Intelligent Software Engineering

Software Testing

Zhilei Ren

Dalian University of Technology

September 26, 2025

Outline

Outline

1947: The First "Computer Bug"

- **September 9, 1947**: Operators of the Harvard Mark II computer found a moth trapped in a relay.
- The term "bug" had been used in engineering before, but this incident popularized it in computing.
- The moth was taped into the logbook with the note: "First actual case of bug being found."
- **Debugging** literally started as removing insects from hardware.

The first computer "bug" from the Harvard Mark II log book.

1950s-1960s: The Dawn of Software Debugging

- **Print Statement Debugging**: The simplest and most enduring method. Programmers inserted print statements to trace program execution and variable states.
- **Core Dumps**: Analyzing the contents of memory after a program crash. A low-level, complex, but powerful technique.
- **Ad-hoc Testing**: Testing was manual, informal, and often performed by the developers themselves near the end of the project.
- **The Challenge**: As software grew in complexity, these methods became insufficient. The "software crisis" was, in part, a crisis of quality and reliability.

```
printf("Got here!  Value of x = %d \n", x);
```

1970s-1980s: Formalizing Testing and Tools

The Rise of Testing Theory

- **Black-box vs. White-box Testing**: Distinguishing between testing functionality (black-box) and testing internal structures (white-box).
- **Levels of Testing**: Unit, Integration, System, and Acceptance testing became standard concepts.
- **Static Analysis**: Compilers began to include more sophisticated warnings for potential code issues.

The First Debugging Tools

- **Symbolic Debuggers** (e.g., gdb): Allowed programmers to interact with a running program, set breakpoints, and

Early GNU Debugger (GDB) interface.

1990s: Automation and Process Integration

- **Automated Regression Testing**: Tools like JUnit (1997) for Java, created by Kent Beck and Erich Gamma, revolutionized testing.
  - Tests could be run automatically and frequently.
  - Provided a safety net for refactoring and adding new features.
  - Embodied the **Test-Driven Development (TDD)** philosophy.
- **Testing Becomes a Discipline**: The role of **Quality Assurance (QA) Engineer** became specialized.
- **Continuous Integration (CI)**: Tools like CruiseControl automated the process of building and testing software after every change, catching integration bugs early.

2000s-2010s: Scaling and Shifting Quality Left

Paradigm Shifts

- **Shift-Left Testing**: The idea of testing earlier in the development lifecycle, involving QA and writing tests during development, not after.
- **Test Automation Pyramid**: A strategy for a balanced test suite: many fast, cheap Unit tests; fewer Integration tests; even fewer UI tests.
- **DevOps and Quality**: With rapid deployments, automated testing became non-negotiable. Quality is everyone's responsibility.

The Test Automation Pyramid.

New Frontiers

- **Selenium**: Automated web browser

2020s: The AI-Powered Future of Debugging

- **AI-Assisted Testing**:
  - AI generates test cases, predicts flaky tests, and optimizes test suites.
  - Tools can automatically detect visual regressions in UI.
- **Intelligent Fault Localization**:
  - AI analyzes code, execution traces, and bug reports to suggest the most likely lines of code causing a failure.
  - **Example**: A tool like **Amazon CodeGuru** or **OpenAI's Debugger** can point directly to the suspicious code.
- **Automated Program Repair**:
  - LLMs can not only find bugs but also suggest and even generate fixes.
  - **Example**: GitHub Copilot Chat can explain a bug and propose a patch.
- **Observability**: The evolution beyond monitoring. Using logs, metrics, and traces (the three pillars) to understand the internal state of a system by its external outputs, making debugging in production much more effective.

The Evolution of Debugging at a Glance

Era	Primary Method	Testing Focus	Ke
1940s-50s	Physical Inspection	Ad-hoc	Th
1960s-70s	Print Statements	Manual, Late	Sy
1980s-90s	Interactive Debuggers (gdb)	Formalized Test Levels	Au
2000s-10s	CI/CD Integrated Tools	Shift-Left, Automation	Tes
2020s+	AI-Powered Analysis	AI-Generated Tests	Int

The Unchanging Goal

To move from reactive bug-fixing to proactive bug-prevention, and to reduce the time between discovering a failure and fixing it from months to minutes.

Outline

Software Testing

Software testing (English: software testing) describes a process used to promote the verification of the correctness, completeness, security, and quality of software. Accordingly, you might think that software testing can never fully establish the correctness of any computer software. However, a simple mathematical proof in computability theory (a branch of computer science) deduces the following result: It is impossible to completely solve the so-called "halting problem," which refers to whether any computer program will enter an infinite loop or halt and produce output. In other words, software testing is an audit or comparison process between actual output and expected output.

The classical definition of software testing is: the process of operating a program under specified conditions to discover program errors, measure software quality, and evaluate whether it can meet design requirements. [1]

[1] https://en.wikipedia.org/wiki/Software_testing

Testing Phases

Unit Testing

Unit testing tests the individual components of software with the purpose of verifying the correctness of the basic units of software design. The object of testing is the smallest unit of software design: functions.

Testing Phases

Integration Testing

Integration testing, also known as comprehensive testing, assembly testing, or joint testing, involves assembling program modules using appropriate integration strategies and testing the interfaces and integrated functionality. Its main purpose is to check whether the interfaces between software units are correct. The objects of integration testing are modules that have already undergone unit testing.

Testing Phases

System Testing

System testing mainly includes functional testing, interface testing, reliability testing, usability testing, and performance testing. Functional testing primarily focuses on aspects such as functional availability, degree of functional implementation (functional processes & business processes, data processing & business data processing).

Testing Phases

Regression Testing

Regression testing refers to testing activities conducted during the software maintenance phase to detect errors introduced by code modifications. Regression testing is an important task in the software maintenance phase, with studies showing that the cost of regression testing accounts for more than one-third of the total cost of the software lifecycle.

Unlike ordinary testing, at the beginning of the regression testing process, testers have a complete set of test cases available. Therefore, how to effectively reuse existing test case sets based on code modifications is an important direction in regression testing research. Additionally, research directions in regression testing involve automation tools, object-oriented regression testing, test case prioritization, and supplementary generation of regression test cases.

Unit Testing

In computer programming, unit testing (English: Unit Testing), also known as module testing, is testing work conducted to verify the correctness of program modules (the smallest units of software design). A program unit is the smallest testable component of an application. In procedural programming, a unit is a single program, function, procedure, etc.; for object-oriented programming, the smallest unit is a method, including methods in base classes (superclasses), abstract classes, or derived classes (subclasses).

Each ideal test case is independent of other cases; to isolate modules during testing, test harness programs such as stubs, mocks, or fakes are often used. Unit testing is typically written by software developers to ensure that their code meets software requirements and follows development goals. Its implementation can be very manual or integrated as part of build automation.[2]

[2] https://en.wikipedia.org/wiki/Unit_testing

Test Oracle

A test oracle, also known as a test criterion, is a mechanism used by software testers or software engineers to determine whether a test has passed. The test oracle determines the expected output of the product for a given test case input, which is then compared with the output of the system under test.
Common test oracles include:

- Design specifications and software documentation
- Other products (for example, as a test oracle for a software program, it might be another program that computes the same mathematical expression using a different algorithm)
- "Heuristic oracles" that provide approximate or accurate results for a small set of test inputs
- "Statistical oracles" that use statistical features
- "Model-based oracles" that use the same model to generate and verify system behavior

Test Oracle


晚上11:22
直播
学习
关注
推荐
小韩你们工地最近怎么停工了?
那天大家闲的没事干，把各自的卷尺拿出来比了比
拼多多买的
淘宝买的
直播买的
五金店买的
路上捡的
2002
104
好家伙，我直呼好家伙
不说了，我准备提桶跑路了
129
@建工学霸
提桶跑路才是工地人最后的归宿#工地 #卷尺 #工程人 #建筑工地
@古月法师创作的原声一;

unittest — Unit testing framework

The unittest unit testing framework was originally inspired by JUnit and has a similar flavor as major unit testing frameworks in other languages. It supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework.

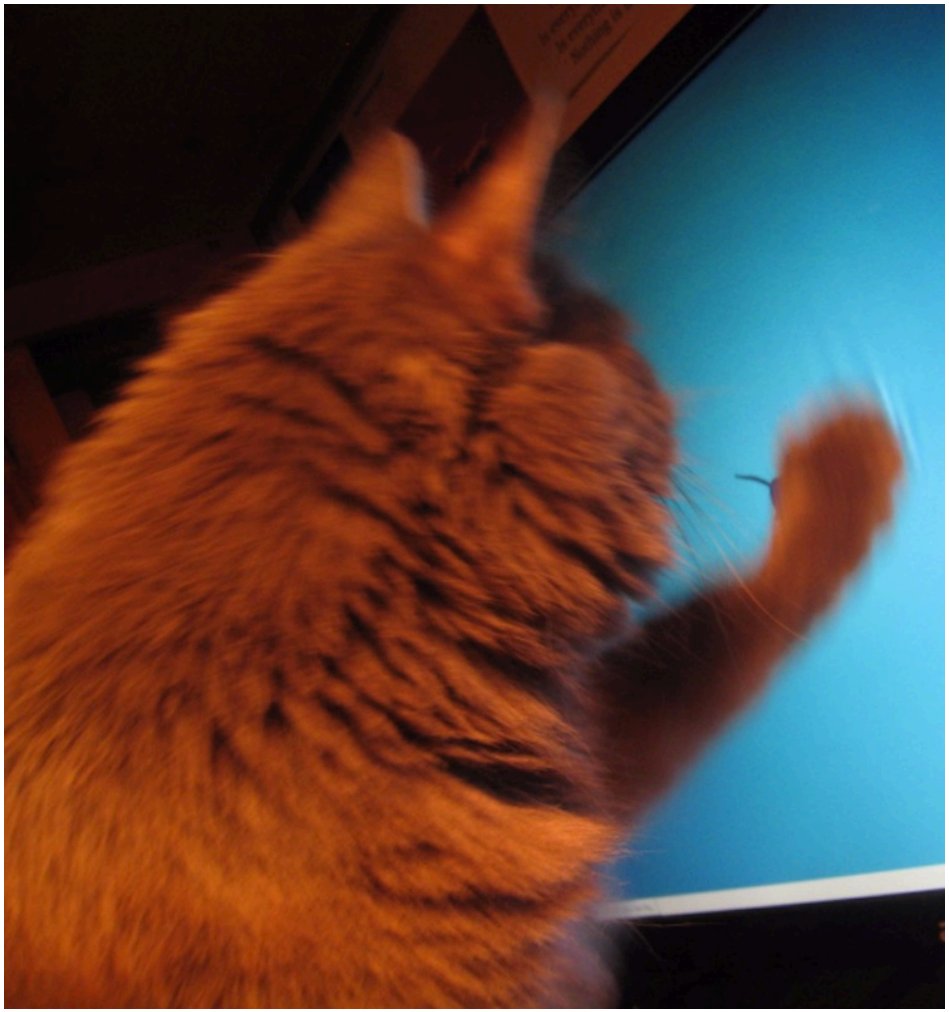
Fun Fact

Xfce Bug #12117

The defualt wallpaper is having my animal scritch all the plastic off my LED MONITOR! Can we choose a different wallpaper? I cannot expect the scratches and whu not? Let's end the mouse games over here.[a]

[a] https://bugzilla.xfce.org/show_bug.cgi?id=12117

Fun Fact



Fun Fact

Outline

Regression and Smoke Testing

Regression Testing

- Ensures **new changes don't break** existing functionality
- Critical during maintenance and enhancements
- Often automated for efficiency

Smoke Testing

- **Basic functionality check** after build
- Determines if further testing is feasible
- "Build verification testing"

Exploratory and Negative Testing

Exploratory Testing

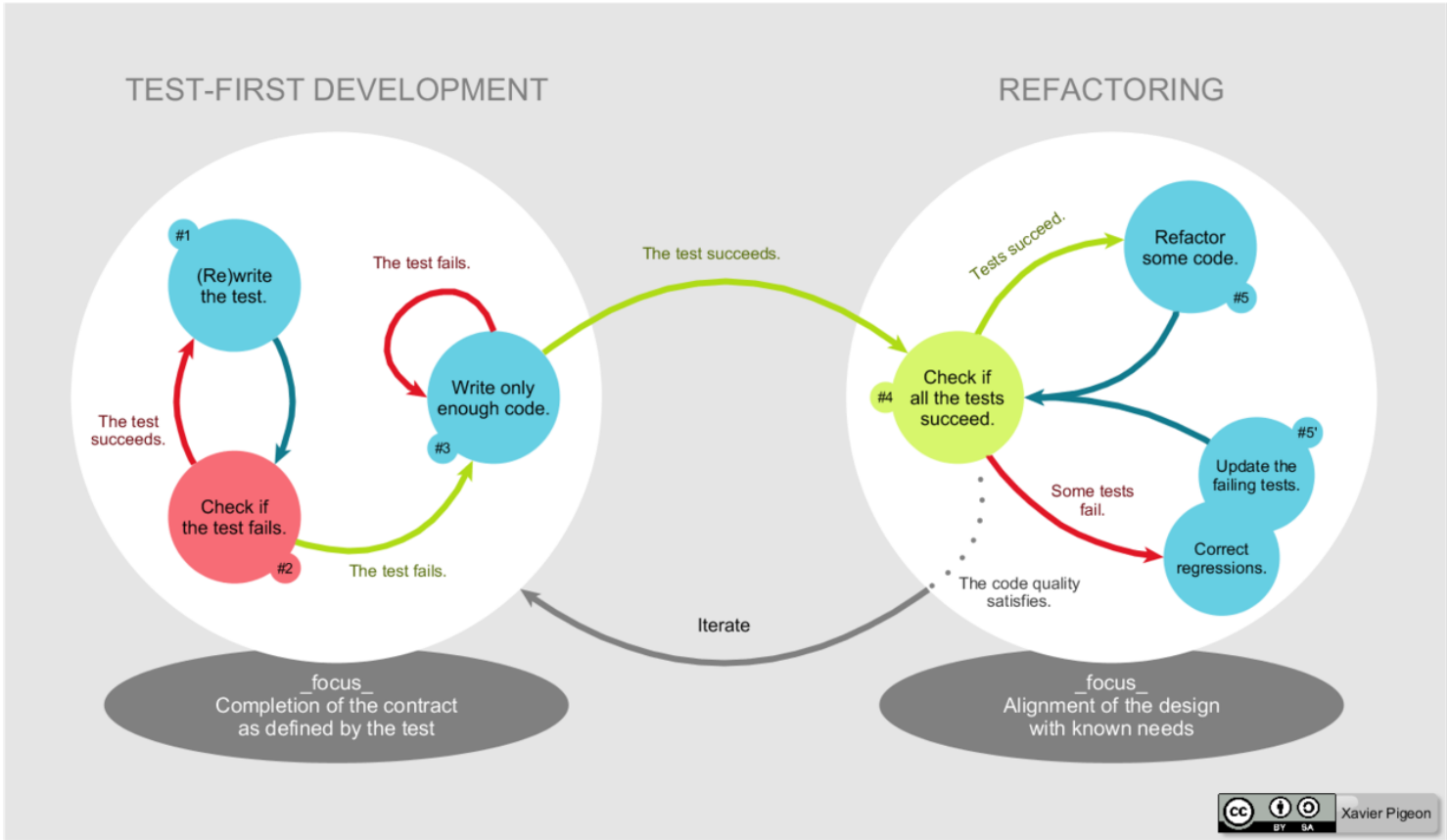
- **Simultaneous learning and testing**
- Relies on tester's creativity
- Unscripted approach

Negative Testing

- Validates system with **invalid inputs**
- Checks **error handling**
- Ensures system robustness

Outline

测试驱动开发[3]



[3] https://en.wikipedia.org/wiki/Test-driven_development

Add a test

In test-driven development, each new feature begins with writing a test. Write a test that defines a function or improvements of a function, which should be very succinct. To write a test, the developer must clearly understand the feature's specification and requirements. The developer can accomplish this through use cases and user stories to cover the requirements and exception conditions, and can write the test in whatever testing framework is appropriate to the software environment. It could be a modified version of an existing test. This is a differentiating feature of test-driven development versus writing unit tests after the code is written: it makes the developer focus on the requirements before writing the code, a subtle but important difference.

Run all tests and see if the new test fails

This validates that the test harness is working correctly, shows that the new test does not pass without requiring new code because the required behavior already exists, and it rules out the possibility that the new test is flawed and will always pass. The new test should fail for the expected reason. This step increases the developer's confidence in the new test.

Write the code

The next step is to write some code that causes the test to pass. The new code written at this stage is not perfect and may, for example, pass the test in an inelegant way. That is acceptable because it will be improved and honed in Step 5.
At this point, the only purpose of the written code is to pass the test. The programmer must not write code that is beyond the functionality that the test checks.

Run tests

If all test cases now pass, the programmer can be confident that the new code meets the test requirements, and does not break or degrade any existing features. If they do not, the new code must be adjusted until they do.

Refactor code

The growing code base must be cleaned up regularly during test-driven development. New code can be moved from where it was convenient for passing a test to where it more logically belongs. Duplication must be removed. Object, class, module, variable and method names should clearly represent their current purpose and use, as extra functionality is added. As features are added, method bodies can get longer and other objects larger. They benefit from being split and their parts carefully named to improve readability and maintainability, which will be increasingly valuable later in the software lifecycle.

Repeat

Starting with another new test, the cycle is then repeated to push forward the functionality. The size of the steps should always be small, with as few as 1 to 10 edits between each test run. If new code does not rapidly satisfy a new test, or other tests fail unexpectedly, the programmer should undo or revert in preference to excessive debugging. Continuous integration helps by providing revertible checkpoints. When using external libraries it is important not to make increments that are so small as to be effectively merely testing the library itself,[4] unless there is some reason to believe that the library is buggy or is not sufficiently feature-complete to serve all the needs of the software under development.

Outline

Fuzz Testing

In programming and software development, fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks[4].

[4] https://en.wikipedia.org/wiki/Fuzzing

Differential Testing

Differential testing, also known as differential fuzzing, is a software testing technique that detect bugs, by providing the same input to a series of similar applications (or to different implementations of the same application), and observing differences in their execution. Differential testing complements traditional software testing because it is well-suited to find semantic or logic bugs that do not exhibit explicit erroneous behaviors like crashes or assertion failures. Differential testing is also called back-to-back testing[5].

[5] https://en.wikipedia.org/wiki/Differential_testing

Metamorphic Testing

Metamorphic testing (MT) is a property-based software testing technique, which can be an effective approach for addressing the test oracle problem and test case generation problem. The test oracle problem is the difficulty of determining the expected outcomes of selected test cases or to determine whether the actual outputs agree with the expected outcomes[6].

[6] https://en.wikipedia.org/wiki/Metamorphic_testing

Metamorphic Testing

Metamorphic relations (MRs) are necessary properties of the intended functionality of the software, and must involve multiple executions of the software. Consider, for example, a program that implements sin x correct to 100 significant figures; a metamorphic relation for sine functions is $sin(\pi - x) = sin(x)$. Thus, even though the expected value of $sinx_1$ for the source test case $x_1 = 1.234$ correct to the required accuracy is not known, a follow-up test case $x_2 = \pi - 1.234$ can be constructed. We can verify whether the actual outputs produced by the program under test from the source test case and the follow-up test case are consistent with the MR in question. Any inconsistency (after taking rounding errors into consideration) indicates a failure:31 of the program, caused by a fault:31 in the implementation.

Differential Testing

Differential testing, also known as differential fuzzing, is a software testing technique that detect bugs, by providing the same input to a series of similar applications (or to different implementations of the same application), and observing differences in their execution. Differential testing complements traditional software testing because it is well-suited to find semantic or logic bugs that do not exhibit explicit erroneous behaviors like crashes or assertion failures. Differential testing is also called back-to-back testing[7].

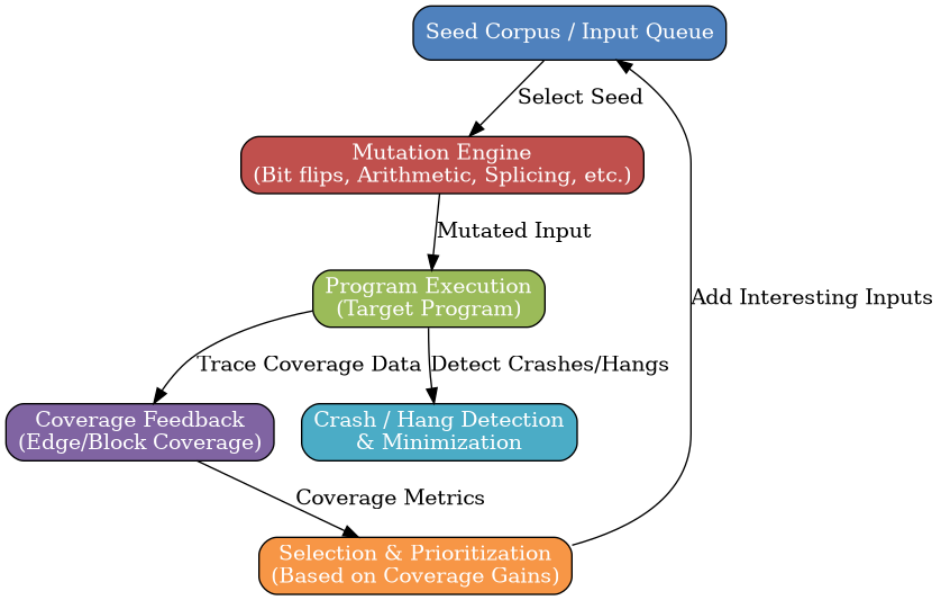
[7] https://en.wikipedia.org/wiki/Differential_testing

EvoSuite

EvoSuite is a tool that automatically generates unit tests for Java software. EvoSuite uses an evolutionary algorithm to generate JUnit tests. EvoSuite can be run from the command line, and it also has plugins to integrate it in Maven, IntelliJ and Eclipse. EvoSuite has been used on more than a hundred open-source software and several industrial systems, finding thousands of potential bugs[8].

[8] https://en.wikipedia.org/wiki/EvoSuite

Evolutionary Fuzz Testing



Main Modules Description

- **Seed Corpus / Input Queue (Blue)**
  - Holds the initial set of seed inputs and manages the queue of inputs waiting to be processed.
- **Mutation Engine (Red)**
  - Generates new test inputs by applying mutation strategies (e.g., bit flips, arithmetic operations, splicing).
- **Program Execution (Green)**
  - Executes the target program using mutated inputs, collects trace coverage data, and monitors for crashes or hangs.
- **Crash / Hang Detection & Minimization (Cyan)**
  - Detects crashes or hangs during execution and minimizes the problematic inputs for debugging.
- **Coverage Feedback (Purple)**
  - Collects data on which parts of the program were exercised by the inputs to guide the fuzzing process.
- **Selection & Prioritization (Orange)**
  - Selects and prioritizes inputs based on coverage metrics to discover new paths or trigger crashes.

Overall Flow

1. **Initialization**
   - Start with a seed corpus or input queue containing initial test inputs.
2. **Mutation**
   - Select seeds from the input queue and apply mutation strategies to generate new inputs.
3. **Execution**
   - Execute the target program with mutated inputs, collect coverage data, and detect crashes or hangs.
4. **Detection and Minimization**
   - Record detected crashes or hangs and minimize the inputs for debugging.
5. **Feedback and Prioritization**
   - Use coverage feedback to evaluate inputs, select and prioritize those that improve coverage.
6. **Iteration**
   - Iterate the process, refining inputs based on coverage feedback to maximize code exploration and crash discovery.

Outline

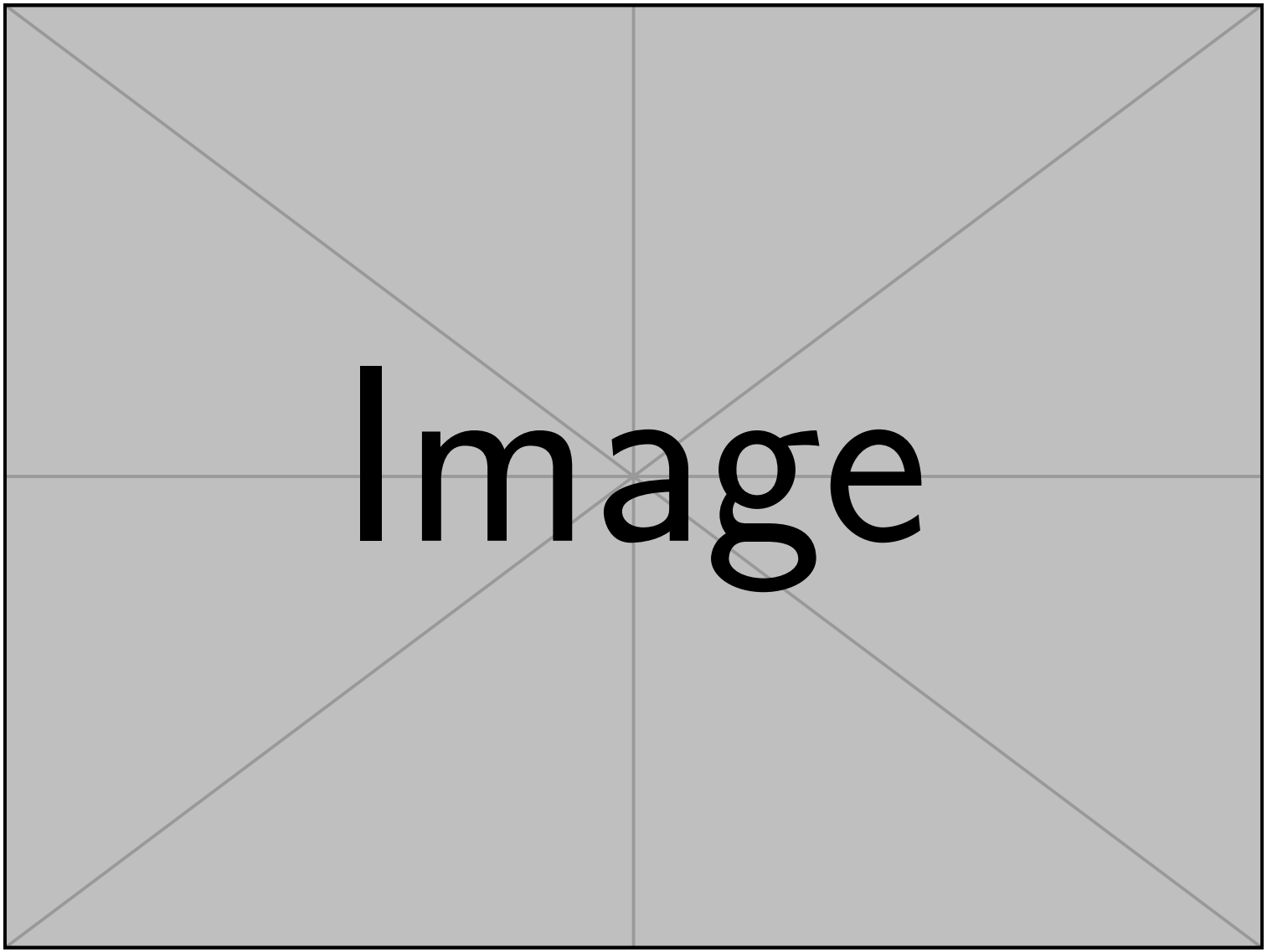
Background

- Robotic software development exhibits stark contrasts to traditional software engineering paradigms
- Simulators are a critical component of the software development toolchain for robotic systems
- Gazebo has emerged as the *de facto standard* for simulation in robotics
- Applications: aerospace missions, subterranean exploration, maritime operational simulations

Background: Challenges

- Bugs in robotic simulators may result in:
  - Simulation errors
  - Data loss
  - Vulnerability to malicious exploitation
  - Compromised efficiency and quality of software development
- Testing robotic simulators like Gazebo is difficult due to:
  - Strict Input Challenge
  - State Space Challenge

Strict Input Challenge (Input File)



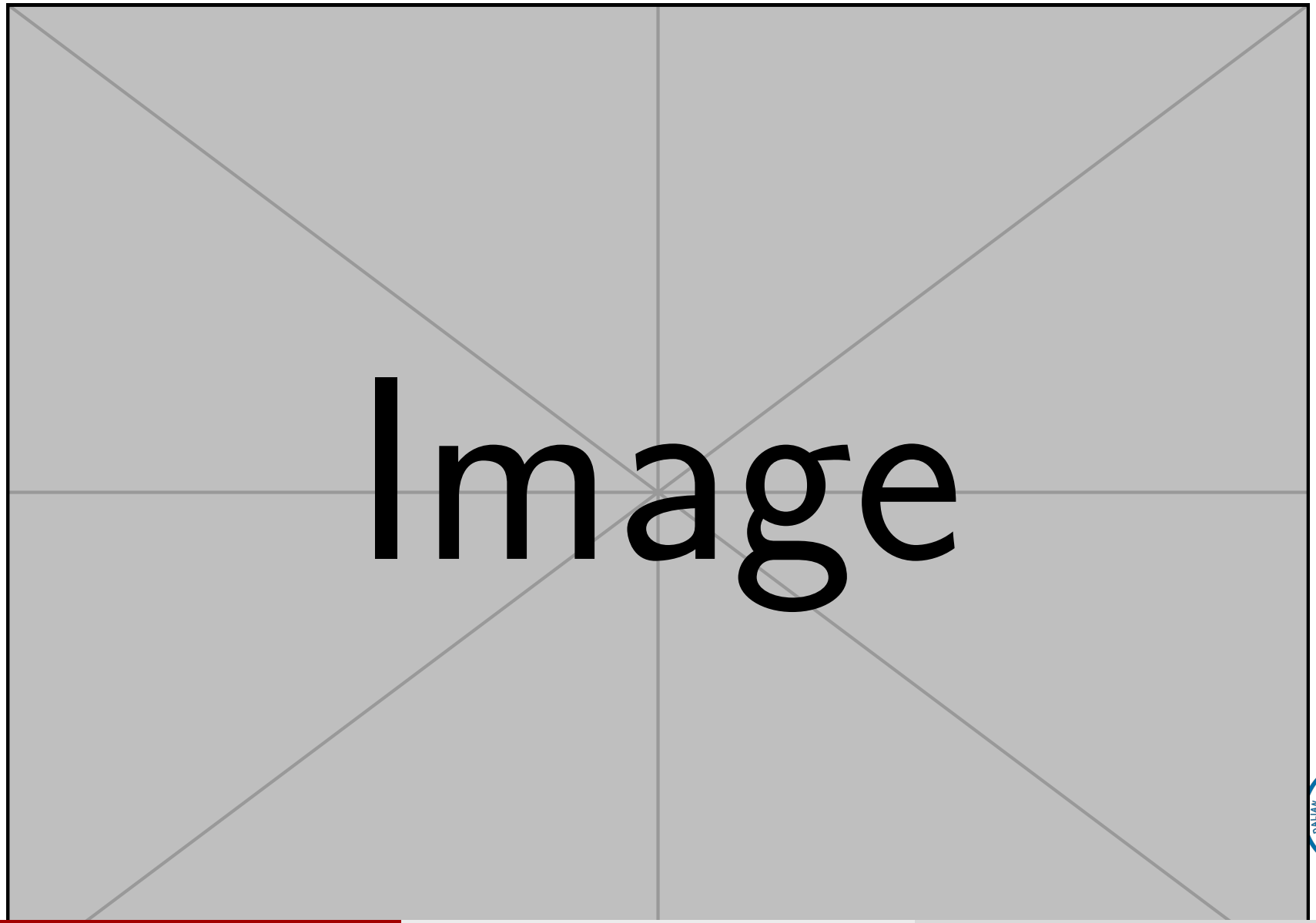
Strict Input Challenge (Command)

```
gz service --timeout 10000 \
-s /world/default/create \
--reptype gz.msgs.Boolean \
--reqtype gz.msgs.EntityFactory \
--req 'sdf: "<model name=\"vehicle_blue\">...</model>"
pose {
  position {
    x: 2.3509603833593182
    y: 5.568459723740165
    z: 9.435891851259887
  }
}
name: "model"'
```

State Space Challenge

- More than 100 plugins/modules
- Joints/links/sensors dynamically created
- More than 50 built-in services and topics
- Plugin-specific services and topics
- Command sequence of arbitrary length

Our Approach: GzFuzz



Syntax-aware Feasible Command Generation

- For Generators 6–9, further determine specific parameters
- Ensures commands are both feasible and diverse

Learning-based Command Generator Selection

- Simplified hierarchical reinforcement learning (Advantage Actor-Critic)
- Actors select command generators
- Sub-Actors select parameters
- Critic updates parameters of Actors and Sub-Actors
- Reward based on crash, coverage improvement, and diversity

Bug Detection

- Commands executed with spawned Gazebo
- Crash-based test oracle
- Once crash detected:
  - Command sequence iteratively reduced
  - Manually checked
  - Bug reports issued

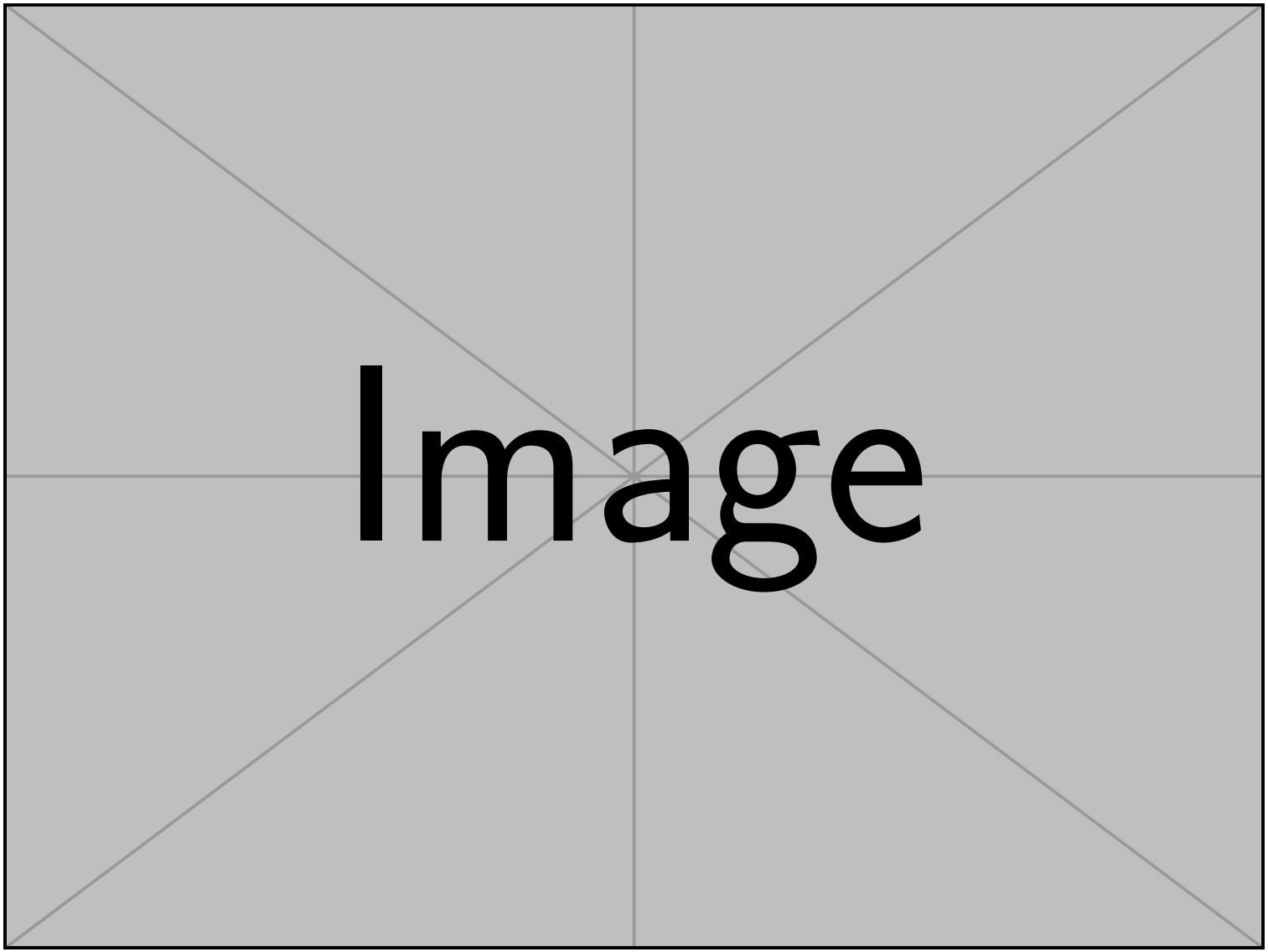
Evaluation: Research Questions

- **RQ1**: Is GzFuzz effective in finding bugs for Gazebo?
- **RQ2**: How effective is GzFuzz compared with existing fuzzing methods?
- **RQ3**: What is the impact of the two main components on GzFuzz?

Evaluation Setup

- Baselines: AFL++, Fuzzotron
- Gazebo Version: latest dev (RQ1), Gazebo 8.0.0 (RQ2, RQ3)
- Data: 309 SDF files, 123 modules with plugins, 117 plugins
- Implementation: PyTorch, lxml, randomproto

Evaluation Results



Answer to RQ1

- GzFuzz detected **25 unique crashes** in less than six months
- **24 bugs** were fixed or confirmed

Answer to RQ2

- GzFuzz significantly outperforms baseline approaches
- Over 12-hour testing: more bugs detected on previous Gazebo release
- Statistically significant improvement

Answer to RQ3

- Syntax-aware generators and RL-based selection both contribute
- Two mechanisms collectively enhance effectiveness

Contributions

- First study to apply fuzz testing to Gazebo
- Syntax-aware feasible command generators
- Learning-based command generator selection
- 25 unique crashes detected, 24 fixed/confirmed

Future Work

- Incorporate differential testing across simulators
- Integrate LLMs for more complex and realistic test cases

Thank You

Zhilei Ren
zren@dlut.edu.cn