Lecture 0 – Introduction

AAA705: Software Testing and Quality Assurance

Jihyeok Park

PLRG

2024 Spring

Course Information

- **Instructor:** Jihyeok Park (박지혁)
  - **Position:** Assistant Professor in CS, Korea University
  - **Expertise:** Programming Languages, Software Analysis
  - **Office hours:** 14:00–16:00, Tuesdays (appointment by e-mail)
  - **Office:** 609A, Science Library Bldg
  - **Email:** jihyeok_park@korea.ac.kr

- **Class:** AAA705: Software Testing and Quality Assurance

- **Lectures:** 15:00–17:45, Mon. and Wed. @ 107 미래융합기술관

- **Homepage:** https://plrg.korea.ac.kr/courses/aaa705/

Schedule

Weak	Date	Contents
1	03/04	Introduction
	03/06	Combinatorial Testing
2	03/11	Random Testing
	03/13	Coverage Criteria (1)
3	03/18	Coverage Criteria (2)
	03/20	Search Based Software Testing (SBST)
4	03/25	Dynamic Symbolic Execution (DSE)
	03/27	Mutation Testing
5	04/01	Regression Testing
	04/03	Fault Localization
6	04/08	Metamorphic Testing
7	04/15	Differential Testing
	04/17	Course Review
12	05/20	Project Presentation
	05/22	Project Presentation

Grading

- **Homework Assignments: 90%**
  - **2 Programming Assignments**:
    - Homework 1: 20% (due on March 27)
    - Homework 2: 20% (due on April 17)
  - Submit your homework on **Blackboard**.
- **Project: 50%** (due on May 20)
  - Personal project. No team project.
  - Presentation on May 20 (Mon.) and May 22 (Wed.) 15:00 – 17:45
- **Attendance: 10%**

Course Materials

- **Self-contained lecture notes.**

  https://plrg.korea.ac.kr/courses/aaa705/

  (Special thanks to Prof. **Shin Yoo** @ KAIST)

- **Reference**: we do not teach these books and these books do not contain answers to this course.
  - **"Introduction to Software Testing (2nd Ed.)"** by Paul Ammann and Jeff Offutt.
  - **"Why Programs Fail (2nd Ed.)"** by Andreas Zeller.

Contents

Errors in Saftety-Critical Software

Unexpected faults in **safety-critical software** cause serious problems:


June 4, 1996: Ariane-5 explodes after lift off

Today in History: June 4, 1996: Ariane-5 explodes after lift off

Published: June 03, 2016 16:41
Abdul Kareem, Head of Archive
Follow us +


Knight Capital Says Trading Glitch Cost It

BY NATHANIEL POPPER AUGUST 2, 2012 9:07 AM 356

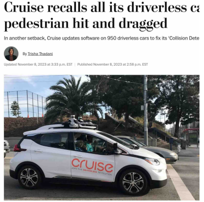
Runaway Trades Spread Turmoil Across Wall St.


Heathrow Airport apologises for IT failure disruption

16 February 2020

PA MEDIA

Heathrow is the busiest airport in Europe


Cruise recalls all its driverless ca pedestrian hit and dragged

In another setback, Cruise updates software on 950 driverless cars to fix its 'Collision Dete

By Trisha Thadani

Updated November 8, 2023 at 3:33 p.m. EST | Published November 8, 2023 at 2:58 p.m. EST

Rocket	**Financial**	**Airport**	**Auto. Vehicle**
(1996)	(2012)	(2020)	(2023)

Then, how can we **prevent** such software faults?

Can we **automatically check** whether a program does not have any software faults?

Detecting Software Faults

How do we know whether a software is correct?

Empiricists – Francis Bacon

vs.

Rationalists – René Descartes

It is correct because I **TESTED** *several times but no error was found!*

It is correct because I formally **PROVED** *that no error exists!*

Detecting Software Faults

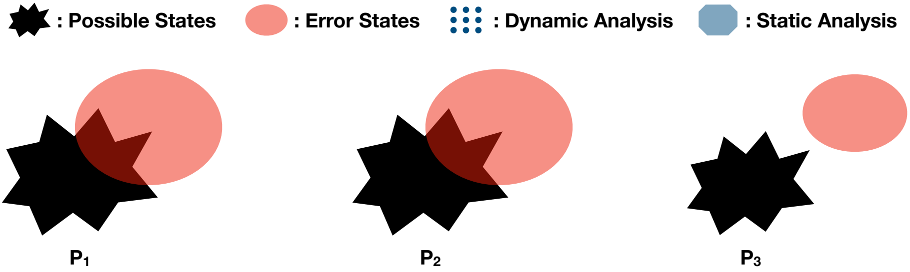
We can use various **analysis** techniques to detect software faults.

An **analyzer** is a program that takes a **program** and a **property** as inputs and determines whether the program **satisfies** the property.

We can categorize them into two groups:

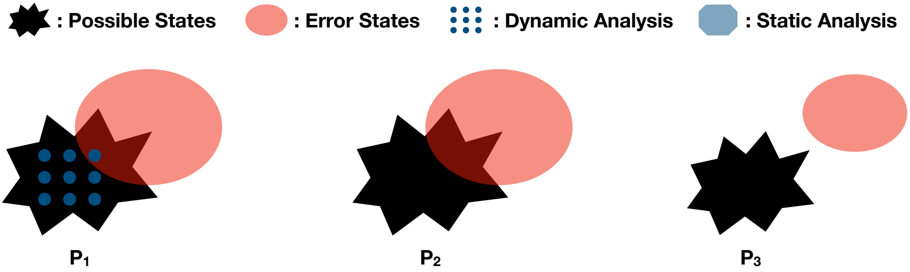
- **Dynamic analyzers** analyze programs by **executing** them.
- **Static analyzers** analyze programs **without executing** them.

Dynamic Analysis vs. Static Analysis



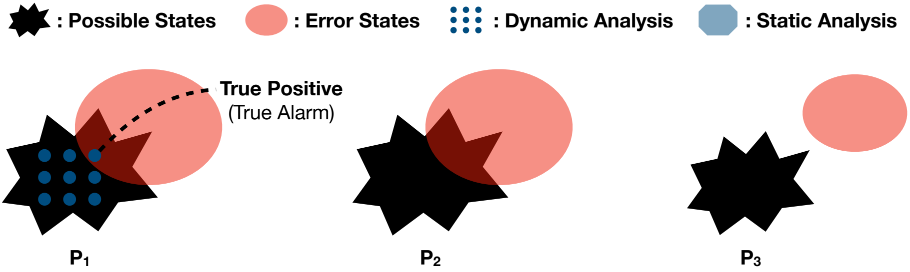
Dynamic Analysis	**Static Analysis**
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

Dynamic Analysis vs. Static Analysis



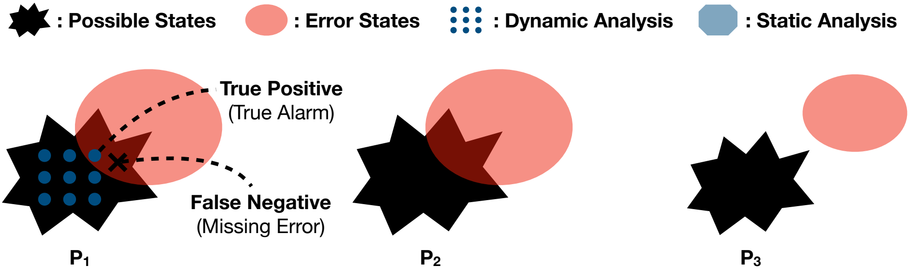
Dynamic Analysis	**Static Analysis**
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

Dynamic Analysis vs. Static Analysis



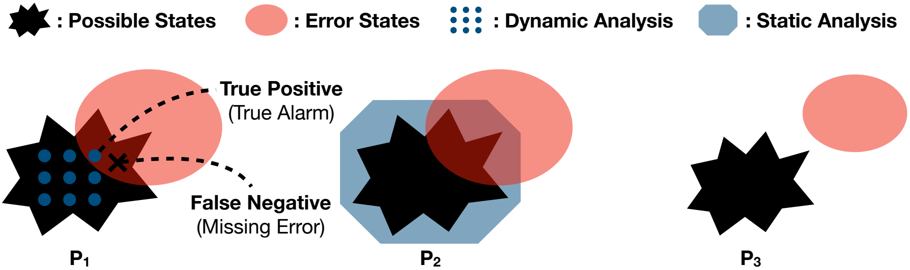
Dynamic Analysis	Static Analysis
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

Dynamic Analysis vs. Static Analysis



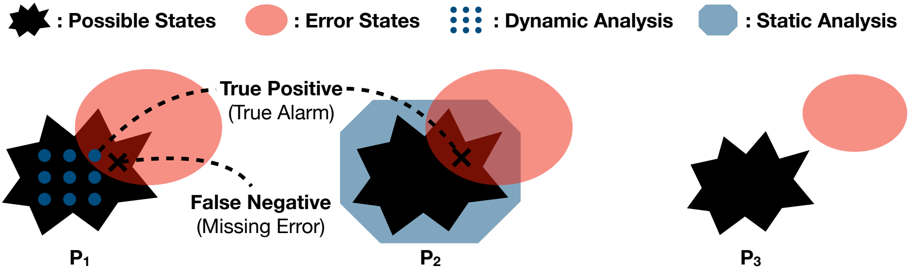
Dynamic Analysis	Static Analysis
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

Dynamic Analysis vs. Static Analysis



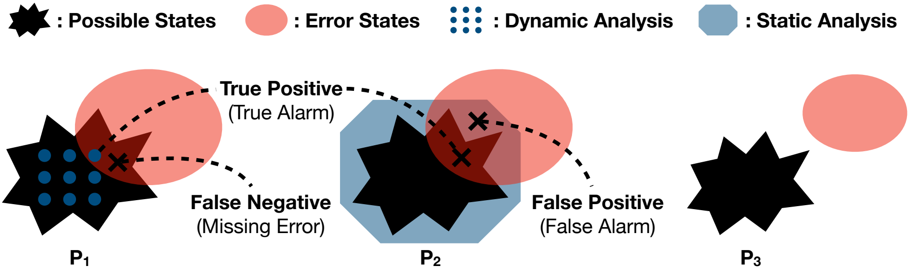
Dynamic Analysis	Static Analysis
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

Dynamic Analysis vs. Static Analysis



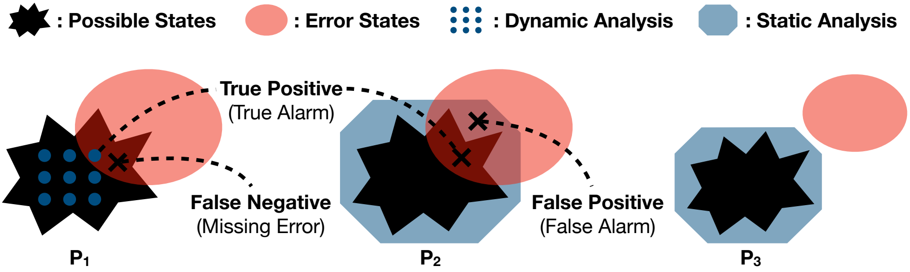
Dynamic Analysis	Static Analysis
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

Dynamic Analysis vs. Static Analysis



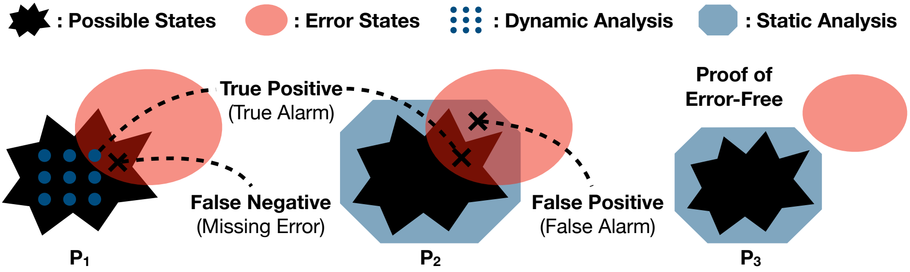
Dynamic Analysis	Static Analysis
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

Dynamic Analysis vs. Static Analysis



Dynamic Analysis	**Static Analysis**
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

Dynamic Analysis vs. Static Analysis



Dynamic Analysis	Static Analysis
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

Why Software Testing?

- Imagine you have two choices when boarding a airplane:
  - While an airplane A has **never been proven** to have any run-time errors, it has been **tested** with a finite number of test flights.
  - While an airplane B has been **formally verified** to have no run-time errors, it has **never been tested** in the real world.
- Some people may choose A, while others may choose B.
- In addition, some properties only can be **tested** but not **verified** (e.g., energy consumption, usability, etc.).

Contents

Software testing *is an* **investigation** *conducted to provide stakeholders with information about the* **quality** *of the product or service under test.*

Types of Software Quality: Dependability

The software should be **dependable**: **correct**, **reliable**, **safe**, and **robust**.

- **Correctness**: the software should exactly **conform** to its **formal specification**.
- **Reliability**: the software should have a **high probability** of being **correct** for period of time.
- **Safety**: the software should be **no risk** of any kind of **hazard** (loss of life, injury, etc.).
- **Robustness**: the software should reasonably **remain dependable** even if surrounding **environment changes**.

Types of Software Quality: Performance

Apart from dependability, the software should meet certain **performance** expectations.

- For example, execution time, network throughput, memory usage, number of simultaneous users, etc.
- **Hard to thoroughly test** due to the heavy reliance on the **execution environment** and **usage patterns**.

Types of Software Quality: Usability

The software should be **usable**.

- In general, there is no universally accepted criterion for **usability**.
- Usability testing usually involves **user studies**, such as **focus groups**, **beta-testing**, **A/B testing**, etc.

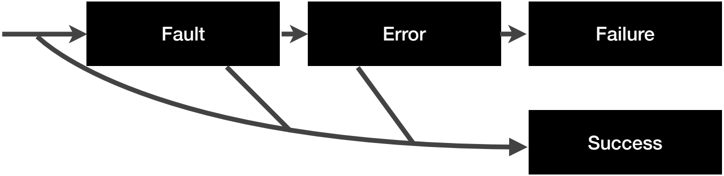
Types of Software Quality: Ethics

The software should be **ethical**.

- Typically, this is applied to AI/ML based systems.
- **[FSE'17]** S. Galhotra, Y. Brun, and A. Meliou. "*Fairness testing: testing software for discrimination.*"
- **[ASE'18]** S. Udeshi, P. Arora, and S. Chattopadhyay. "*Automated directed fairness testing.*"
- **[ICSE'20]** P. Zhang, J. Wang, J. Sun, G. Dong, X. Wang, X. Wang, J. S. Dong, and T. Dai. "*White-box fairness testing through adversarial sampling.*"

Faults vs. Errors vs. Failures

The purpose of testing is to **detect** and **remove** faults, errors, and failures.



From **IEEE Standard 729-1983**, IEEE Standard Glossary of Software Engineering Terminology[1]

- **Fault**: an anomaly in the software that may lead to an error.
- **Error**: a runtime effect of executing a fault, which may cause a failure.
- **Failure**: a manifestation of an error external to the software.

[1] https://ieeexplore.ieee.org/document/7435207

Faults vs. Errors vs. Failures

We want to implement a JavaScript function that computes the sum of elements in a given array.

```
function sum(arr) {
  let result = 0;
  for (let i = 0; i < arr.length; i++) {
    // fault: `i` should be fixed to `arr[i]`
    result += i;
  }
  return result;
}
```

It is a **fault** but **not an error** until the function is executed.

```
// the faulty statement is not reached at runtime (no error)
assert(sum([]) === 0);
```

Faults vs. Errors vs. Failures

We want to implement a JavaScript function that computes the sum of elements in a given array.

```
function sum(arr) {
  let result = 0;
  for (let i = 0; i < arr.length; i++) {
    // fault: `i` should be fixed to `arr[i]`
    result += i;
  }
  return result;
}
```

It is an **error** with the following input but **not a failure** because the output is **coincidentally correct**.

```
// the faulty statement is reachable at runtime (error)
// the output is coincidentally correct (no failure)
assert(sum([4, -2, 1]) === 3);
```

Faults vs. Errors vs. Failures

We want to implement a JavaScript function that computes the sum of elements in a given array.

```
function sum(arr) {
  let result = 0;
  for (let i = 0; i < arr.length; i++) {
    // fault: `i` should be fixed to `arr[i]`
    result += i;
  }
  return result;
}
```

It is a **failure** with the following input because the output is **incorrect**.

```
// the output is incorrect (failure)
assert(sum([3, 7, 4]) === 14);
```

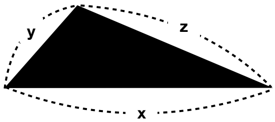
More Terminologies

- **Test Input**: a set of inputs that are used to test a program.
- **Test Oracle**: a mechanism to determine whether the program behaves correctly.
- **Test Case**: a pair of a test input and a test oracle.
- **Test Suite**: a set of test cases.
- **Test Effectiveness**: the ability of a test suite to detect faults or achieve other testing objectives.
- **Testing** vs. **Debugging**: testing is the process of detecting faults, while debugging is the process of fixing faults.

Contents

Problem – Sampling the Input Space

- **Exhaustive Testing**: Can we test a program with all possible inputs? In theory, **Yes!**
- However, it is infeasible for most programs.
- For example, consider a program that takes three 32-bit integers as inputs and returns they can form a **triangle** and **its type**.



  - How many possible inputs are there?

$$2^{32} \times 2^{32} \times 2^{32} = 2^{96} \approx 7.9 \times 10^{28}$$

  - Approximated number of stars in the universe: 10^{24}
- Testing allows only a **sampling** of an enormous **input space**. The difficulty lies in how to come up with **effective sampling**.

Problem – Test Oracle

- For every test input, we need to know the **expected behavior** of the program. (i.e., the **oracle**).

- How to define the **oracle**?

- Without an explicit oracle, we can only small subset of faults. (e.g., **crash**, **unintended infinite loop**, **division by zero**, etc.)

- We need to **define** or **infer** the oracle for testing.

Software Testing Techniques

- There is no fixed recipe for software testing.
- We need to understand the pros and cons of each testing technique.
- There are two major categories of testing techniques:
  - **Black-box Testing**: testing **without** knowing the internal structure of the program.
  - **White-box Testing**: testing **with** the knowledge of the internal structure of the program.

Black-box Testing

- **Combinatorial Testing**
  - Tester utilizes **input specifications** to generate test cases.

- **Random Testing**
  - Tester **randomly** selects test cases from the input space.

White-box Testing

Sometimes called **structural testing** because it uses the **internal structure** of the program to derive test cases.

- **Coverage Criteria**
  - The adequacy of a test suite is measured in terms of the **coverage** of the program's internal structure.
- **Search Based Software Testing (SBST)**
  - A technique that uses **meta-heuristic search** algorithms to maximize/minimize a certain **fitness function**.
- **Dynamic Symbolic Execution (DSE)**
  - A technique that systematically explores the input space using **symbolic execution** with **dynamic analysis**.

General Techniques

- **Mutation Testing**
  - A technique that evaluates the quality of a test suite by introducing **artificial faults** to the program.
- **Regression Testing**
  - A technique that ensures that a change in the program does not introduce new faults.
- **Fault Localization**
  - A technique that identifies the **location** of a fault in the program.
- **Metamorphic Testing**
  - A technique that tests a program using **metamorphic relations**.
- **Differential Testing**
  - A technique that tests a program by comparing the outputs of **multiple implementations**.

Next Lecture

- Combinatorial Testing

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr