Mutation Testing

CS454 AI-based Software Engineering

Shin Yoo | COINSE@KAIST

Mutation Testing

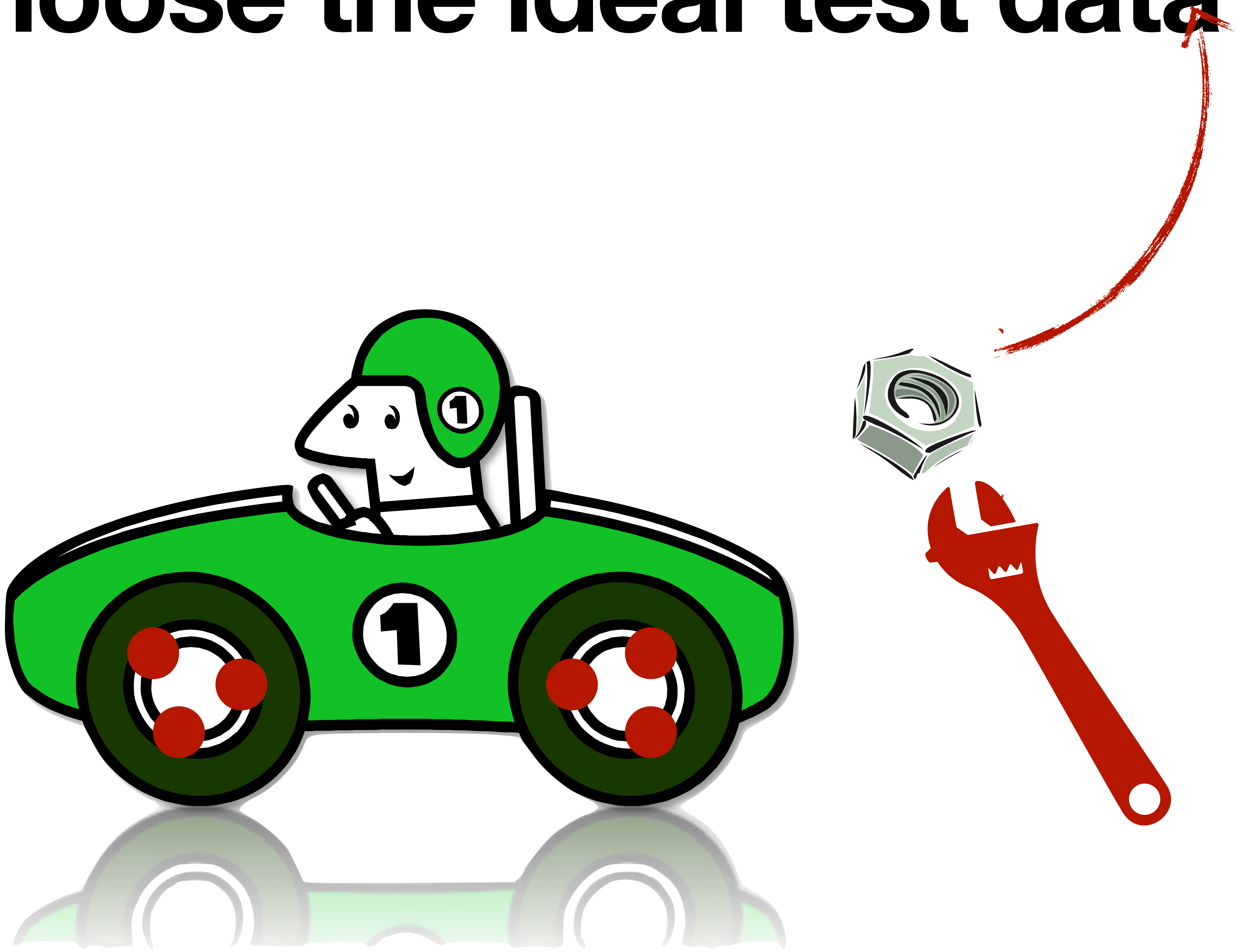
- White-box, fault-based testing technique
- Inverts the testing adequacy: the goal is to assess the effectiveness of the existing test suite in terms of its fault detection capabilities.
  - Test suites test programs
  - Mutants test test suites
- The most widely used adequacy score is called Mutation Score: it measures the quality of the given test suite as the percentage of injected faults that you can detect.

How do you choose the ideal test data?

How do you choose the ideal test data?

How do you demonstrate that the bendy road is the better test environment?

How do you choose the ideal test data?



Sabotage the car!

How do you choose the ideal test data?

Some tests are kinder(?) to faults: we want tests that are mean to faults.

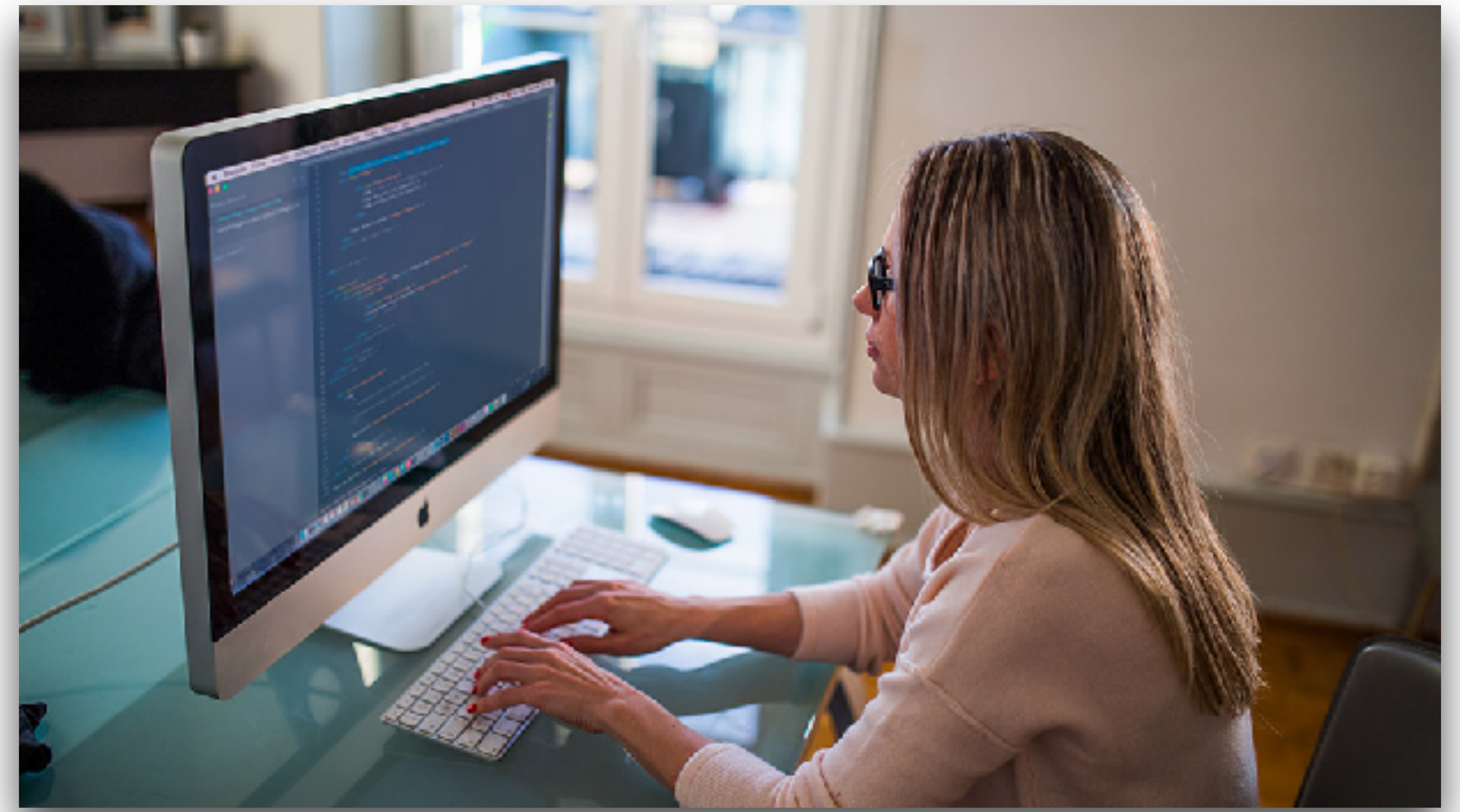
Mutation Testing

- Testing is a sampling process: without a priori knowledge of faults, it is hard to assess how well a technique samples.
- Mutation testing: the quality of a test suite can be indirectly measured by artificially injecting faults and checking how well the test suite can detect them.
    - Seed the original implementation with faults (the seeded versions are called mutants)
    - Execute the given test suite
    - If we get different test results, the introduced faults (the mutant) has been identified (i.e., the mutant is killed). If not, the mutant is still alive.

Fundamental Hypothesis

- Competent Programmer Hypothesis
- Coupling Effect Hypothesis

Competent Programmer Hypothesis

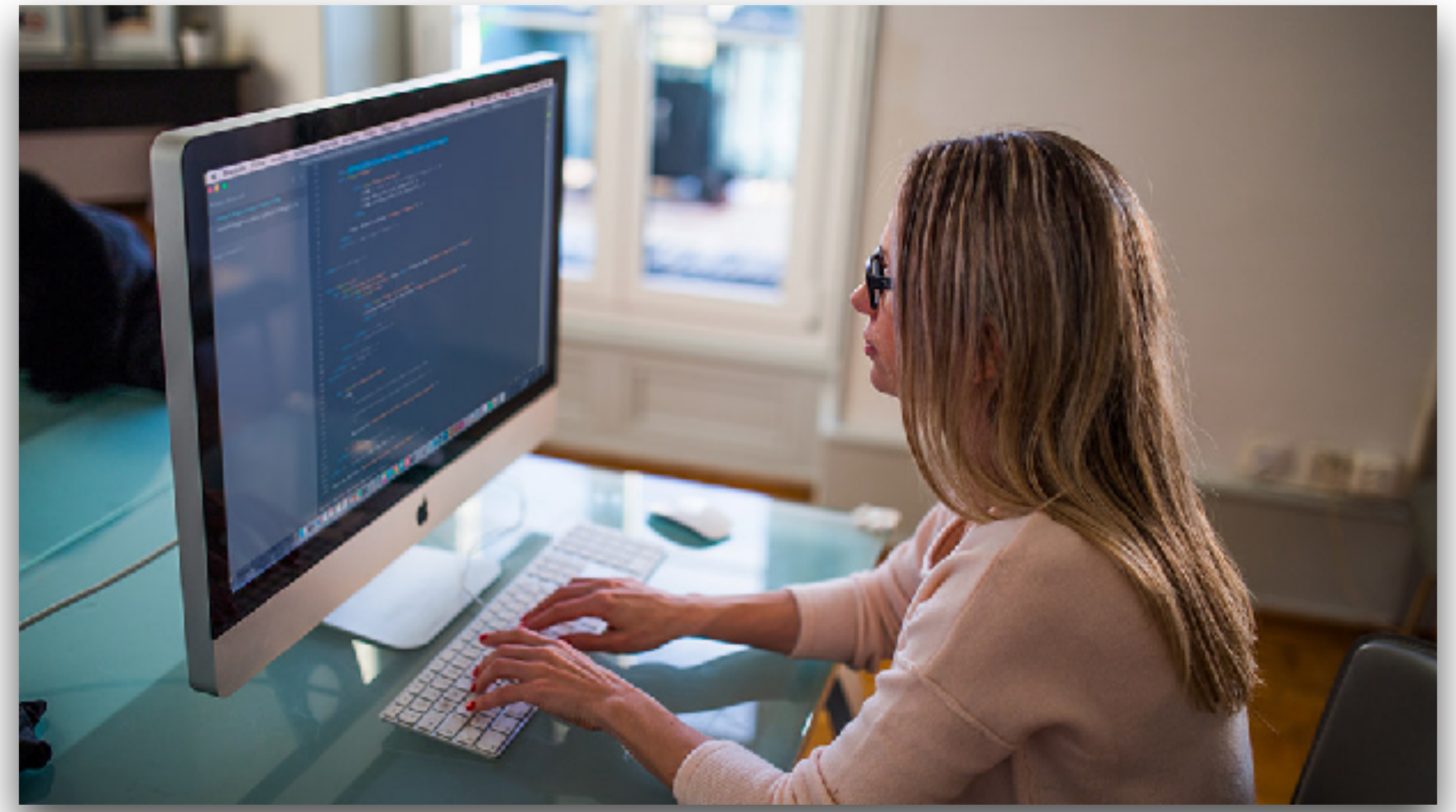


Q: what do the programmers and the monkeys have in common when it comes to programming?

A: they write buggy code.

Competent Programmer Hypothesis

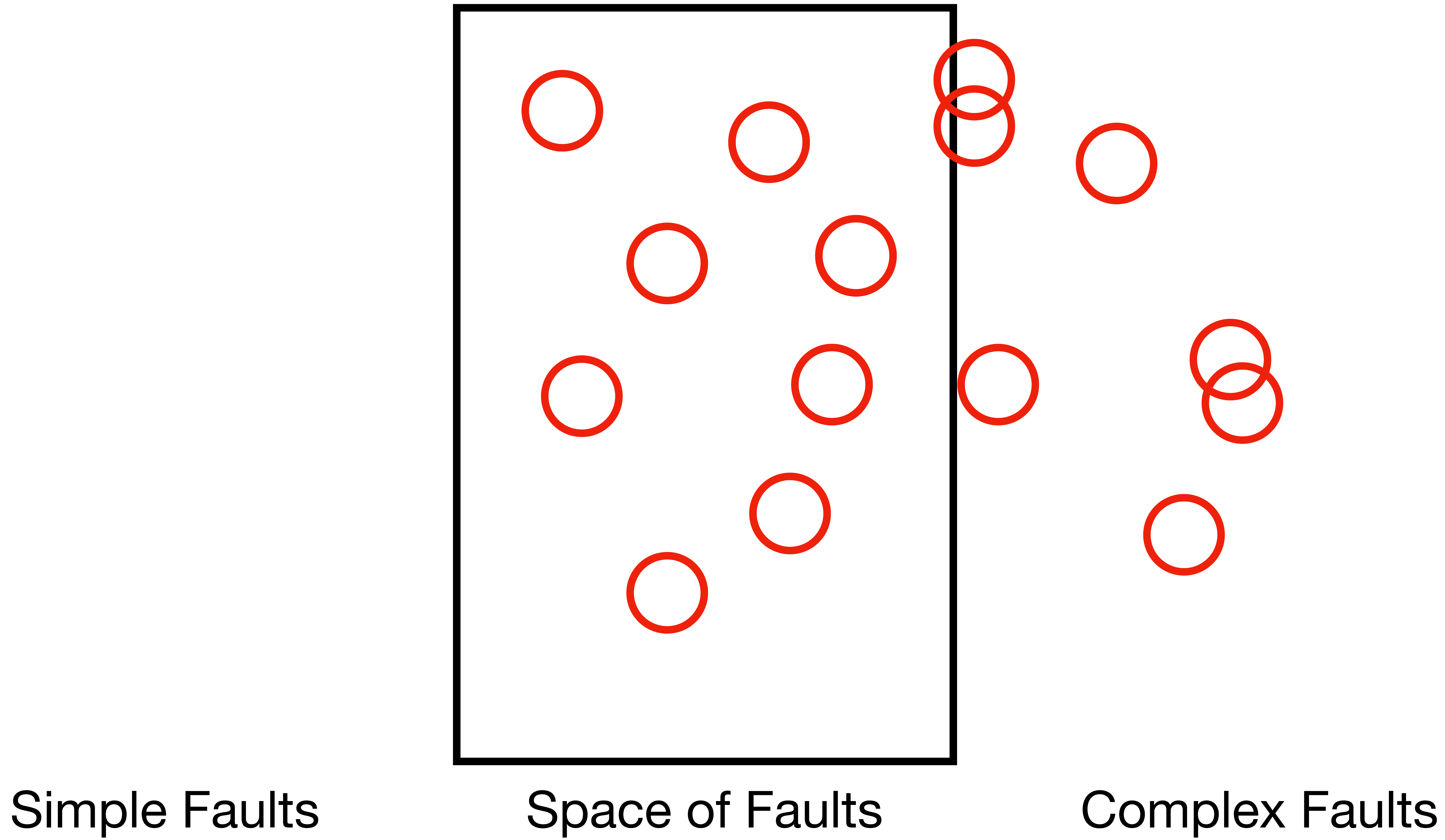
- On average, programmers are competent, i.e., they write almost correct programs. A faulty program source code is different from the correct one only in a few, minor detail.



Coupling Effect Hypothesis

- If a test set detects all small syntactic faults, it will also detect larger, semantic faults: especially if those semantic faults are coupled with the small faults.
  - Richard A. DeMillo and Richard J. Lipton and Frederick Gerald Sayward, Hints on Test Data Selection: Help for the Practicing Programmer, Computer, 11(4), 1978.
  - A. Jefferson Offutt, Investigations of the Software Testing Coupling Effect , ACM Transactions on Software Engineering and Methodology, 1(1), January 1992.

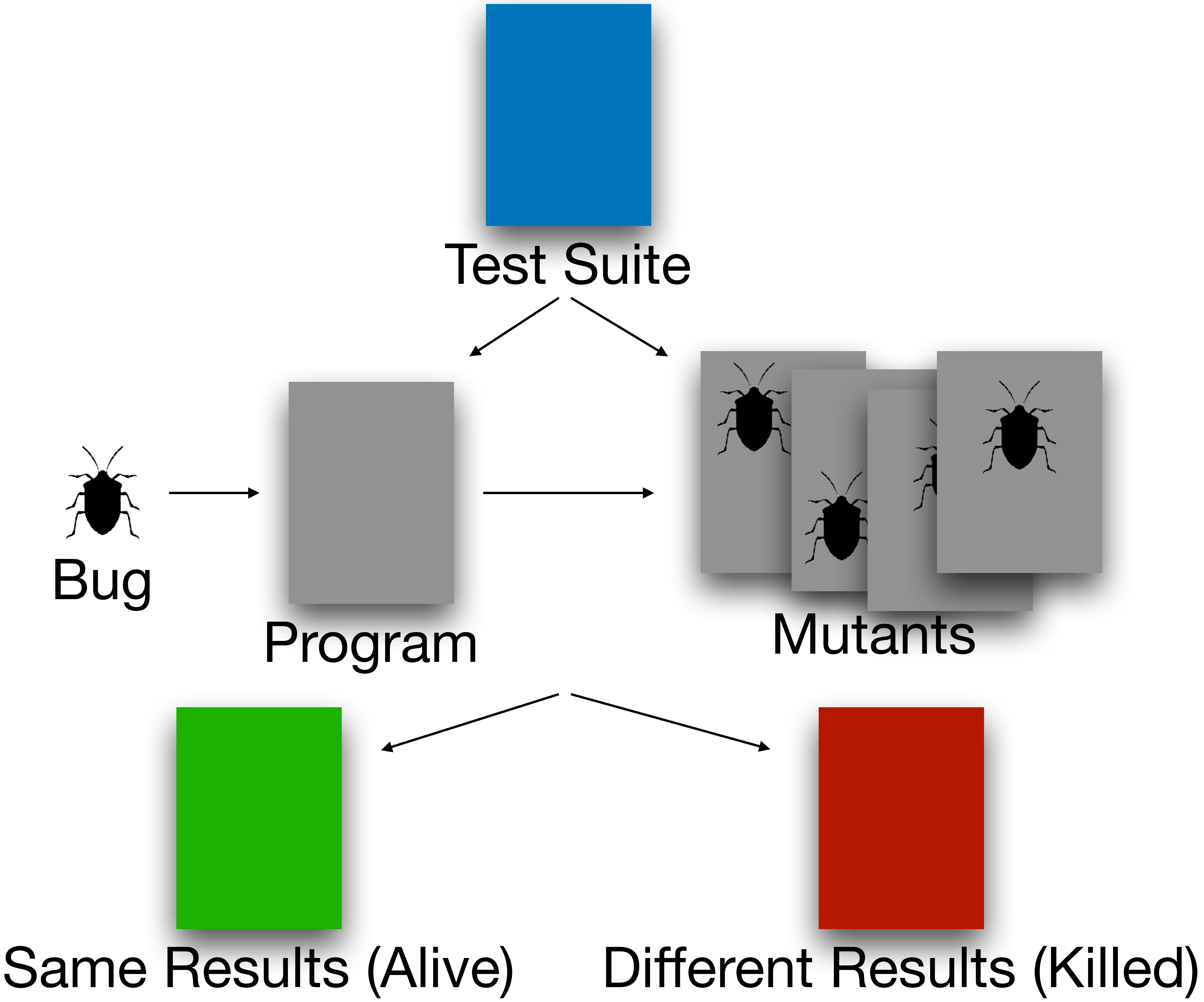
Coupling Effect Hypothesis



Fundamental Hypothesis

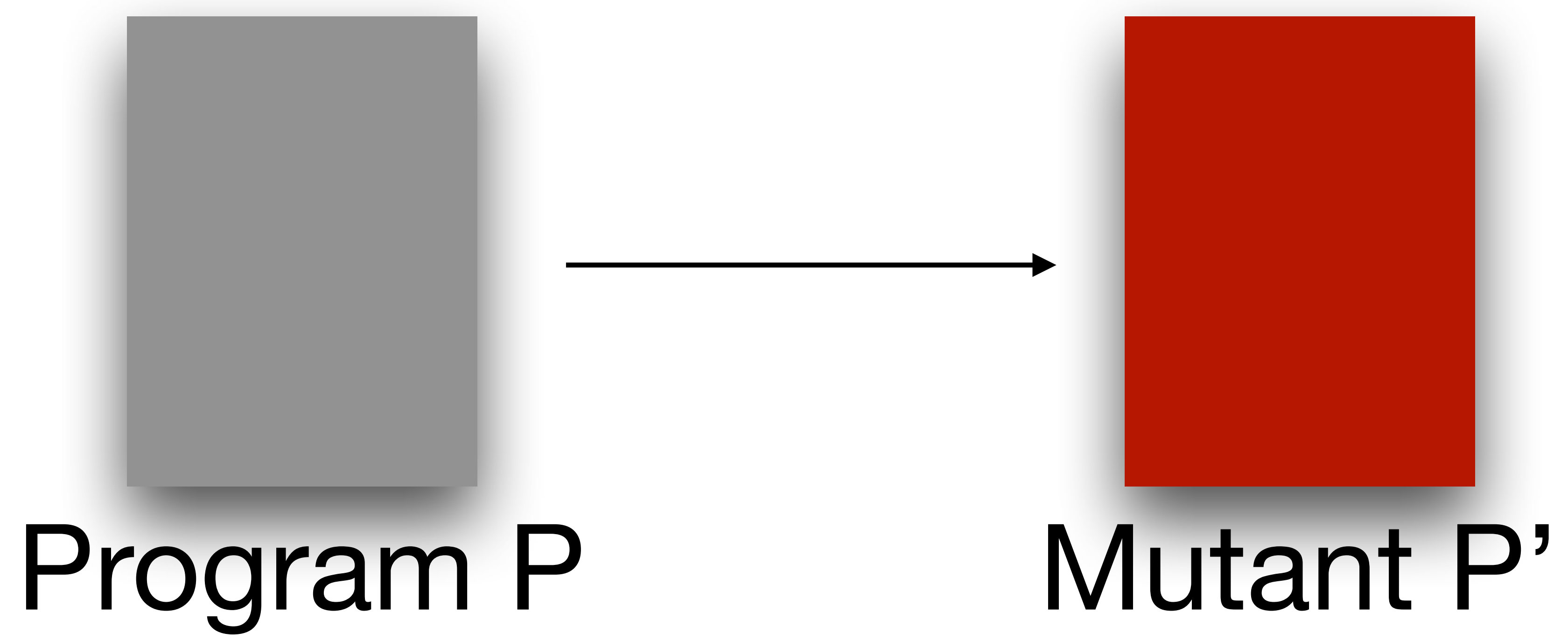
- Competent Programmer Hypothesis: programmers are likely to make simple faults.
- Coupling Effect Hypothesis: if we catch all the simple faults, we will be able to catch more complicated faults.
- Mutation testing: therefore, let us artificially inject simple faults!

Mutation Testing Process



Mutant Generation

- P’ differs from P by a single mutation
- Mutation: a typical simple error programmers are likely to make - off by one, typo, mistaken identity, etc.



Mutation Operator

An atomic rule that is used to generate a mutant

ABS: Absolute Value Insertion

```
                          x = 4 * abs(y);

x = 4 * y;  ——————→       x = 4 * -abs(y);

                          x = 4 * failOnZero(y);
```

A Fortran Language System for Mutation-Based Software Testing, Kim N. King and Jeff Offutt.
Software Practice and Experience, 21(7):686-718, July 1991

Mutation Operator

An atomic rule that is used to generate a mutant

AOR: Arithmetic Operator Replacement

```
x = y + z;   ──────→   x = y * z;
                       x = y - z;
                       x = y / z;
```

A Fortran Language System for Mutation-Based Software Testing, Kim N. King and Jeff Offutt.
Software Practice and Experience, 21(7):686-718, July 1991

Mutation Operator

An atomic rule that is used to generate a mutant

ROR: Relational Operator Replacement

```
if(x >= y)    ──────►    if(x > y)
                         if(x == y)
                         if(x < y)
                         if(x!=y)
                         ...
```

A Fortran Language System for Mutation-Based Software Testing, Kim N. King and Jeff Offutt. Software Practice and Experience, 21(7):686-718, July 1991

Mutation Operator

An atomic rule that is used to generate a mutant

COR: Conditional Operator Replacement

```
                      if(x || y)

if(x && y)   ──────►  if(x & y)

                      if(x | y)
```

A Fortran Language System for Mutation-Based Software Testing, Kim N. King and Jeff Offutt.
Software Practice and Experience, 21(7):686-718, July 1991

Mutation Operator

An atomic rule that is used to generate a mutant

SDL: Statement Deletion

```
x = 3;                         x = 3;

y = x + 5;     ──────────►

z = x - y;                     z = x - y;
```

A Fortran Language System for Mutation-Based Software Testing, Kim N. King and Jeff Offutt.
Software Practice and Experience, 21(7):686-718, July 1991

Mutation Operator

- Any systematic and syntactic change operator can be considered.
- For C: 71 Mutation Operators (Statement 15, Operator 46,Variable 7, Constant 3)
  - Design of Mutant Operators for the C Programming Language
    by Hiralal Agrawal, Richard A DeMillo, R Hathaway,William Hsu,Wynne Hsu, Edward W Krauser, Rhonda J Martin, Aditya P Mathur, Eugene H Spafford, technical report, Purdue University, 1989
- For Class Mutation: 24 Mutation Operators (Access Control 1, Inheritance 7, Polymorphism 4, Overloading 4, Java-Specific Features 4, Common Programming Mistakes 4)
  - Y.-S. Ma, Y.-R. Kwon, and J. Offutt. Inter-class mutation operators for java. In Proceedings of the 13th International Symposium on Software Reliability Engineering, ISSRE ’02, pages 352–, Washington, DC, USA, 2002. IEEE Computer Society.

Killing a Mutant

Program P

```
x = y + z;
...
print(x);
```

Mutant P’

```
x = y * z;
...
print(x);
```

	Program P	Mutant P’	
Test: y = 2, z = 2	4	4	Alive
Test: y = 3, z = 1	4	3	Killed

Killing a Mutant

```
x = y + y;
...
print(x);
```

Program P

```
x = y * 2;
...
print(x);
```

Mutant P'

	Program P	Mutant P'	
Test: y = 2	4	4	Alive
Test: y = 3	6	6	Alive

Equivalent Mutant

- An equivalent mutant is syntactically different from, but semantically identical to, the original program.
  - x = y + y; vs. x = y * 2;
- Checking whether an arbitrary mutant is equivalent or not is undecidable.
- This is one of the major obstacles to the mainstream adoption of mutation testing.
  - “My mutation score is 70%. Is my test suite bad, or are there too many equivalent mutants?”

Mutation Score

$$MS = \frac{\#\text{ of killed mutants}}{\#\text{ of non-equivalent mutants}}$$

$$MS = \frac{\#\text{ of killed mutants}}{\#\text{ of all mutants}}$$

How to kill a mutant

- Reachability: your test execution needs to reach (i.e. cover) the mutant
- Infection: the mutated code should infect the program state (i.e. the value of the mutated expression differs from the value of the original expression)
- Propagate: the infected state results in an observable state

How to kill a mutant

- Reachability + Infection: weak kill (i.e. we stop after confirming infection, do not check the propagation to the outside world)
- Reachability + Infection + Propagation: strong kill (i.e. the kill can be observed from the outside world)

Killing me ~~softly~~ weakly…

Mutation:

```
if(x < y){
   if(z < y){if(z < y + 1){
      if(x < z)
         result = z;
      else
         result = x;
   }
   else
      result = y;
   }
else
   result = 0;
```

Reachability Condition:

$x < y$

Infection Condition:

$(z < y) \mathrel{!=} (z < y + 1)$

Weak Kill Condition:

$(x < y) \mathbin{\&\&} (z < y) \mathrel{!=} (z < y + 1)$

$\therefore (x < y) \mathbin{\&\&} (z == y)$

Killing me ~~softly~~ strongly…

```
          Mutation:
if(x < y){
  if(z < y){if(z < y + 1){
    if(x < z)
      result = z;
    else
      result = x;
  }
  else
    result = y;
  }
else
  result = 0;
```

Reachability Condition:
x < y

Infection Condition:
(z < y) != (z < y + 1)

Weak Kill Condition:
(x < y) && (z < y) != (z < y + 1)

∴ (x < y) && (z == y)

Propagation Condition:

After infection, x < y == z

Under this condition,
Original: result = y
Mutant: result = z
∴ (x < y) && (z == y) && (y != z)

Scalability

- Normal testing: 1 program * 100 test cases
- Mutation testing: 1 program * 10000 mutants (including compilation!) * 100 test cases…
- We tend to get a large number of mutants:
    - No prior knowledge of which mutation operator is the most effective (w.r.t. improving the test suite quality): the default is to apply everything
    - Programs are large!

Scalability: do fewer

- Mutation Sampling: generate a large number of mutants, but use only a subset of them (natural question: how do we select?)
- Subsuming Mutant: a mutant M_1 subsumes another mutant M_2 if and only if killing M_1 guarantees killing of M_2.
  - True subsumption relationship: not computable
  - Dynamic subsumption: defined w.r.t. a given test suite
  - Static subsumption: results of static analysis, still an approximation
- Selective Mutation: apply only a subset of mutation operators

Scalability: do smarter

- Super-mutant: compile all mutants into a single program, then activate a specific subset at the runtime (saves the compilation time)
- Weak mutation testing: relax the kill criterion to weak kills (requires instrumentation for the embedded oracle)
    - Parallel/distributed mutation testing: obvious.

Trivial Compiler Equivalence

- Idea: some syntactic changes may compile into the same binary code thanks to compiler optimisation - if the binary is the same, the corresponding syntactic change is an equivalent mutant.

- A large scale empirical study showed that TCE can detect 7% of the mutants to be equivalent; more importantly, 21% of all mutants were duplicates (i.e. not equivalent, but identical to another non-equivalent mutant).

- M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In Proceedings of the 37th International Conference on Software Engineering-Volume 1, pages 936–946. IEEE Press, 2015.

Higher Order Mutants

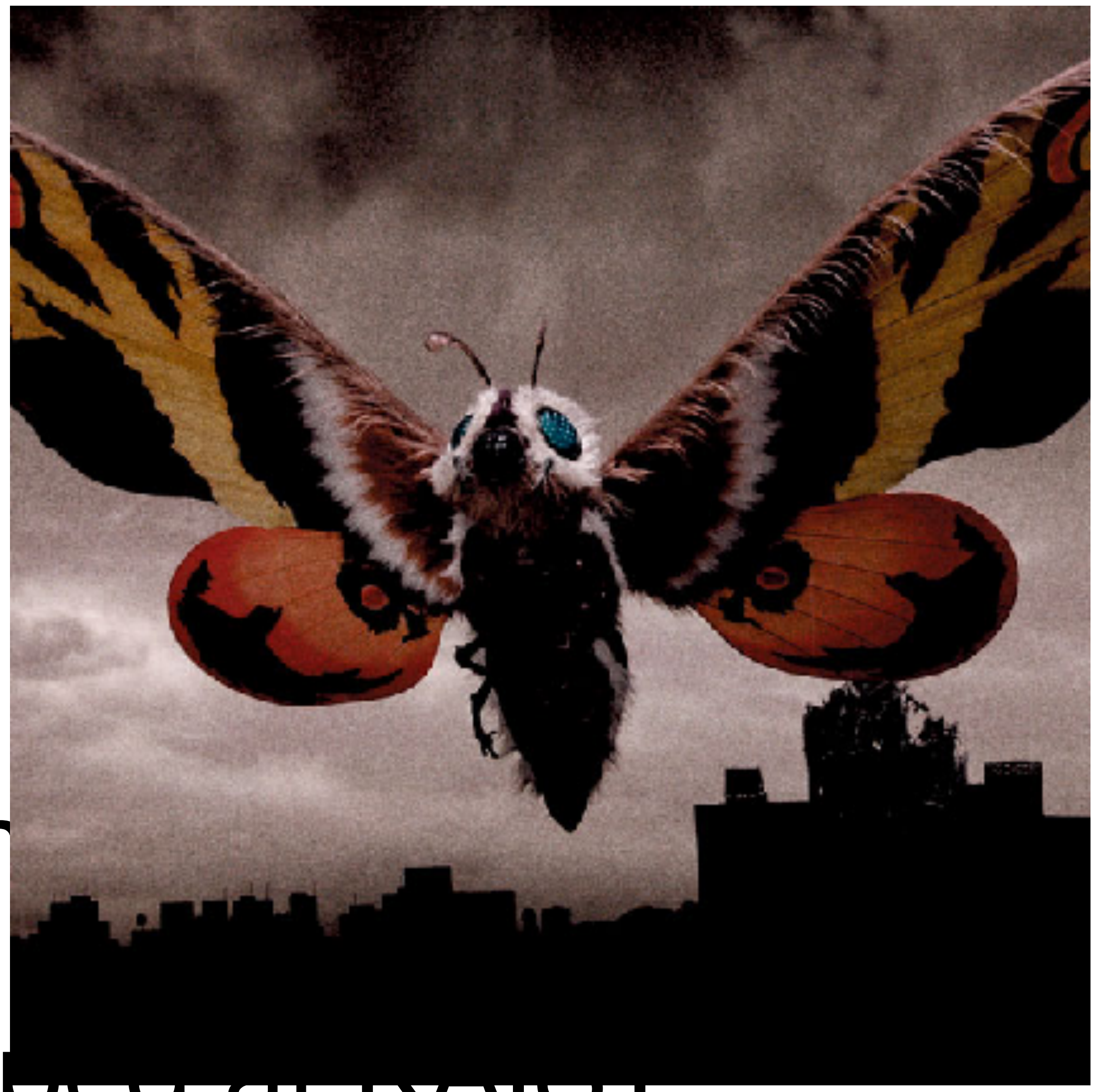
- FOM (First Order Mutant): mutants that are generated by a single application of one mutation operator
- HOM (High Order Mutant): mutants that are generated by two or more applications of a set of mutation operators
- Some studies claim that, while most of the FOMs are trivial to kill, few of them are coupled with real faults.
- We can search for a combination of FOMs that result in a hard-to-kill HOM.

Industrial Scalability

Google has adopted mutation testing for its CI/CD since 2018.

- State of mutation testing at google, Petrovic et al., ICSE-SEIP 2018 (https://ieeexplore.ieee.org/abstract/document/9524503)
    - Perform mutation Just-in-Time, i.e., at each commit; ignore lines that a re arid (i.e., not covered by any existing test case, not likely to be interesting, etc) —> This enables mutation testing
- Does mutation testing improve testing practices? Petrovic. et al., ICSE 2021 (https://ieeexplore.ieee.org/abstract/document/9402038)
    - Shows that developers using mutation testing do write more tests, and mutants are indeed coupled with real bugs.

Tools



- Fortran: Mothra - had a long-lasting impact with its definition of m
- C/C++: Proteum, MiLU (also searches for HOMs), MUSIC (developed at KAIST, https://github.com/swtv-kaist/MUSIC)
- Java: muJava (a special tie to KAIST, https://www.albany.edu/faculty/offutt/mujava/), Major (https://mutation-testing.org/), Javalanche (bytecode mutation), PIT (https://pitest.org/)
- JavaScript: Stryker (https://stryker-mutator.io/)
- Ruby: Heckle (https://github.com/seattlerb/heckle), mutant (https://github.com/mbj/mutant)

Syntactic vs. Semantic Mutation

- The coupling hypothesis is, in some sense, a **necessity**: simple, random syntactic changes are easy to make. Imitating an error that a real developer may make is **significantly** harder, and was almost infeasible around the time mutation testing was born.
- In contrast, semantic mutation aims to introduce a realistic error, even if it requires more than simple syntactic change.
  - How? Learn reverse bug patches!
  - We need 1) a large collection of bug patches, and 2) ways to abstract and learn patterns.

SemSeed

https://dl.acm.org/doi/10.1145/3468264.3468623

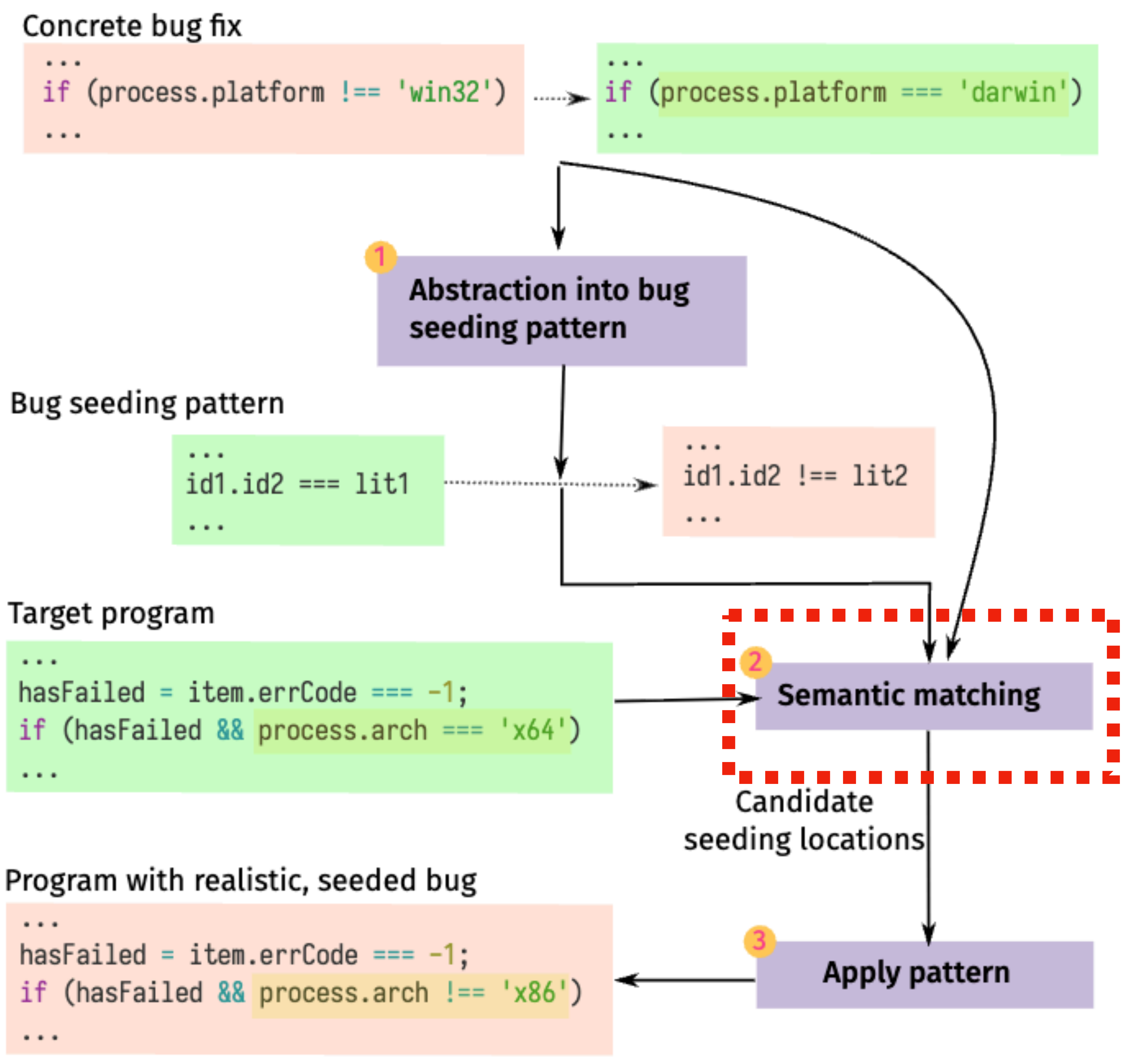


Figure 1: Overview of the approach and running example.

Table 2: Examples of reproduced real-world bugs.

Correct code	Buggy code
Bug to imitate: Commit b776e2b7 of jQuery	
`var opt = speed &&` `typeof speed === "object"`	`var opt =` `typeof speed === "object"`
Seeded bug: Commit b94532c2 of Chart.js	
`if ( style &&` `typeof style === 'object') {`	`if (typeof style === 'object') {`

“Did someone say semantic just now?”

LLM based semantic mutation 😎

- Obviously, we can prompt an LLM to inject errors into a given code snippet. Given how they are trained, it is even likely that the injected error is semantically realistic!
  - Interpretability: the changes are not constrained in size, and may not be repeatable (as we lack permanent mutation operator definitions)
  - Cost: from the traditional mutation testing perspective, LLM-based mutation is extremely expensive. Will quality overcome the quantity?
  - Equivalent mutation problem remains a concern.

Summary

- Mutation testing aims to inject artificial bugs into your software, so that you can check whether your testing is okay.
- The idea that random changes to code can be useful is a very powerful one (eventually leading to Automated Program Repair).
- Traditionally, mutation testing is syntactic. Machine learning enabled semantic mutations. Whether LLMs will take us even further remains to be seen.