suites.
- Let’s start with some terminology

A Chitten, art from gyyporama.com

Software Mutations

A software mutation is a **small** change in your codebase



Software Mutants

- Don’t have super powers or adamantium bones (luckily!)
- They’re just **versions** of your software containing one or more mutations

Mutation Testing

- You have your software, and a nice and shiny test suite
- Generate (**automatically**) a lot of mutants for your codebase
- Run (**a selection of**) your tests on each mutant
- See what happens:
  - If one of the test fails, we say that the mutant is **killed** by the test suite
  - If all the tests pass, the mutant **escapes**
  - Timeout (mutant introduced an infinite loop!)
- A **good** test suite is one that kills a lot of mutants!

Not all mutants are bad

- An escaped mutant is not necessarily a bad thing
- Some mutants might be equivalent to the original
- Some might even happen to fix bugs in your code!

```
public boolean isSumZero(int[] array) {
    int sum = 0;
    for(int i : array)
        sum = sum + i;
    return sum == 0;
}
```

Equivalent mutant

```
public boolean isSumZero(int[] array) {
    int sum = 0;
    for(int i : array)
        sum = sum - i;
    return sum == 0;
}
```

Challenges

CPU intensive and **time consuming** (can be parallelized)

- Suppose your codebase has 100 classes and 1000 tests
- Suppose a test runs, on average, in 0.1 seconds
- Suppose that you generate 10 mutants for each class.
- If we ran the entire test suite on each mutant, it would take

$$100 \cdot 10 \cdot 1000 \cdot 0.1\ seconds = 100\,000\ seconds \approx 28\ hours$$

Test selection/prioritization techniques come in handy!

Mutation Testing in practice

- Infection (PHP) https://infection.github.io/
- Cosmic Ray (Python) https://github.com/sixty-north/cosmic-ray
- Stryker (JS, C#, Scala) https://stryker-mutator.io/
- PIT (Java) https://pitest.org/

PITest

- State-of-the-art Java mutation testing system
- Insert mutations through bytecode manipulation
- Several optimization heuristics
  - Only runs tests covering the mutated lines
  - Interrupts tests taking too long
- Large set of built-in mutators
- Easy integration with build tools/IDEs/CI-CD
- Detailed and human-readable reports

pitest.org

PIT demo

Let’s kill some mutants

Our codebase

```
public class SavingsAccount {
    private double balance;

    public SavingsAccount() { this.balance = 0; }

    public boolean deposit(double amount) {
        if(amount > 0) {
            this.balance = this.balance + amount;
            return true;
        }
        else {
            return false;
        }
    }
    /* ... */
}
```

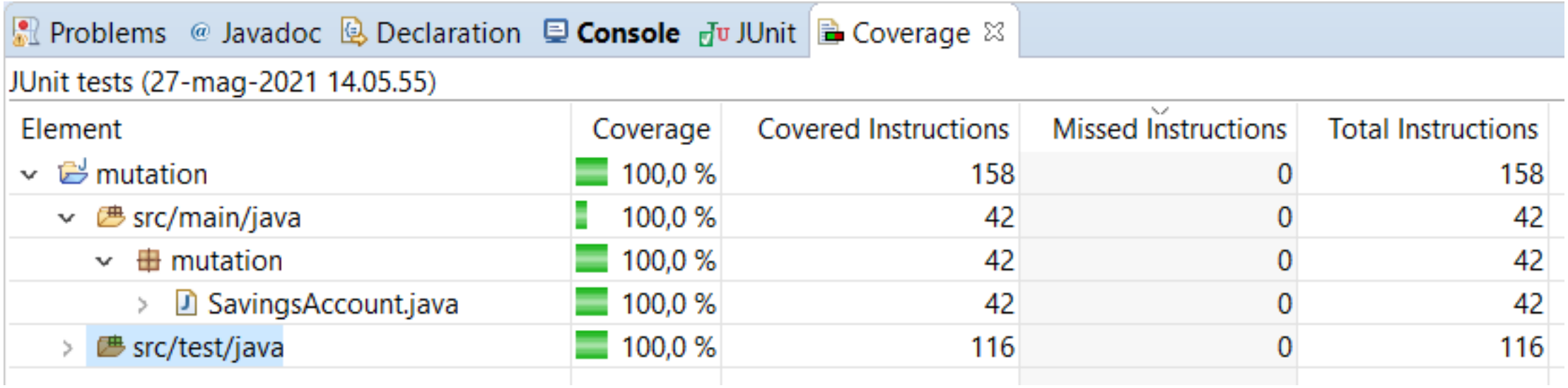
Our codebase

```
@Test
@DisplayName("Should correctly deposit a positive amount")
void shouldDepositCorrectly() {
    SavingsAccount s = new SavingsAccount();
    boolean ret = s.deposit(42.0);
    assertThat(ret,is(true));
    assertThat(s.getBalance(),is(closeTo(42.0,0.001)));
}

@Test
@DisplayName("Should not deposit negative amounts")
void shouldNotDepositNegativeAmounts() {
    SavingsAccount s = new SavingsAccount();
    boolean ret = s.deposit(-42.0);
    assertThat(ret,is(false));
    assertThat(s.getBalance(),is(closeTo(0.0,0.001)));
}
```

What about that test suite?

- Both the tests pass
- They get us to 100% statement (and branch!) coverage



Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
mutation	100,0 %	158	0	158
src/main/java	100,0 %	42	0	42
mutation	100,0 %	42	0	42
SavingsAccount.java	100,0 %	42	0	42
src/test/java	100,0 %	116	0	116

Is the test suite really **effective**, though?

Let’s try mutation testing with PIT

Configure Maven to use the PIT plugin

```
<plugin> <!- pom.xml file -->
    <groupId>org.pitest</groupId>
    <artifactId>pitest-maven</artifactId>
    <version>1.4.3</version>
    <!- dependencies, executions... -->
    <configuration>
      <targetClasses>
        <param>mutation.*</param><!--mutate all classes in the mutation package-->
      </targetClasses>
      <targetTests>
        <param>mutation.*</param><!--consider only tests in the same package-->
      </targetTests>
    </configuration>
</plugin>
```

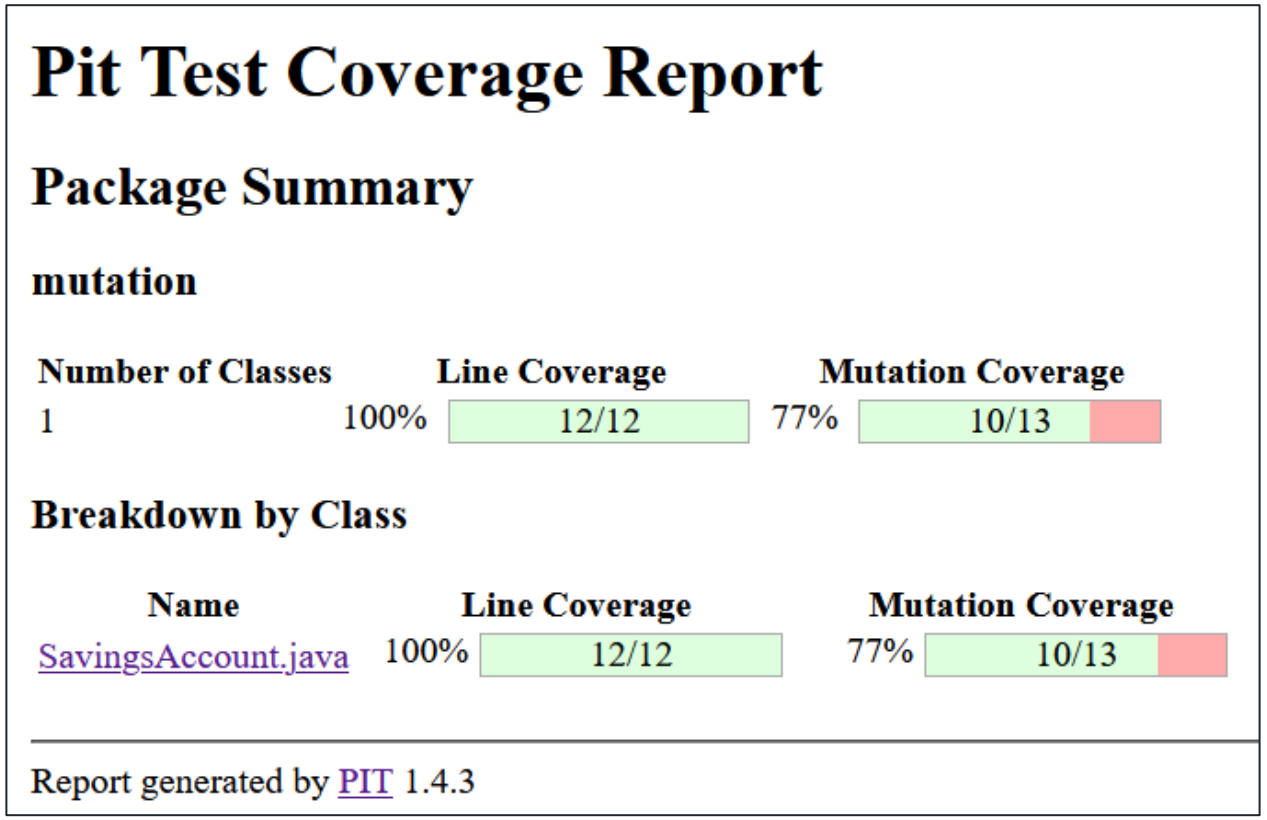
Let’s try mutation testing with PIT

PIT execution

```
================================================================================
- Timings
================================================================================
> scan classpath : < 1 second
> coverage and dependency analysis : < 1 second
> build mutation tests : < 1 second
> run mutation analysis : < 1 second
--------------------------------------------------------------------------------
> Total  : 1 seconds
================================================================================
- Statistics
================================================================================
>> Generated 13 mutations Killed 10 (77%)
>> Ran 21 tests (1.62 tests per mutation)
```

Let’s try mutation testing with PIT

- PIT produces also a **detailed**, **human-readable** HTML report
- The report allows us to see which mutants were generated and which ones escaped



Pit Test Coverage Report

Package Summary

mutation

Number of Classes	Line Coverage		Mutation Coverage	
1	100%	12/12	77%	10/13

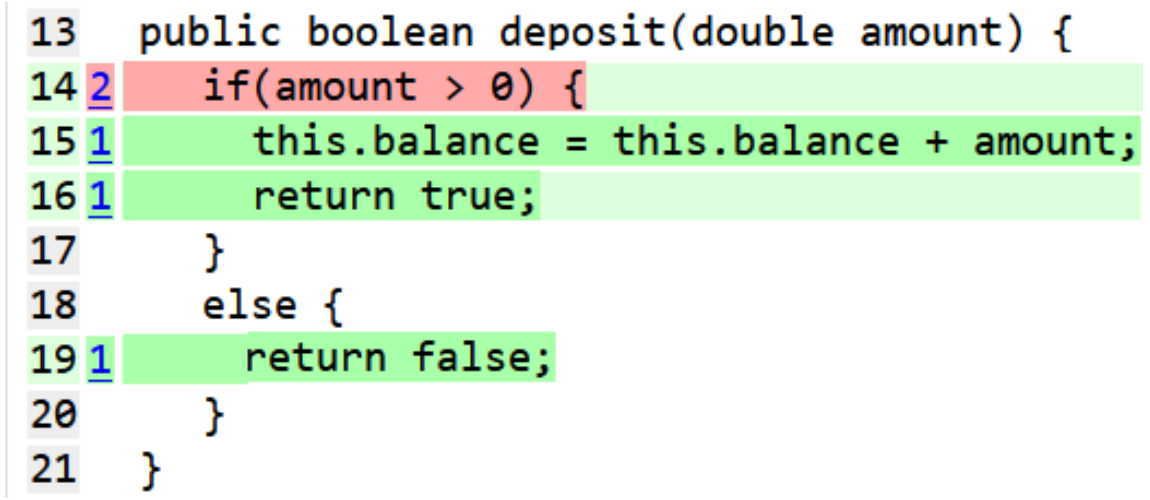
Breakdown by Class

Name	Line Coverage		Mutation Coverage	
SavingsAccount.java	100%	12/12	77%	10/13

Report generated by PIT 1.4.3

Reading PIT reports

- The blue numbers indicate how many mutants were generated by modifying the corresponding line (e.g.: 2 mutants for line 14)
- Red lines are those for which at least one of the corresponding mutants escaped
- Every mutant for the green lines was killed



Reading PIT reports

- PIT also tells us which mutation operators were applied on each line

```
1. deposit : changed conditional boundary → SURVIVED    (i.e.: changed > with >= )
2. deposit : negated conditional → KILLED               (i.e.: changed > with <= )
```

```
13  public boolean deposit(double amount) {
14 2    if(amount > 0) {
15 1      this.balance = this.balance + amount;
16 1      return true;
17      }
18      else {
19 1      return false;
20      }
21  }
```

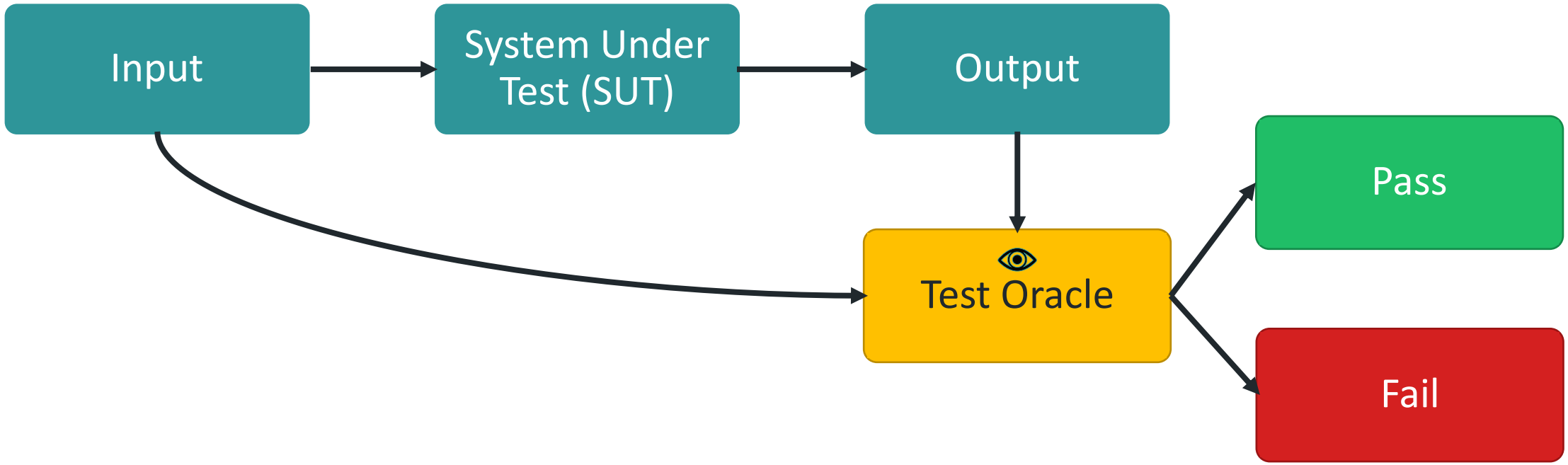
The mutant escaped because we did not properly test boundary values. **What happens when we try to withdraw zero money?**

Metamorphic Testing

Solving the oracle problem, one metamorphosis at a time

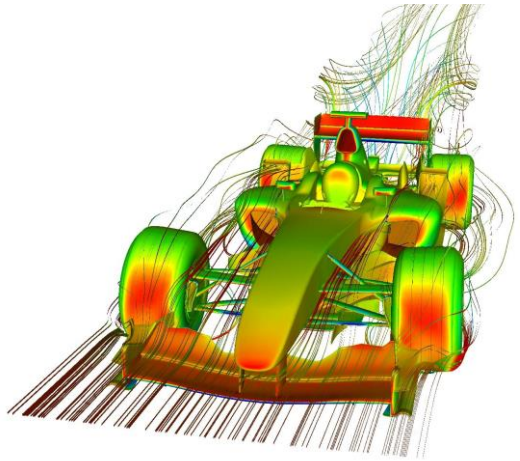
Test oracles

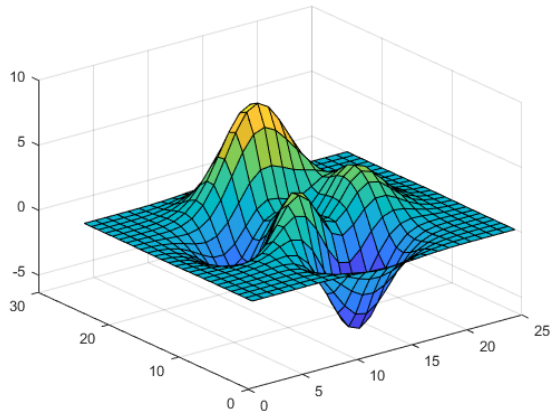
- A **test oracle** is a mechanism to decide whether a test output is correct

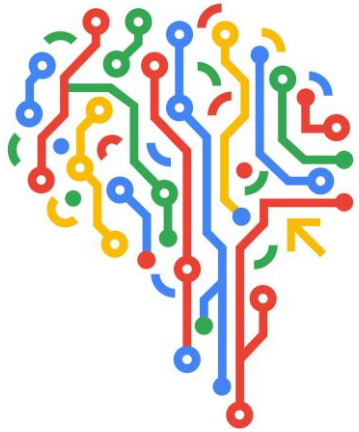


The oracle problem

- Sometimes it’s not feasible to get an automated test oracle
- This is an issue especially when trying to **automatically generate** tests

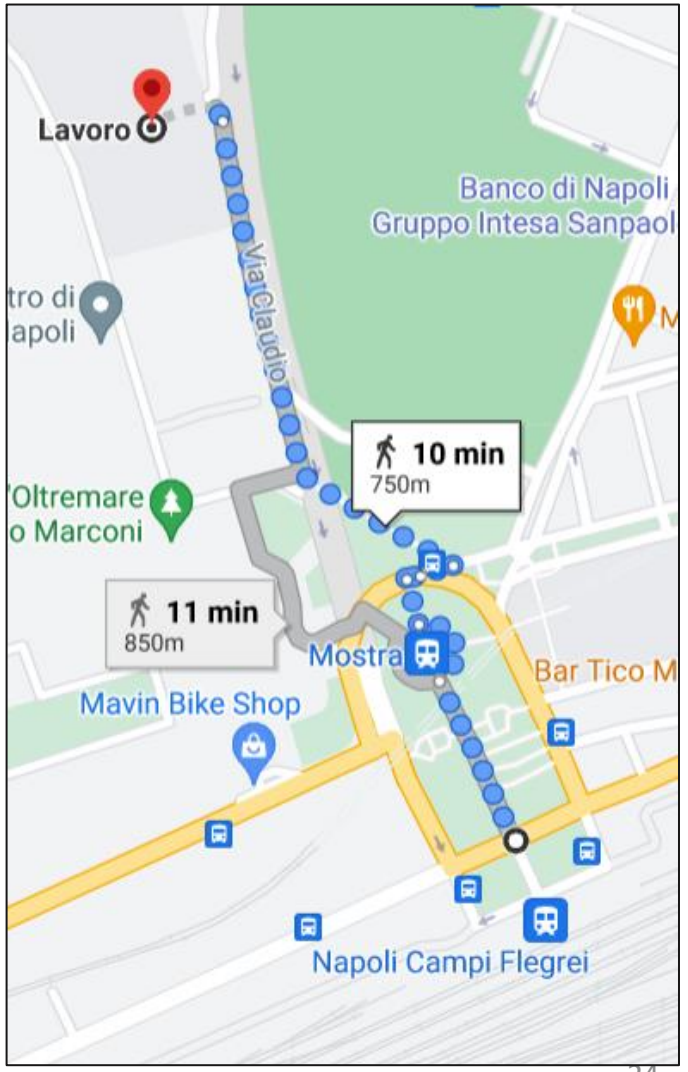






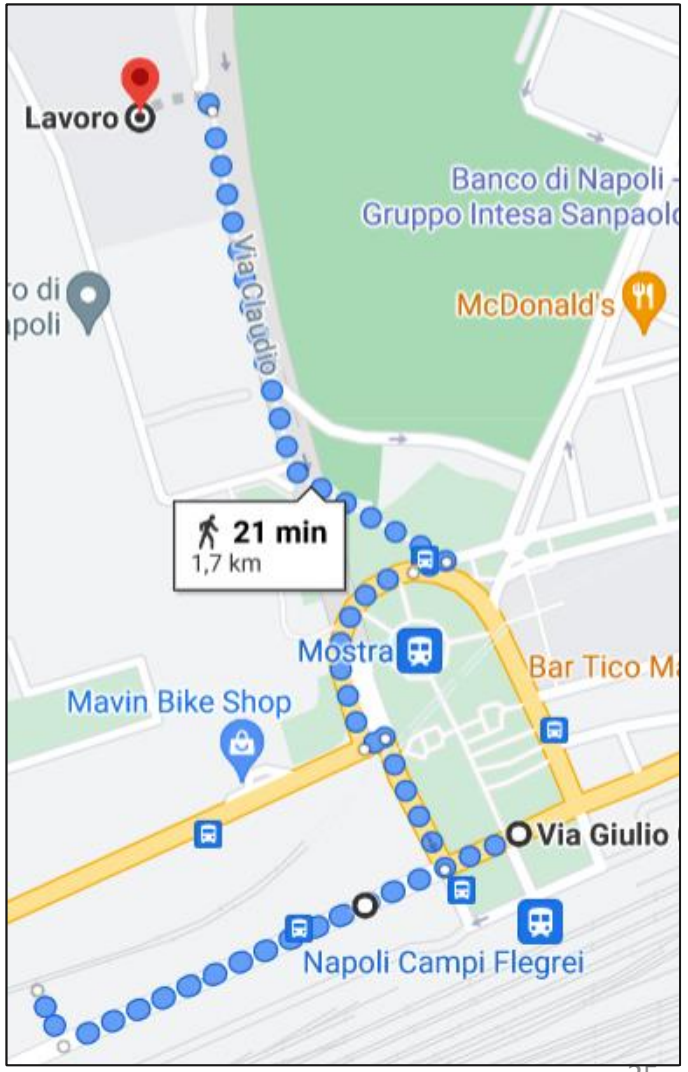
Metamorphic Testing by example (Maps)

- Suppose you're an automated testing tool
- Your task is to test the Google Maps app
- You start with a first query…
- Is the result correct?
- Hard to tell for an automatic tool!



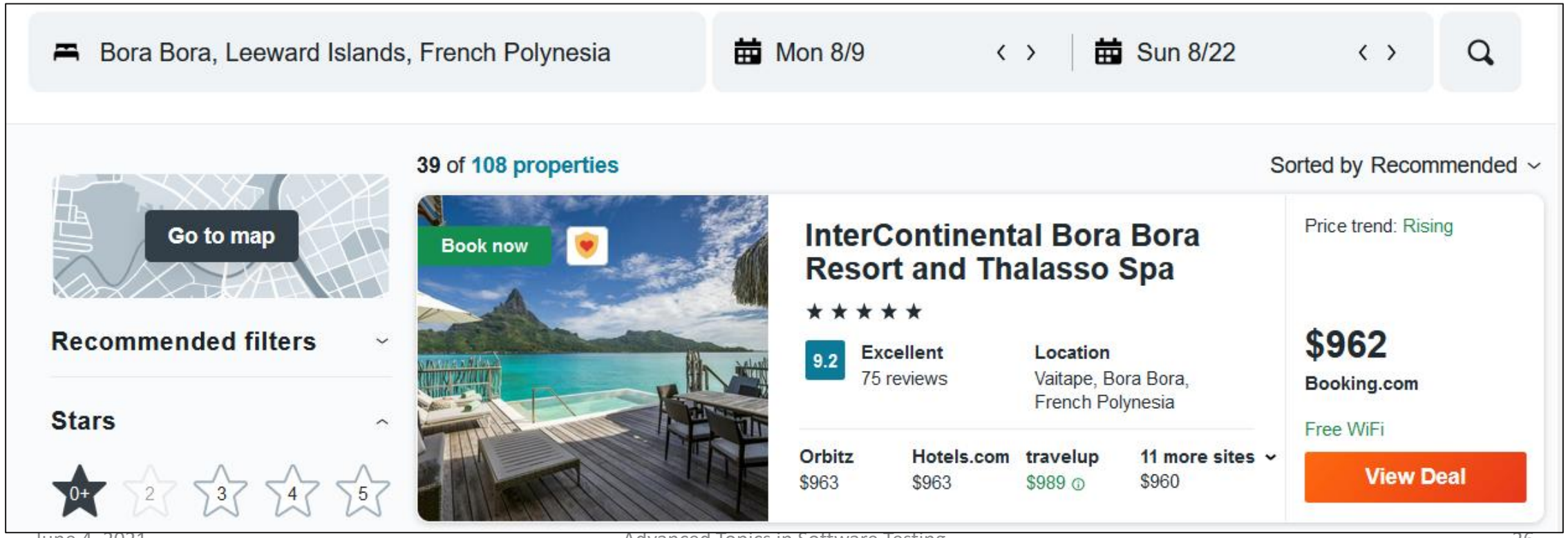
Metamorphic Testing by example (Maps)

- You can try a slightly different query
- The starting point is moved 10 meters west this time
- It is reasonable to expect the proposed path to be roughly the same
- Otherwise, it is likely that there's a bug in the system!



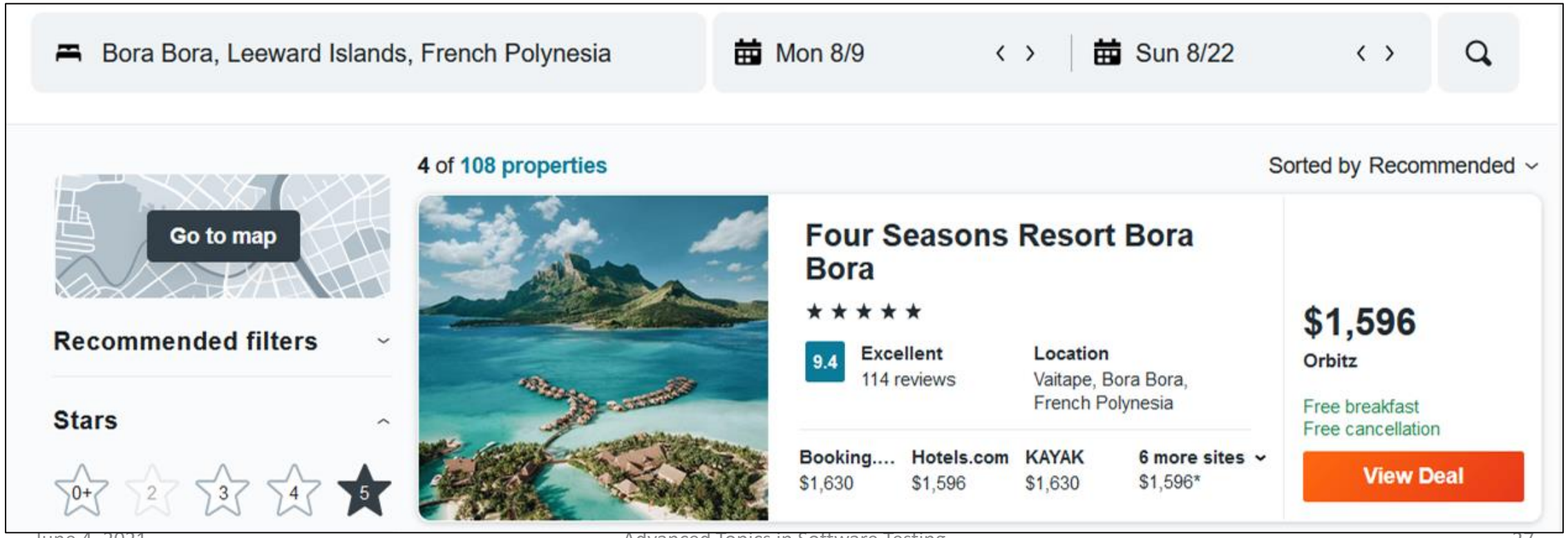
Metamorphic Testing by example (Search)

- Suppose you're testing a search engine-like application
- You get 39 results matching your initial query



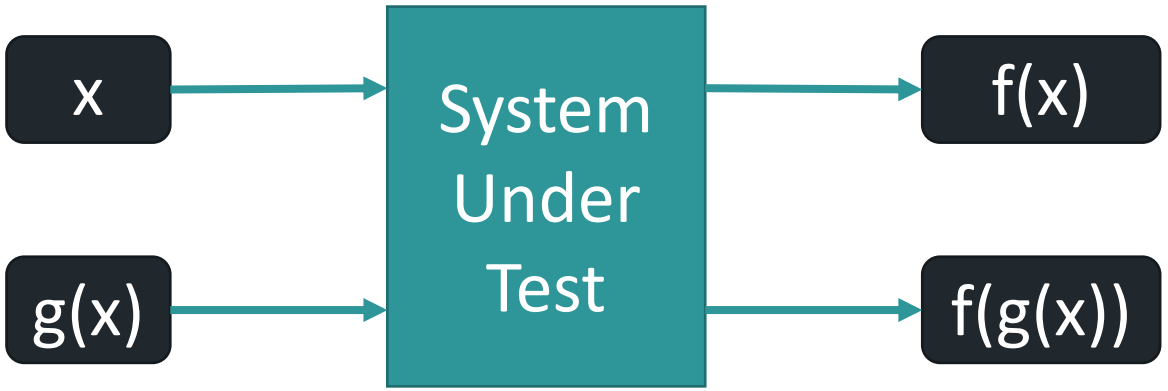
Metamorphic Testing by example (Search)

- If you restrict your search (e.g.: consider only 5 star structures)
- You expect to get a subset of the initial set of hotels!



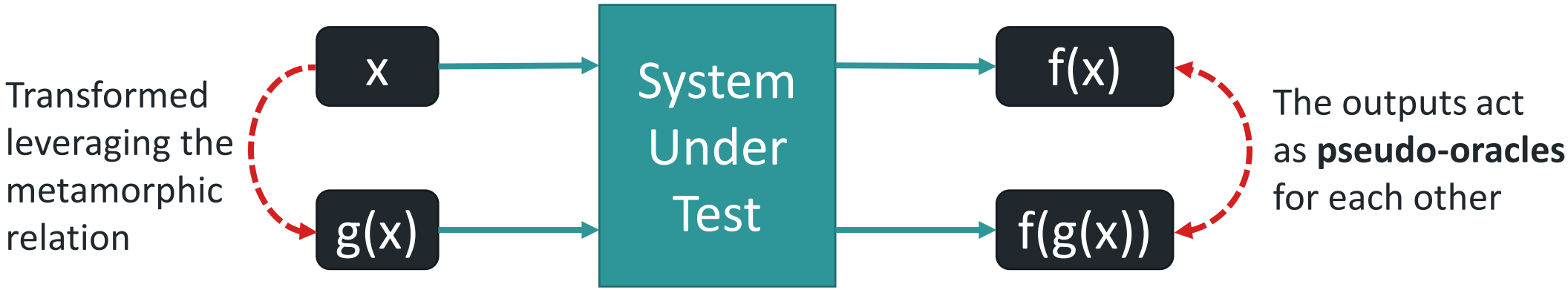
The intuition behind Metamorphic Testing

- Sometimes, we do not know whether the output of any individual input is correct
- **But**, we may know the relation between some inputs and their outputs!



Metamorphic Testing

- Identify some meaningful properties of the problem (the so-called **metamorphic relations**)
- Define a ***source*** (initial) test case
- Generate ***follow-up*** test cases by using the metamorphic relations to transform the *source* input and validate the output.

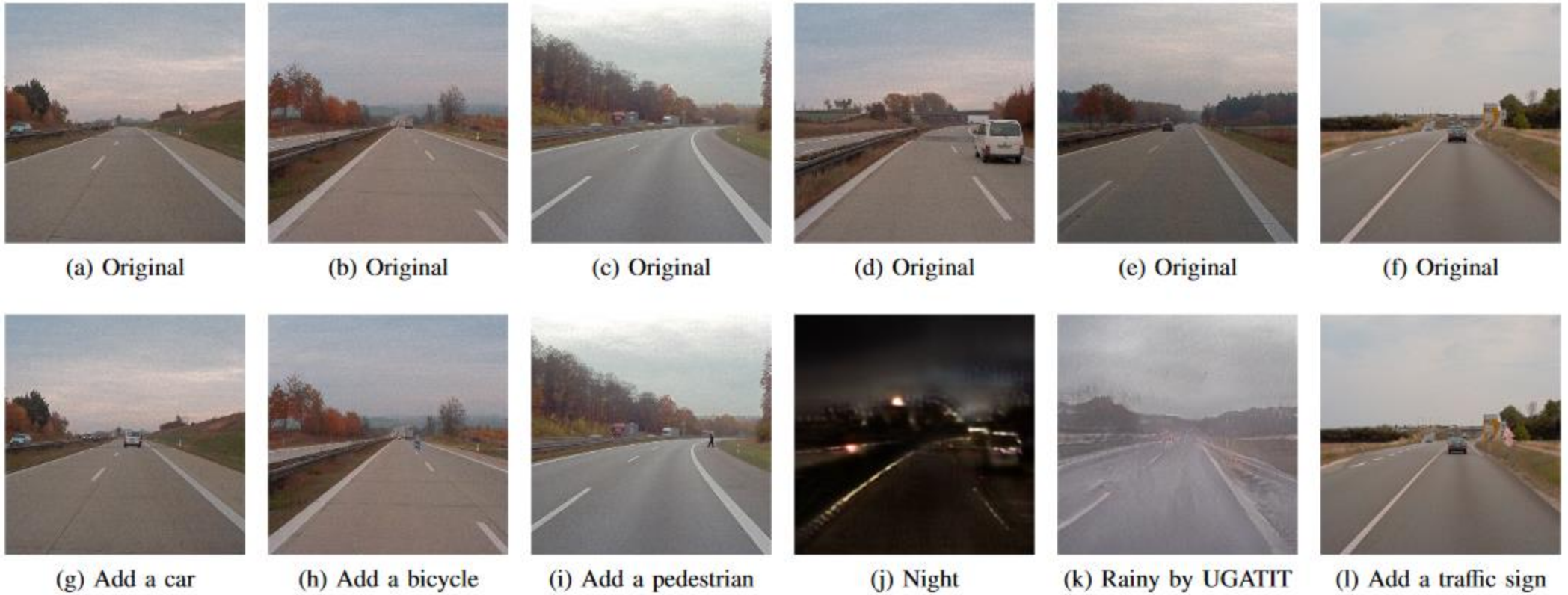


Successful applications

Metamorphic testing has been applied with promising results in many different scenarios (in addition to the one we already talked about!):

- Compilers and Graphic Drivers
- Cybersecurity (e.g. code obfuscators)
- Bioinformatics and numerical programs
- Machine Learning applications
- Autonomous driving
- … and many more!

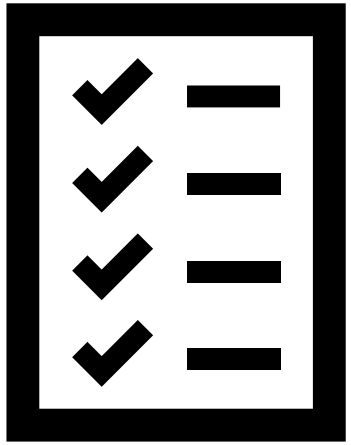
Metamorphic Testing and AI



From Deng, Y., Zheng, X., Zhang, T., Lou, G., & Kim, M. (2020). RMT: Rule-based Metamorphic Testing for Autonomous Driving Models. arXiv.

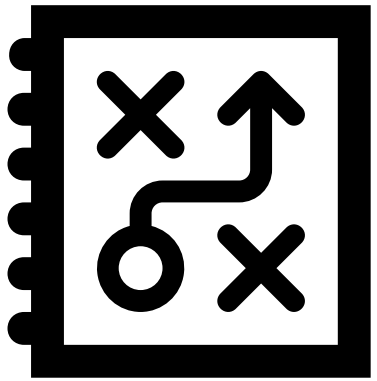
Challenges & Open Issues

- No systematic guidelines to design metamorphic relations (yet!)



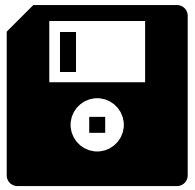
Challenges & Open Issues

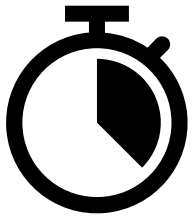
- No systematic guidelines to design metamorphic relations (yet!)
- Automating the generation of *likely* metamorphic relations



Challenges & Open Issues

- No systematic guidelines to design metamorphic relations (yet!)
- Automating the generation of *likely* metamorphic relations
- Investigating application to non-functional testing





Challenges & Open Issues

- No systematic guidelines to design metamorphic relations (yet!)
- Automating the generation of *likely* metamorphic relations
- Investigating application to non-functional testing
- No mainstream tools to support the use of the technique (yet!)

References

References (Mutation Testing)

1. Jia, Y., & Harman, M. (2010). An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, *37*(5), 649-678.
2. Coles, H., Laurent, T., Henard, C., Papadakis, M., & Ventresque, A. (2016, July). Pit: a practical mutation testing tool for java. In *Proceedings of the 25th international symposium on software testing and analysis* (pp. 449-452).
3. Petrovic, G., Ivankovic, M., Kurtz, B., Ammann, P., & Just, R. (2018, April). An industrial application of mutation testing: Lessons, challenges, and research directions. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (pp. 47-53). IEEE.
4. Usaola, M. P., & Mateo, P. R. (2010). Mutation testing cost reduction techniques: A survey. IEEE software, 27(3), 80-86.

References (Metamorphic Testing)

1. Chen, T. Y., Kuo, F. C., Liu, H., Poon, P. L., Towey, D., Tse, T. H., & Zhou, Z. Q. (2018). Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)*, *51*(1), 1-27.
2. Brown, J., Zhou, Z. Q., & Chow, Y. W. (2019). Metamorphic Testing of Mapping Software. In *Towards Integrated Web, Mobile, and IoT Technology* (pp. 1-20). Springer, Cham.
3. Chan, W. K., Cheung, S. C., & Leung, K. R. (2007). A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Services Research (IJWSR)*, *4*(2), 61-81.
4. Segura, S., Towey, D., Zhou, Z. Q., & Chen, T. Y. (2018). Metamorphic testing: Testing the untestable. *IEEE Software*, *37*(3), 46-53.
5. Zhang, M., Zhang, Y., Zhang, L., Liu, C., & Khurshid, S. (2018, September). DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 132-142). IEEE.