Software Testing Part 2

CS350 Introduction to Software Engineering

Shin Yoo

Software Testing Techniques & Approaches

- There are many variable factors when you are “testing” a software system… to the point that you go to an academic conference on software testing and really do not understand a particular presentation :)
    - Domain (embedded, web, app, enterprise middleware, education…), foundational background (symbolic execution, static analysis, search-based, random, manual…), abstraction level (unit testing, system testing, GUI testing…),…
- A justifiable holistic taxonomy is probably too much to attempt here; we will look at some of the major ideas in no particular order.

Random Testing / Fuzzing

- Randomness can be a good thing when applied properly to software testing: it will induce program behavior that you did not expect!
  - Similarly, we highly value diversity in testing: both low-level diversity (as in diverse test input) and high-level diversity (as in diversity in your developers, testers, and users).
- More formally, random search/optimisation is a good strategy when there is no gradient, and the target function is not continuous / differentiable.
  - Our ultimate objective function when finding a test input is to find a test input that will reveal a fault: this objective is clearly not differentiable.

Randoop

Automatic unit test generation for Java

https://randoop.github.io/randoop/

https://netflix.github.io/chaosmonkey/

```
@Test
public void test168() throws Throwable {
  if (debug)
    System.out.format("%n%s%n", "RegressionTest4.test168");
  My my1 = new My((int) 'a');
  java.lang.Class<?> wildcardClass2 = my1.getClass();
  java.lang.Class<?> wildcardClass3 = my1.getClass();
  java.lang.Class<?> wildcardClass4 = my1.getClass();
  my1.testMe((int) (short) 10, (int) (short) 0);
  java.lang.Class<?> wildcardClass8 = my1.getClass();
  java.lang.Class<?> wildcardClass9 = my1.getClass();
  my1.testMe((int) (byte) 1, (int) (byte) 0);
  my1.testMe((int) (short) 100, 0);
  my1.testMe((int) (short) 10, (int) (short) 1);
  java.lang.Class<?> wildcardClass19 = my1.getClass();
  java.lang.Class<?> wildcardClass20 = my1.getClass();
  java.lang.Class<?> wildcardClass21 = my1.getClass();
  org.junit.Assert.assertNotNull(wildcardClass2);
  org.junit.Assert.assertNotNull(wildcardClass3);
  org.junit.Assert.assertNotNull(wildcardClass4);
  org.junit.Assert.assertNotNull(wildcardClass8);
  org.junit.Assert.assertNotNull(wildcardClass9);
  org.junit.Assert.assertNotNull(wildcardClass19);
  org.junit.Assert.assertNotNull(wildcardClass20);
  org.junit.Assert.assertNotNull(wildcardClass21);
}
```

Gremlin — Product · Pricing · Resources · Company · Login · GET STARTED

Chaos Monkey > The Origin of Chaos Monkey

THE ORIGIN OF CHAOS MONKEY

Why Netflix Needed to Create Failure

4 MIN READ · Last Updated October 17, 2018

In this chapter we'll take a deep dive into the origins and history of Chaos Monkey, how Netflix streaming services emerged, and why

DOWNLOAD PDF →

https://www.gremlin.com/chaos-monkey/the-origin-of-chaos-monkey/

Fuzzing

- Fuzzing is a testing technique that involves providing unexpected random input to the target program.
  - Apparently “fuzz” originated from 1988 class project in U. of Wisconsin, where Prof. Barton Miller told students to generate random inputs and parameters to Unix utilities.
  - During 90s, fuzzing was shown to be effective at revealing security bugs (note: the objective is not differentiable).
  - Now, at least in industry, any type of automated non-deterministic testing technique tend to be called “fuzzing” 🫠

Strengths/Weaknesses

- To really benefit from the randomness, we need to sample a lot of inputs.
  - This is easy to do, as we are going to do it randomly 🤠
  - Random testing is intuitive, easy to implement, and effective when used right.
- However, if we are to execute a massive number of random inputs…
  - We need a massive number of oracles…? 🥺
  - Typically we do random testing against implicit oracles only (security issues are often detected by segfaults or other crashes!)

Structural Testing

- If we cannot formulate “fault detection” as an objective, we can at least try to execute all code at least once - since execution is the necessary condition for dynamic analysis to find faults (what is the sufficient condition?)
- Structural Testing aims to achieve various executions: statements, branches, particular condition in branch, etc…
  - We can quantify the progress of structural testing as “structural coverage”: out of X enumerable objectives, my testing actually executes Y: $\frac{Y}{X} \cdot 100\,\%$

Structural Testing & Coverage

- What does 100% coverage tell you?
  - Very little - as it is only necessary condition.
- What does the 60% coverage tell you?
  - The unmistakable fact that you are currently NOT testing 40% of your code.
- Okay, next day, I have improved by coverage up to 90%. How many brownie points do I get?
  - Not much - ideally, coverage itself should not be a goal. Have you simply executed the increase of 30%, or have you really tested it?

Coverage Criteria

- You can target different structural elements
  - Statements and branches: popular choices
  - Condition coverage: are all boolean subexpressions evaluated as both True and False?
  - Function: are all functions being called?
  - Path: not practical, as loops can produce an exponentially large number of paths
  - …

Automated Test Input Generation for Structural Testing

- The problem of achieving specific coverage can be solved automatically at least - there are two major approaches: concolic testing & search-based approaches.
- Oracle problems still remain: "Okay, input (x, y, z) does execute the False branch of this specific if statement.. but what should be the return value?"
  - Automated Test Input Generation typically target unit level testing - so humans should write assertions.
  - You can capture current behavior.

Concolic Testing

Concrete + Symbolic Execution Testing

- First, execute the program with random concrete inputs: `x = 0`, `y = 0`
- During execution, collect any condition observed symbolically: `z = 2 * y && x != 100000`
- Negate the last part, and solve the condition using an SMT solver: `z = 2 * y && x == 100000 –>` You `get x =100000, y = 0`
- Execute again, and collect conditions: `z = 2 * y && x == 100000 && x >= z`
- Negate and solve: `z = 2 * y && x == 100000 && 100000 < z –> x == 100000, y = 50001, z = 100002`

```
void f(int x, int y) {
    int z = 2*y;
    if (x == 100000) {
        if (x < z) {
            assert(0); /* error */
        }
    }
}
```

https://en.wikipedia.org/wiki/Concolic_testing

Search-Based Test Data Generation

Formulate the problem as optimisation

- Define a fitness (objective) function that measures the distance between current execution path and your target: then use an optimisation technique to minimize this.
- First, we need to penetrate `x == 100000`. This is equivalent to minimizing the function $f(x, y) = |x - 100000|$ to zero.
- Then we subsequently find out that increasing `y` helps making `x < z` <u>more true</u>, so the optimisation algorithm will increase `y` until `x < z`.

```
void f(int x, int y) {
    int z = 2*y;
    if (x == 100000) {
        if (x < z) {
            assert(0); /* error */
        }
    }
}
```

https://en.wikipedia.org/wiki/Concolic_testing

GUI Testing

- How do we ensure that various use cases are correctly implemented? How do we do this automatically?
- We need a mechanism to execute user interaction automatically.
  - Monkeys (again!)
  - Scripting (via structural handles in the UI frameworks)
  - Visual GUI Testing

Monkey Testing

- Supply a stream of random user events to the GUI, see if it crashes
- A form of stress testing with a long history
- https://developer.android.com/studio/test/other-testing-tools/monkey

GUI Scripting

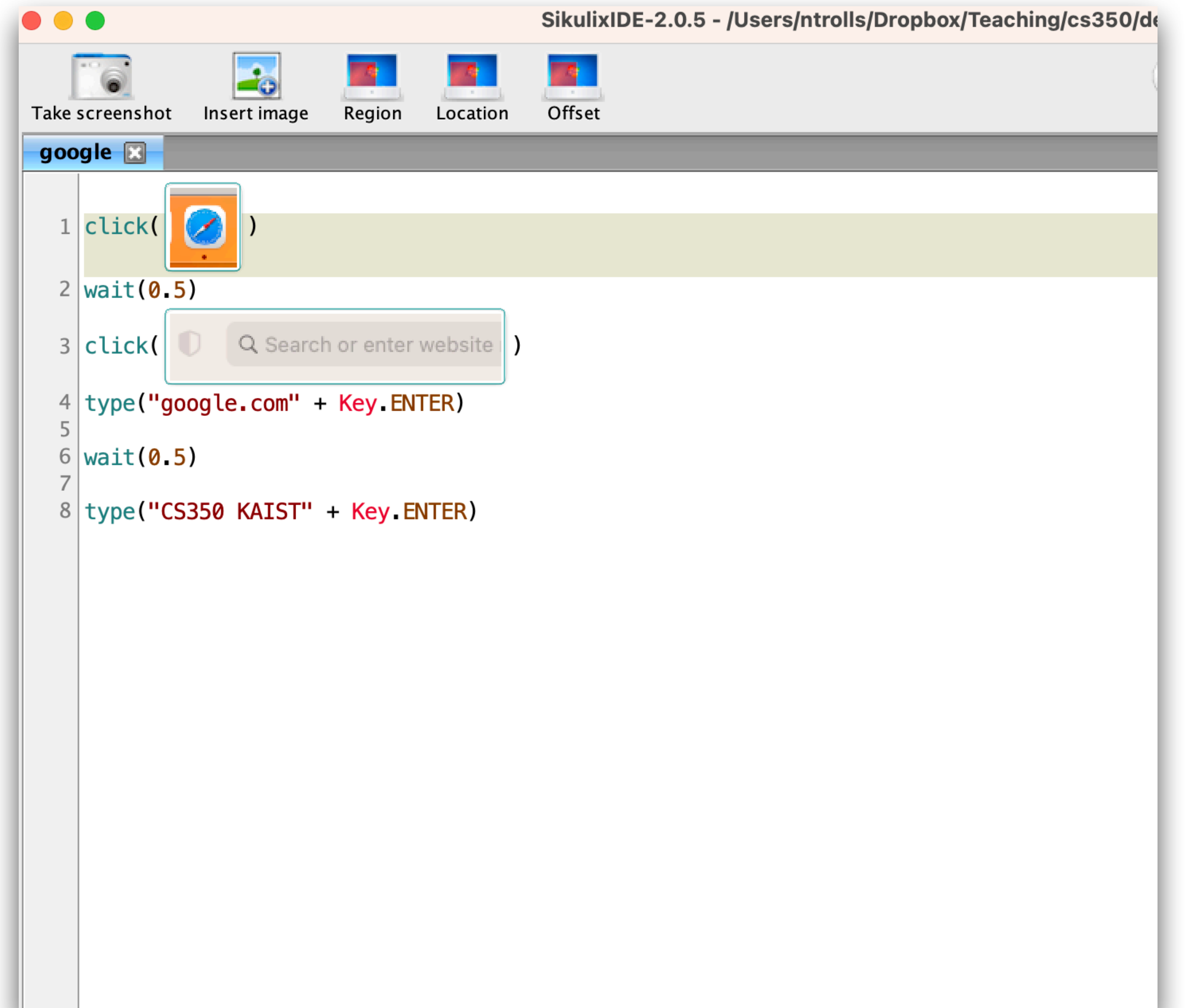
Example is based on Selenium (https://www.selenium.dev/)

- Identify GUI widgets by their properties (CSS selector, HTML element type, etc)
- Interact with widgets by creating events for them (text input, click, etc)
- You essentially do web browsing via a script

```
from selenium import webdriver
from selenium.webdriver.common.by import By


def test_eight_components():
    driver = webdriver.Chrome()

    driver.get("https://www.selenium.dev/selenium/web/web-form.html")

    title = driver.title
    assert title == "Web form"

    driver.implicitly_wait(0.5)

    text_box = driver.find_element(by=By.NAME, value="my-text")
    submit_button = driver.find_element(by=By.CSS_SELECTOR, value="button")

    text_box.send_keys("Selenium")
    submit_button.click()

    message = driver.find_element(by=By.ID, value="message")
    value = message.text
    assert value == "Received!"

    driver.quit()
```

Visual Scripting

- A scripting language that can take images as arguments + a script runtime that will use computer vision to interpret the image arguments = GUI scripting without internal structure knowledge
- SikuliX from MIT
- The idea was proven to be useful in industrial contexts (see for example: https://ieeexplore.ieee.org/abstract/document/8048660)



GUI Exploration

- What if you want to automatically “cover” the GUIs, i.e., I want to reach all views in this Android app?
- The GUI as a whole can be represented as a kind of state machine: each activity is a state, each event triggers its own transition to another state (i.e., activity).
- Many automated techniques have been suggested to explore this state space automatically. Once you have the state machine representation, it can help you automatically derive further testing scenarios and scripts.

Regression Testing

- A regression fault refers to a feature that used to be okay, but is now indirectly broken due to recent addition/modification of other features.
- To detect regressions, you need to execute all previous test cases (that correspond to all existing features); however, there may be too many of them.
- How do you optimize this process without losing fault detection capabilities?
  - Select test cases that are guaranteed to execute any changed parts
  - Prioritize execution order based on achieved coverage / diversity

Mutation Testing

- “All my test passes!” - does it mean that…
  - my program is perfect and bug free! 🤠, or
  - my test suite is very bad 🫠
- But if we do not know which bugs are out there, how do we know whether our tests are any good?
- A brilliant answer to this is came out back in 1978: we can create our own bugs!

Richard A. DeMillo, Richard J. Lipton, and Fred G. Sayward. Hints on test data selection: Help for the practicing programmer. IEEE Computer, 11(4):34-41. April 1978

Mutation Testing

- Simply create syntactic variants of the program: swapping + to -, changing the function argument order, changing logical operator && to II, etc… (these are called mutation operators)
- Two underlying assumptions
  - Competent Programmer Hypothesis: most programmers are good, and the mistakes they make are relatively small
  - Coupling Effect Hypothesis: the small syntactic mistakes we inject are semantically linked to more complex real faults

Mutation Testing

- Create N mutants
- Execute your test suite against each of the N mutants
  - If a test case fails, you killed the mutant (i.e., your test can discern the difference)
  - If a test case passes, you did not kill it
- Report the percentage of the killed mutants (which is called mutation score)

Mutation Testing

- For a long time, people thought mutation testing is a wonderful thing except for a couple of real drawbacks
  - Cost is really high: tens of thousands of mutants to build and test against
  - Equivalent mutants: some syntactic variants are still semantically equivalent, but we cannot filter them out (semantic equivalence is undecidable)
- Yet practical adoption is slowly happening!
  - CI/CD automatically applies only a small number of mutations and reports the results as part of automated code review: if any mutation to the incoming change is not killed by tests, it becomes an warning.

G. Petrović, M. Ivanković, G. Fraser and R. Just, "Practical Mutation Testing at Scale: A view from Google," in *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3900-3912, 1 Oct. 2022, doi: 10.1109/TSE.2021.3107634.

Testing ML-based Systems

One of the hottest testing topics right now

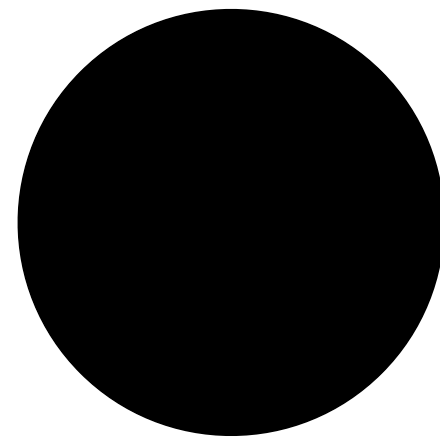
- Most of testing techniques assume that the system under test is deterministic, and we can decide whether the test outcome is correct or incorrect.
- Suddenly we have a surge of ML-based systems that use some machine learning model as (part of) the core business logic.
  - Their outputs are often not binary or even discrete.
  - We cannot realistically expect 100% correctness.
- How do we ensure their functional behavior?

Testing ML-based Systems

Fearfully large size of input space

Number of
stars in the universe

$\simeq 2^{80}$



Number of
inputs for a program
that can be the coursework
for Programming 101
(three 32bit integers)

$= 2^{84}$

Number of
possible 28 by 28 B&W images,
(the size of an MNIST input)

$= 2^{3136}$

Testing ML-based Systems

Ambiguities in inputs

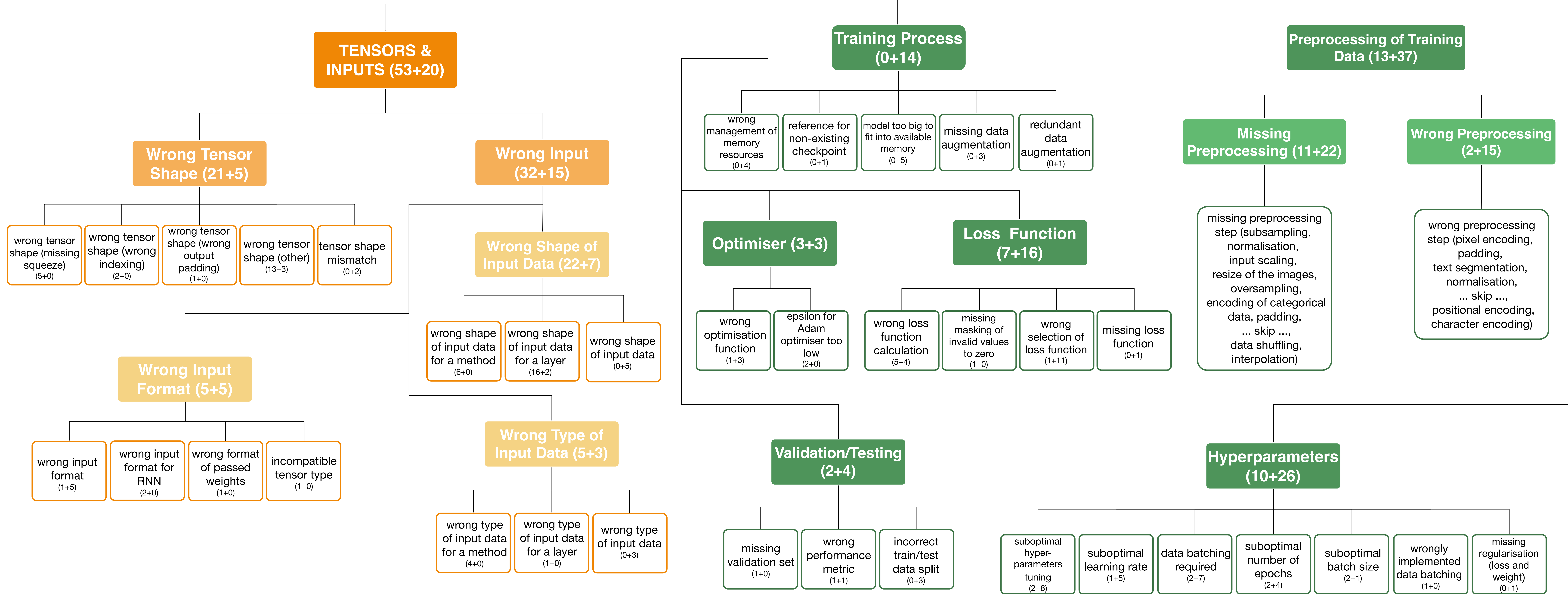
- In case of DNNs, we often rely on them because they are so good at perceptive tasks (vision, speech recognition, …)
  - These inputs cannot be easily abstracted into equivalence classes: too much degree of freedom in the input space.
  - Certain inputs are genuinely difficult to classify for humans, i.e., we do not know what the oracles should be.
  - Oracle here is human labelling, which is not only expensive but sometimes even ethically questionable.

Testing ML-based Systems

State of the Art

- Most research efforts are focused on the concept of robustness, i.e., the ML-system should be resilient to minor perturbations in inputs.
  - For example, if an autonomous driving system correctly turns right at a specific crossing to reach a destination, it should do the same on a rainy day (rain = perturbation to normal circumstances).
  - ML-systems should be also resilient to adversarial examples, i.e., inputs with injected noise with the intention of tricking the ML model.
  - Robustness testing can be reasonably automated, but is still fundamentally limited in its completeness (i.e., we cannot anticipate all possible perturbations)

Taxonomy of Real Faults in Deep Learning Systems, N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco,
The 42nd International Conference on Software Engineering, 2020



Taxonomy of Real Faults in Deep Learning Systems, N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco,
The 42nd International Conference on Software Engineering, 2020

Who writes the tests?

- Two traditional views on who should test your system
  - Dedicated testers: resulting in a dedicated role of QA testers, whose job is simply to test implemented system
    - Testing requires creativity, domain expertise, broad knowledge of regression history… so dedicated QA engineers are necessary!
  - Developer themselves: developers are typically required to write unit tests at least, but some organizations require them to write all tests
    - If tests break, then who else is going to fix them? If developers can bring the mindset of dedicated QA engineers into programming, perhaps the best! DevOps and other automations also support this trend.

Summary

- Testing is a dynamic analysis: it is sound (i.e., if testing finds a problem, it is real) but not complete (i.e., testing will not find all problems).
- Typically we define an alternative, practical and achievable adequacy criterion and try to satisfy it - ideally using some automation in the process.
- Test oracle problem is fundamentally difficult, and will not easily go away - but we are making progress wherever possible.
- Various testing techniques tackle different aspects of quality criteria, domains, and lifecycle stages.