Lecture 11 – Testing Oracles

AAA705: Software Testing and Quality Assurance

Jihyeok Park

2024 Spring

Recall

- Delta Debugging (DD)
  - Recursive Delta Debugging – `ddmin`
  - Hierarchical Delta Debugging
  - Probabilistic Delta Debugging (ProbDD)
  - Delta Debugging for Program Debloating
- Information Retrieval based Fault Localization (IRFL)
  - Vector Space Model (VSM)
  - Tf-Idf
  - Cleansing Bug Reports and Source Code
  - VSM and Similarity
- Spectrum-based Fault Localization (SBFL)
  - Genetic Algorithm for SBFL
  - Theoretical Analysis
  - Method-level Aggregation
  - Hybrid SBFL

Contents

Contents

History of π

How to calculate the value of π?

History of π

How to calculate the value of π?

- BC 2000, Babylonia – Just approximate it to $3 + 1/8 = 3.125$

History of π

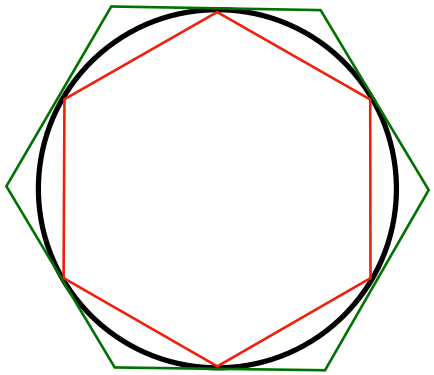
How to calculate the value of π?

- BC 2000, Babylonia – Just approximate it to $3 + 1/8 = 3.125$
- BC 250, Archimedes – First calculation of π using polygons:

History of π

How to calculate the value of π?

- BC 2000, Babylonia – Just approximate it to $3 + 1/8 = 3.125$
- BC 250, Archimedes – First calculation of π using polygons:

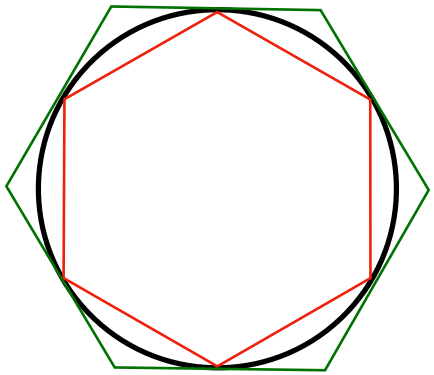


$$6 \cdot \frac{1}{2} = \boxed{3 < \pi < 3.47} \approx 6\left(\frac{1}{\sqrt{3}}\right)$$

History of π

How to calculate the value of π?

- BC 2000, Babylonia – Just approximate it to $3 + 1/8 = 3.125$
- BC 250, Archimedes – First calculation of π using polygons:



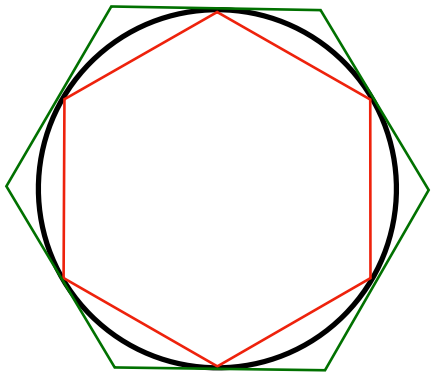
$$6 \cdot \frac{1}{2} = \boxed{3 < \pi < 3.47} \approx 6\left(\frac{1}{\sqrt{3}}\right)$$

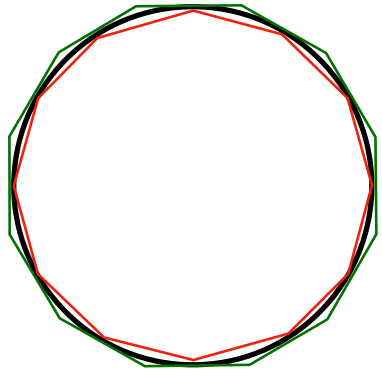
$$\sin(\frac{\alpha}{2}) = \sqrt{\frac{1-\cos\alpha}{2}} \qquad \cos(\frac{\alpha}{2}) = \sqrt{\frac{1+\cos\alpha}{2}} \qquad \tan(\frac{\alpha}{2}) = \sqrt{\frac{1-\cos\alpha}{1+\cos\alpha}}$$

History of π

How to calculate the value of π?

- BC 2000, Babylonia – Just approximate it to $3 + 1/8 = 3.125$
- BC 250, Archimedes – First calculation of π using polygons:





$$6 \cdot \frac{1}{2} = \boxed{3 < \pi < 3.47} \approx 6\left(\frac{1}{\sqrt{3}}\right) \qquad 12\left(\frac{\sqrt{2-\sqrt{3}}}{2}\right) \approx \boxed{3.10 < \pi < 3.22} \approx 12\left(\frac{\sqrt{2-\sqrt{3}}}{\sqrt{2+\sqrt{3}}}\right)$$

$$\sin(\frac{\alpha}{2}) = \sqrt{\frac{1-\cos\alpha}{2}} \qquad \cos(\frac{\alpha}{2}) = \sqrt{\frac{1+\cos\alpha}{2}} \qquad \tan(\frac{\alpha}{2}) = \sqrt{\frac{1-\cos\alpha}{1+\cos\alpha}}$$

History of π

PLRG

- BC 250, Archimedes – First calculation of π using 96-gon:

$$223/71 = \mathbf{3.14}0845 < \pi < \mathbf{3.14}2857 = 22/7 \qquad \textbf{(2 digits)}$$

n	$c_n = n \sin\left(\frac{\pi}{n}\right)$	Approx.	$C_n = n \tan\left(\frac{\pi}{n}\right)$	Approx.
6	3	3	$2\sqrt{3}$	3.46410161
12	$12 \cdot \frac{\sqrt{2-\sqrt{3}}}{2}$	3.10582854	$12 \cdot \frac{\sqrt{2-\sqrt{3}}}{\sqrt{2+\sqrt{3}}}$	3.21539031
24	$24 \cdot \frac{\sqrt{2-\sqrt{2+\sqrt{3}}}}{2}$	3.13262861	$24 \cdot \frac{\sqrt{2-\sqrt{2+\sqrt{3}}}}{\sqrt{2+\sqrt{2+\sqrt{3}}}}$	3.15965994
48	$48 \cdot \frac{\sqrt{2-\sqrt{2+\sqrt{2+\sqrt{3}}}}}{2}$	3.13935020	$48 \cdot \frac{\sqrt{2-\sqrt{2+\sqrt{2+\sqrt{3}}}}}{\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{3}}}}}$	3.14608622
96	$96 \cdot \frac{\sqrt{2-\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{3}}}}}}{2}$	3.14103195	$96 \cdot \frac{\sqrt{2-\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{3}}}}}}{\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{3}}}}}}$	3.14271460

History of π

- BC 250, Archimedes – First calculation of π using 96-gon:

$$223/71 = \mathbf{3.14}0845 < \pi < \mathbf{3.14}2857 = 22/7 \qquad (\mathbf{2\ digits})$$

- AD 263, Liu Hui – First calculation of π using **areas** of 96-gon:

$$\mathbf{3.141}024 < \pi < \mathbf{3.142}074 \qquad (\mathbf{3\ digits})$$

History of π

- BC 250, Archimedes – First calculation of π using 96-gon:

$$223/71 = \mathbf{3.14}0845 < \pi < \mathbf{3.14}2857 = 22/7 \qquad (\mathbf{2\ digits})$$

- AD 263, Liu Hui – First calculation of π using **areas** of 96-gon:

$$\mathbf{3.141}024 < \pi < \mathbf{3.142}074 \qquad (\mathbf{3\ digits})$$

- AD 480, Zu Chongzhi – Liu Hui's method with 12288-gon:

$$\mathbf{3.1415926} < \pi < \mathbf{3.1415927} \qquad (\mathbf{7\ digits})$$

History of π

- BC 250, Archimedes – First calculation of π using 96-gon:

$$223/71 = \mathbf{3.14}0845 < \pi < \mathbf{3.14}2857 = 22/7 \qquad \textbf{(2 digits)}$$

- AD 263, Liu Hui – First calculation of π using **areas** of 96-gon:

$$\mathbf{3.141}024 < \pi < \mathbf{3.142}074 \qquad \textbf{(3 digits)}$$

- AD 480, Zu Chongzhi – Liu Hui's method with 12288-gon:

$$\mathbf{3.1415926} < \pi < \mathbf{3.1415927} \qquad \textbf{(7 digits)}$$

- AD 1400, Madhava of Sangamagrama – First calculation of π using infinite series:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} \qquad \textbf{(10 digits)}$$

History of π

- AD 1596, Ludolph van Ceulen – Calculation of π using 2^{62}-gon:

  35 digits (by spending 25 years)

History of π

- AD 1596, Ludolph van Ceulen – Calculation of π using 2^{62}-gon:

  35 digits (by spending 25 years)

- AD 1706, William Jones – First use of the **Greek letter** π.

History of π

- AD 1596, Ludolph van Ceulen – Calculation of π using 2^{62}-gon:

  35 digits (by spending 25 years)

- AD 1706, William Jones – First use of the **Greek letter** π.
- AD 1775, Euler – **Popularized** the use of π by using it in his book.

History of π

- AD 1596, Ludolph van Ceulen – Calculation of π using 2^{62}-gon:

  35 digits (by spending 25 years)

- AD 1706, William Jones – First use of the **Greek letter** π.
- AD 1775, Euler – **Popularized** the use of π by using it in his book.
- AD 1910, Srinivasa Ramanujan – **Rapidly converging** infinite series:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

History of π

PLRG

- AD 1596, Ludolph van Ceulen – Calculation of π using 2^{62}-gon:

  35 digits (by spending 25 years)

- AD 1706, William Jones – First use of the **Greek letter** π.
- AD 1775, Euler – **Popularized** the use of π by using it in his book.
- AD 1910, Srinivasa Ramanujan – **Rapidly converging** infinite series:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

- AD 1995, Bailey, Borwein, and Plouffe (**BBP**) Formula is discovered:

$$\pi = \sum_{k=0}^{\infty} \left[\frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right) \right]$$

History of π

PLRG

- AD 1596, Ludolph van Ceulen – Calculation of π using 2^{62}-gon:

  35 digits (by spending 25 years)

- AD 1706, William Jones – First use of the **Greek letter** π.
- AD 1775, Euler – **Popularized** the use of π by using it in his book.
- AD 1910, Srinivasa Ramanujan – **Rapidly converging** infinite series:

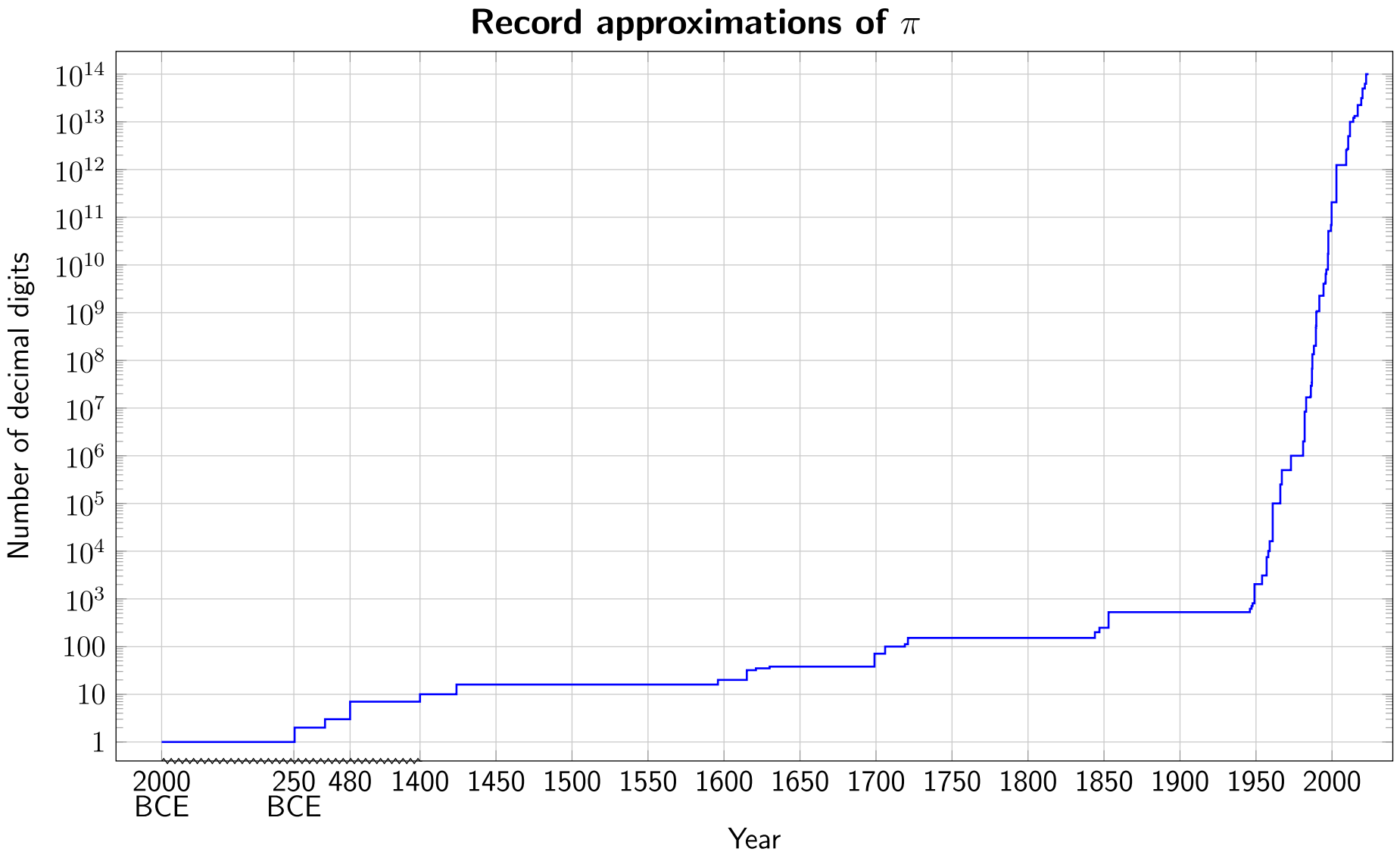
$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

- AD 1995, Bailey, Borwein, and Plouffe (**BBP**) Formula is discovered:

$$\pi = \sum_{k=0}^{\infty} \left[\frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right) \right]$$

- AD 1997, **Fabrice Bellard's Formula** (43% faster than BBP).

History of π



Computing π

- How to write a **program** that computes π?

Computing π

- How to write a **program** that computes π?
- Then, how to **test** the program that actually computes π?

Computing π

PLRG

- How to write a **program** that computes π?
- Then, how to **test** the program that actually computes π?
- Some programs do not have a **test oracle** to compare the output.

Computing π

PLRG

- How to write a **program** that computes π?
- Then, how to **test** the program that actually computes π?
- Some programs do not have a **test oracle** to compare the output.
- We call such programs as **non-testable programs**.

Computing π

PLRG

- How to write a **program** that computes π?
- Then, how to **test** the program that actually computes π?
- Some programs do not have a **test oracle** to compare the output.
- We call such programs as **non-testable programs**.
- Then, how to **test** such non-testable programs?

Types of Non-Testable Programs

- **Type 1**: Programs was written to determine the answer to a problem we have not yet solved. (If we knew the oracle, we would not need the program.)

Types of Non-Testable Programs

PLRG

- **Type 1**: Programs was written to determine the answer to a problem we have not yet solved. (If we knew the oracle, we would not need the program.)

- **Type 2**: Programs produce so much output that it is impractical to verify all of it.

Types of Non-Testable Programs

PLRG

- **Type 1**: Programs was written to determine the answer to a problem we have not yet solved. (If we knew the oracle, we would not need the program.)
- **Type 2**: Programs produce so much output that it is impractical to verify all of it.
- **Type 3**: Programs for which the tester has a **misconception**.

Types of Non-Testable Programs

From our point of view, type 1 programs are the most interesting.

- **Type 1**: Most scientific computation, certain branches of Artificial Intelligence or Machine Learning, etc.

Types of Non-Testable Programs

From our point of view, type 1 programs are the most interesting.

- **Type 1**: Most scientific computation, certain branches of Artificial Intelligence or Machine Learning, etc.
  - What is the **correct** value of π?

Types of Non-Testable Programs

From our point of view, type 1 programs are the most interesting.

- **Type 1**: Most scientific computation, certain branches of Artificial Intelligence or Machine Learning, etc.
  - What is the **correct** value of π?
  - What is the **correct** result of the sin, cos, and tan functions?

Types of Non-Testable Programs

From our point of view, type 1 programs are the most interesting.

- **Type 1**: Most scientific computation, certain branches of Artificial Intelligence or Machine Learning, etc.
  - What is the **correct** value of π?
  - What is the **correct** result of the sin, cos, and tan functions?
  - What is the **correct** way to play a video game, if you are applying reinforcement learning?

Types of Non-Testable Programs

From our point of view, type 1 programs are the most interesting.

- **Type 1**: Most scientific computation, certain branches of Artificial Intelligence or Machine Learning, etc.
  - What is the **correct** value of π?
  - What is the **correct** result of the sin, cos, and tan functions?
  - What is the **correct** way to play a video game, if you are applying reinforcement learning?
  - What is the **correct** way to classify an image, if you are applying deep learning?

Types of Non-Testable Programs

From our point of view, type 1 programs are the most interesting.

- **Type 1**: Most scientific computation, certain branches of Artificial Intelligence or Machine Learning, etc.
  - What is the **correct** value of π?
  - What is the **correct** result of the sin, cos, and tan functions?
  - What is the **correct** way to play a video game, if you are applying reinforcement learning?
  - What is the **correct** way to classify an image, if you are applying deep learning?
  - What is the **correct** way to translate a sentence, if you are applying machine translation?

Implementing sin

- Let's assume that you implemented the sin function.

Implementing sin

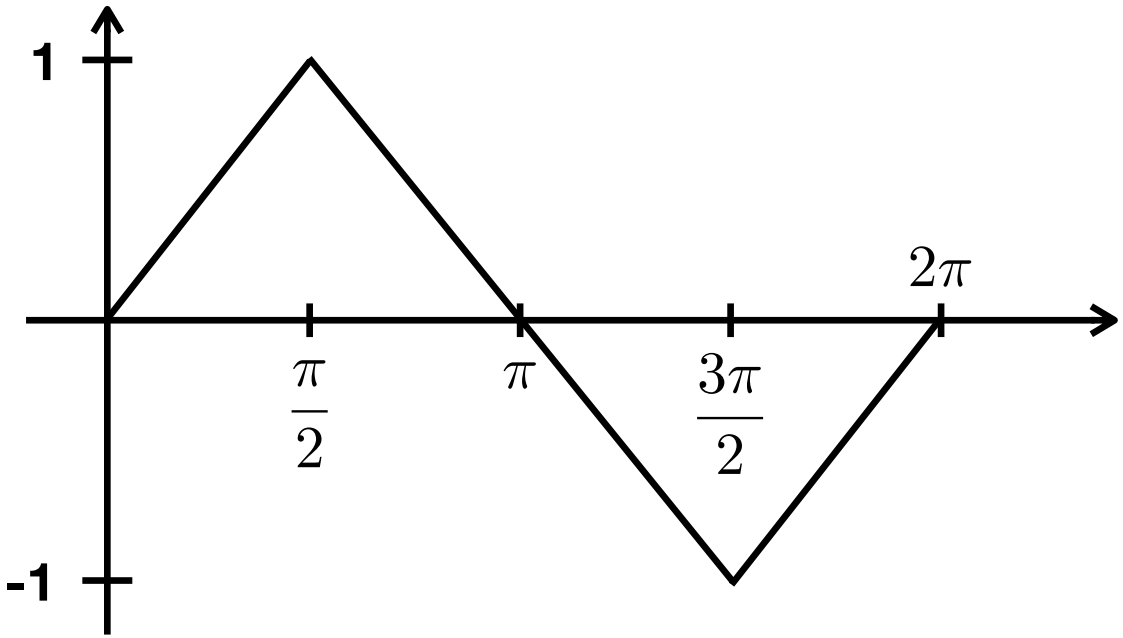
- Let's assume that you implemented the sin function.
- How to test the sin function?

Implementing sin

- Let's assume that you implemented the sin function.
- How to test the sin function?
- We can test it using the known values of the sin function.
  - $\sin(0) = 0$
  - $\sin(\pi/2) = 1$
  - $\sin(\pi) = 0$
  - $\sin(3\pi/2) = -1$
  - $\sin(2\pi) = 0$

Implementing sin

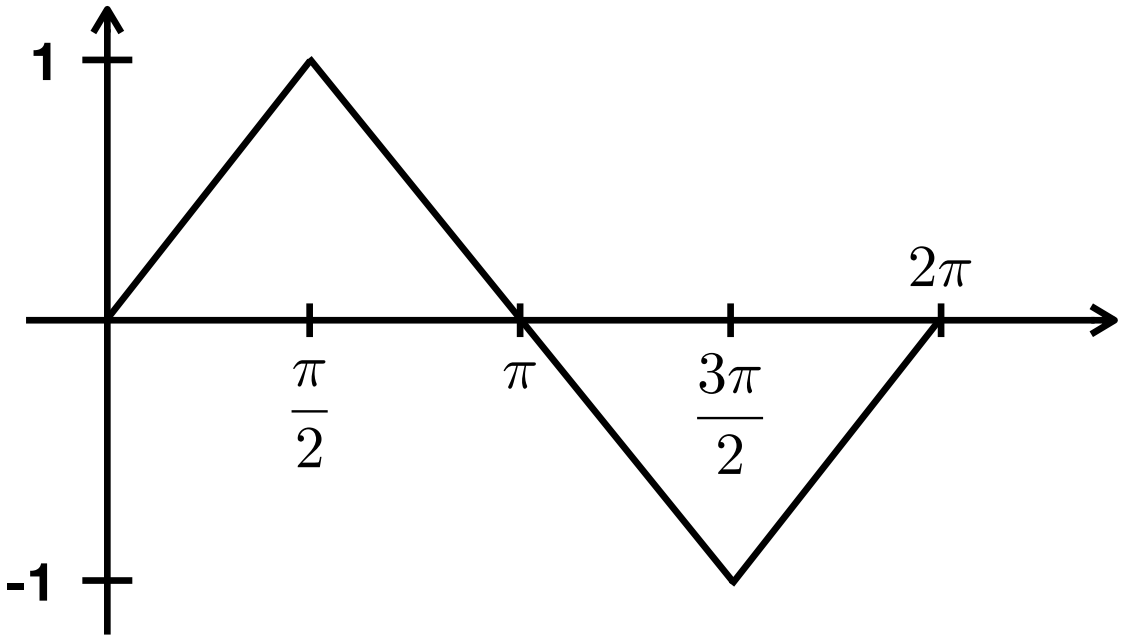
However, a wrong implementation of the sin function can pass all of them:



Implementing sin

PLRG

However, a wrong implementation of the sin function can pass all of them:

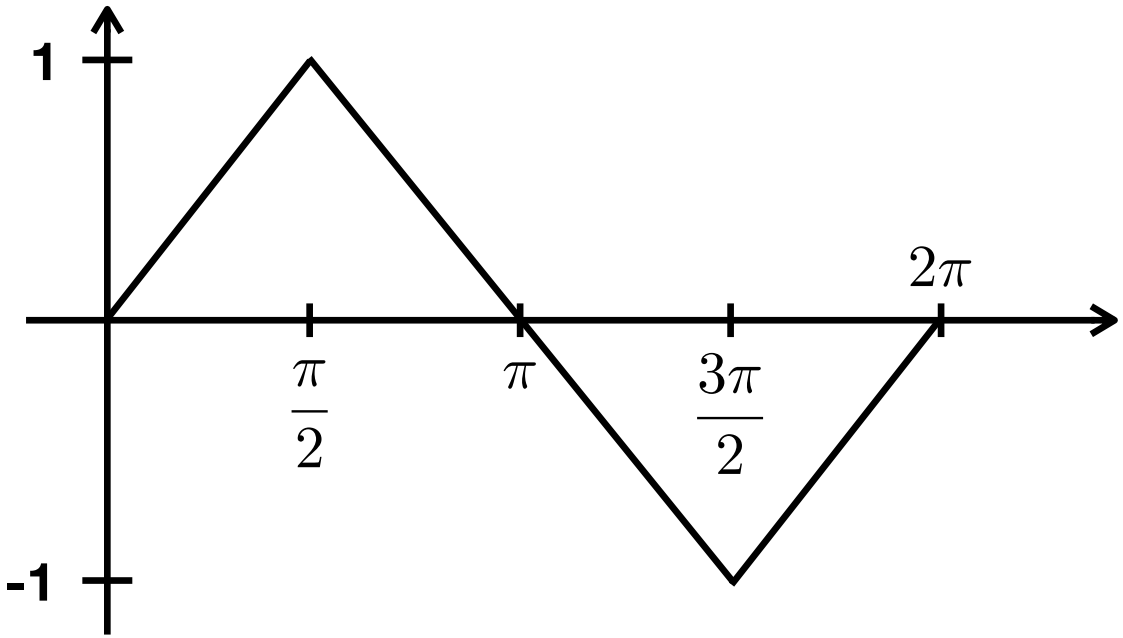


Let's utilize a domain knowledge to test the sin function:

Implementing sin

PLRG

However, a wrong implementation of the sin function can pass all of them:



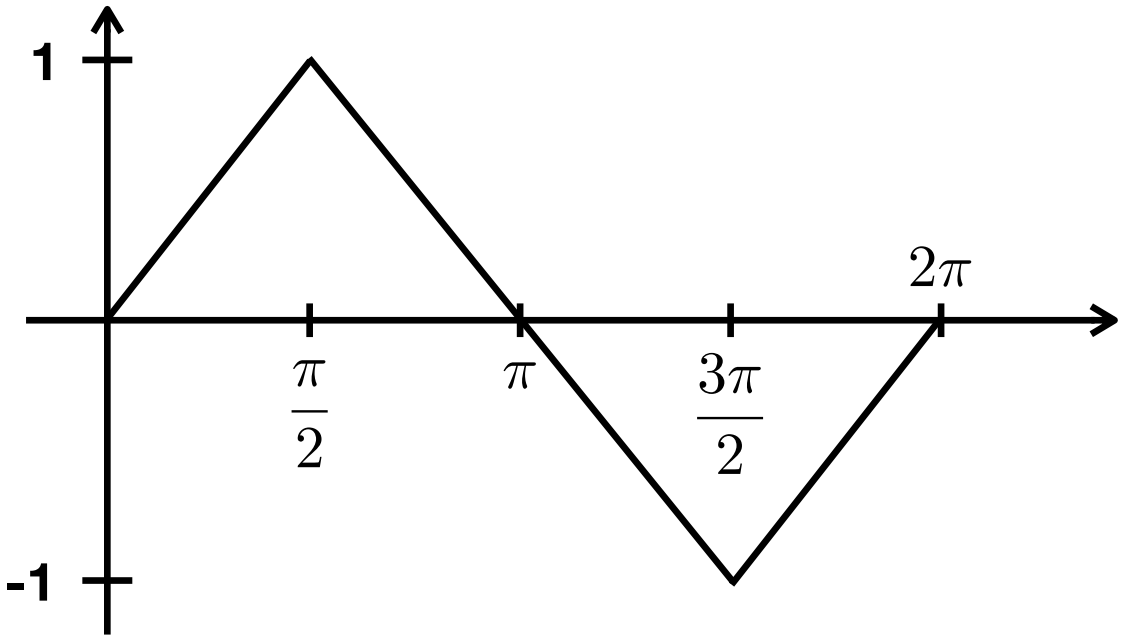
Let's utilize a domain knowledge to test the sin function:

- $\sin(x) = \sin(\pi - x)$

Implementing sin

PLRG

However, a wrong implementation of the sin function can pass all of them:



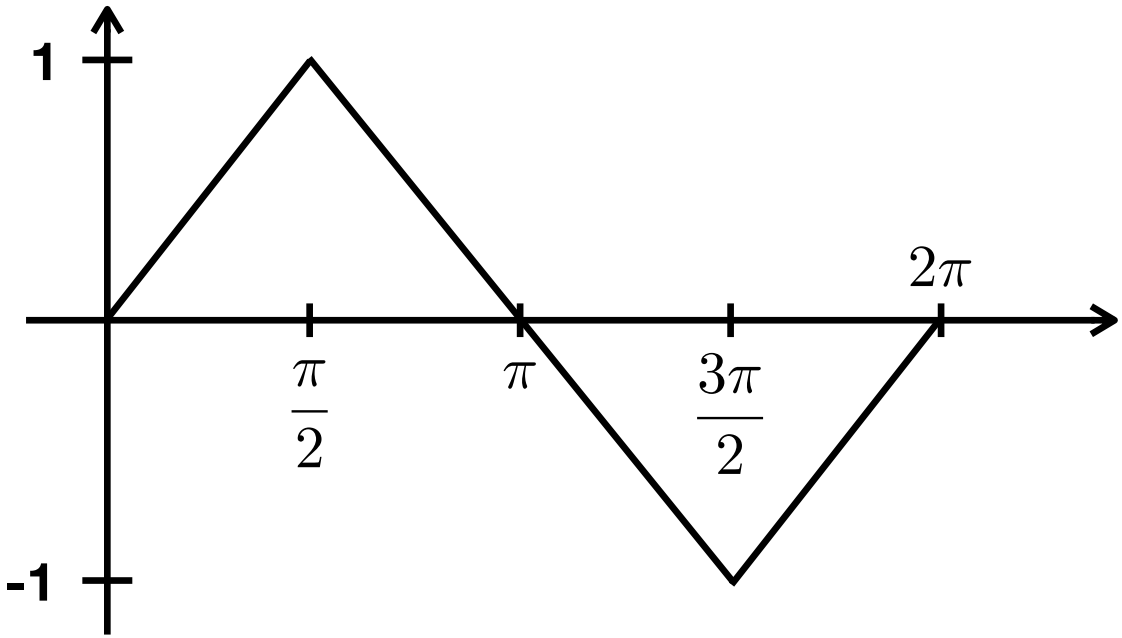
Let's utilize a domain knowledge to test the sin function:

- $\sin(x) = \sin(\pi - x)$
- $\sin(x) = -\sin(x + \pi)$

Implementing sin

PLRG

However, a wrong implementation of the sin function can pass all of them:



Let's utilize a domain knowledge to test the sin function:

- $\sin(x) = \sin(\pi - x)$
- $\sin(x) = -\sin(x + \pi)$
- $\sin^2(x) + \sin^2(x + \frac{\pi}{2}) = 1$

Contents

Metamorphic Relationship

PLRG

Definition (Metamorphic Relationship)

A program $p : X \to Y$ has a **metamorphic relationship** $f : X \to X$ with a relation $R \subseteq Y \times Y$ if and only if:

$$\forall x \in X.\ (p(x), p \circ f(x)) \in R$$

Metamorphic Relationship

PLRG

Definition (Metamorphic Relationship)

A program $p : X \to Y$ has a **metamorphic relationship** $f : X \to X$ with a relation $R \subseteq Y \times Y$ if and only if:

$$\forall x \in X.\ (p(x), p \circ f(x)) \in R$$

For example, the sin function has the following metamorphic relationships:

f	**Relationship R**
$f(x) = \pi - x$	$\sin(x) = \sin(\pi - x)$
$f(x) = x + \pi$	$\sin(x) = -\sin(x + \pi)$
$f(x) = x + \frac{\pi}{2}$	$\sin^2(x) + \sin^2(x + \frac{\pi}{2}) = 1$

Metamorphic Testing

- **Metamorphic Testing** is a testing technique that is based on the metamorphic relationships of the program.

Metamorphic Testing

PLRG

- **Metamorphic Testing** is a testing technique that is based on the metamorphic relationships of the program.

- However, manual identification of metamorphic relationships is **labor intensive** and **error-prone**.

Metamorphic Testing

PLRG

- **Metamorphic Testing** is a testing technique that is based on the metamorphic relationships of the program.
- However, manual identification of metamorphic relationships is **labor intensive** and **error-prone**.
- Can we **automatically learn** metamorphic relationships?

Examples: SMT Solvers

D. Winterer, C. Zhang, and Z. Su. Validating SMT Solvers via Semantic Fusion, PLDI 2020.

$$\varphi_1 = x > 0 \wedge \boxed{x} > 1$$
$$\varphi_2 = \boxed{y} < 0 \wedge y < 1$$
$$\varphi_{concat} = (x > 0 \wedge \boxed{x} > 1) \wedge (\boxed{y} < 0 \wedge y < 1)$$
$$\varphi_{fused} = (x > 0 \wedge \boxed{z - y} > 1) \wedge (\boxed{z - x} < 0 \wedge y < 1)$$

1. **Formula Concatenation** – Concatenate two formulas ϕ_1 and ϕ_2 using **conjunction** ($\wedge$) or **disjunction** ($\vee$).
2. **Variable Fusion** – Create fresh variables to connect the variable sets of ϕ_1 and ϕ_2 using **fusion functions**.
3. **Variable Inversion** – Substitute some occurrences of the chosen variables in ϕ_1 and ϕ_2 using **inversion functions**.

Examples: SMT Solvers

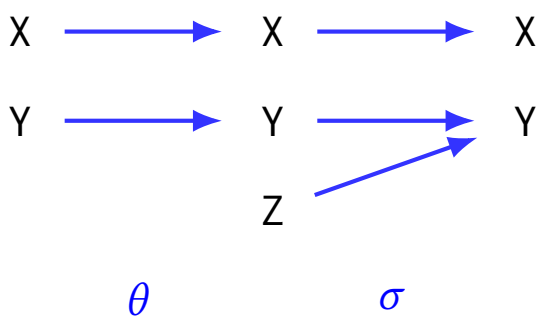
D. Winterer, C. Zhang, and Z. Su. Validating SMT Solvers via Semantic Fusion, PLDI 2020.

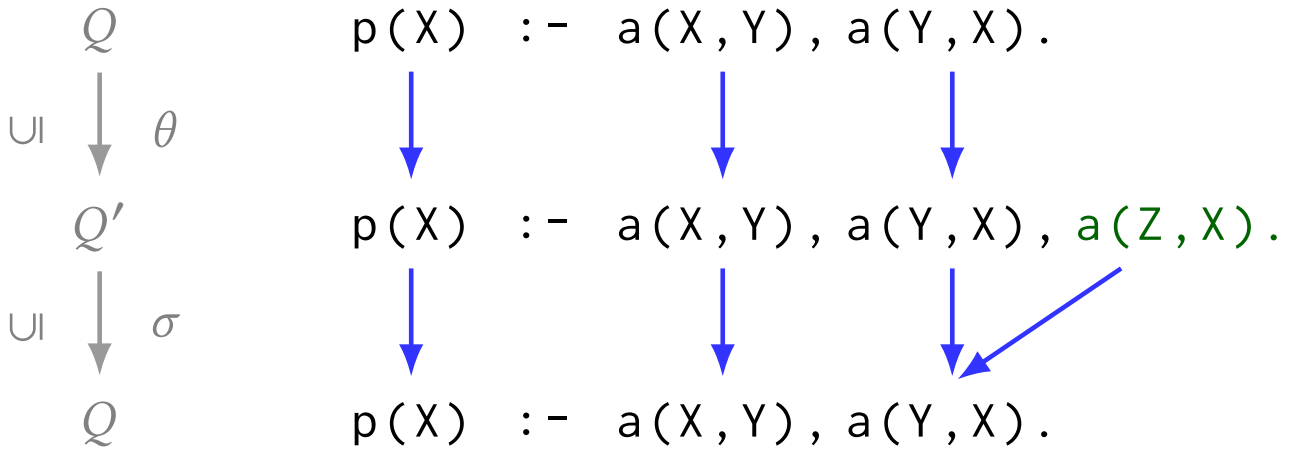
Type	**Fusion Function**	**Variable Inversion Functions**	
		r_x	r_y
Int	$x + y$	$z - y$	$z - x$
	$x + c + y$	$z - c - y$	$z - c - x$
	$x * y$	$z \; div \; y$	$z \; div \; x$
	$c_1 * x + c_2 * y + c_3$	$(z - c_2 * y - c_3) \; div \; c_1$	$(z - c_1 * x - c_3) \; div \; c_2$
Real	$x + y$	$z - y$	$z - x$
	$x + c + y$	$z - c - y$	$z - c - x$
	$x * y$	z/y	z/x
	$c_1 * x + c_2 * y + c_3$	$(z - c_2 * y - c_3)/c_1$	$(z - c_1 * x - c_3)/c_2$
String	$x \; str{+}{+} \; y$	$str.substr \;\; z \; 0 \; (str.len \; x)$	$str.substr \;\; z \; (str.len \; x) \; (str.len \; y)$
	$x \; str{+}{+} \; y$	$str.substr \;\; z \; 0 \; (str.len \; x)$	$str.replace \; z \; x \; ""$
	$x \; str{+}{+} \; c \; str{+}{+} \; y$	$str.substr \;\; z \; 0 \; (str.len \; x)$	$str.replace \; (str.replace \; z \; x \; "") \; c \; ""$

Above **fusion functions** and **inversion functions** are used to generate the **metamorphic transformations** to test the **SMT solvers**.

Examples: Datalog Engines

M. N. Mansur, M. Christakis, and V. Wüstholz. Metamorphic Testing of Datalog Engines, ESEC/FSE 2021.





Metamorphic transformation for Datalog engines using **containment mapping** (e.g., AddEqu – addition (Add) for equivalence (Equ)).

Examples: Web Browser Debugger

S. Tolksdorf, D. Lehmann, and M. Pradel. Interactive Metamorphic Testing of Debuggers, ISSTA 2019.

It defines the **metamorphic transformations** for the 1) **debugger actions** and 2) **input programs** to test the **web browser debugger**.

```
1  // Original input:
2  var a = 5;       //  (i) pauses --> continue
3  var slideOverMe;
4  var C = class{};// (ii) pauses --> continue
5  var b = 42;      //(iii) pauses --> continue
```

```
1  // Transformed input:
2  var a = 5;       //  (i) pauses --> continue
3     var slideOverMe;
4  var C = class{};//       (no pausing)
5  var b = 42;      // (ii) pauses
```

In the upper case, a breakpoint is originally set at **line 3**, but debugger **slides** it to **line 4**. In the lower case, a breakpoint is **directly** set at **line 4**.

Examples: Compilers

V. Le, M. Afshari, and Z. Su. Compiler Validation via Equivalence Modulo Inputs, PLDI 2014.

> Two programs $P, Q \in \mathscr{L}$ are *equivalent modulo inputs (EMI) w.r.t.* an input set I common to P and Q (*i.e.*, $I \subseteq \mathsf{dom}(P) \cap \mathsf{dom}(Q)$) iff
>
> $$\forall i \in I \;\; [\![P]\!](i) = [\![Q]\!](i).$$

Examples: Compilers

V. Le, M. Afshari, and Z. Su. Compiler Validation via Equivalence Modulo Inputs, PLDI 2014.

> Two programs $P, Q \in \mathscr{L}$ are *equivalent modulo inputs (EMI) w.r.t.* an input set I common to P and Q (*i.e.*, $I \subseteq \mathsf{dom}(P) \cap \mathsf{dom}(Q)$) iff
>
> $$\forall i \in I \;\; [\![P]\!](i) = [\![Q]\!](i).$$

It measures the **code coverage** for given inputs and generates **variants** of the program by randomly **pruning** the **unexecuted statements**.

```
def prune_visit(prog, statement, coverage_set):
  if statement not in coverage_set and flip_coin(statement):
    prog.delete(statement)
  else for child in statement.children:
      prune_visit(prog, child, coverage_set)
def gen_variant(prog, coverage_set):
  emi_variant = clone(prog)
  for statement in emi_variant:
    prune_visit(emi_variant, statement, coverage_set)
  return emi_variant
```

Examples: Compilers

```
struct tiny { char c; char d; char e; };
f(int n, struct tiny x, struct tiny y, struct tiny z, long l) {
  if (x.c != 10) abort(); if (x.d != 20) abort(); if (x.e != 30) abort();
  if (y.c != 11) abort(); if (y.d != 21) abort(); if (y.e != 31) abort();
  if (z.c != 12) abort(); if (z.d != 22) abort(); if (z.e != 32) abort();
  if (l != 123) abort(); }
main() {
  struct tiny x[3]; x[0].c = 10; x[1].c = 11; x[2].c = 12; x[0].d = 20;
  x[1].d = 21; x[2].d = 22; x[0].e = 30; x[1].e = 31; x[2].e = 32;
  f(3, x[0], x[1], x[2], (long)123); exit(0); }
```

```
struct tiny { char c; char d; char e; };
f(int n, struct tiny x, struct tiny y, struct tiny z, long l) {
  if (x.c != 10) /* X */; if (x.d != 20) abort(); if (x.e != 30) /* X */;
  if (y.c != 11) abort(); if (y.d != 21) abort(); if (y.e != 31) /* X */;
  if (z.c != 12) abort(); if (z.d != 22) /* X */; if (z.e != 32) abort();
  if (l != 123) /* X */; }
main() {
  struct tiny x[3]; x[0].c = 10; x[1].c = 11; x[2].c = 12; x[0].d = 20;
  x[1].d = 21; x[2].d = 22; x[0].e = 30; x[1].e = 31; x[2].e = 32;
  f(3, x[0], x[1], x[2], (long)123); exit(0); }
```

Examples: Compilers

C. Sun, V. Le, and Z. Su. Finding Compiler Bugs via Live Code Mutation, OOPSLA 2016.

> Two programs $P, Q \in \mathscr{L}$ are *equivalent modulo inputs (EMI) w.r.t.* an input set I common to P and Q (*i.e.*, $I \subseteq \mathsf{dom}(P) \cap \mathsf{dom}(Q)$) iff
>
> $$\forall i \in I \;\; [\![P]\!](i) = [\![Q]\!](i).$$

Examples: Compilers

C. Sun, V. Le, and Z. Su. Finding Compiler Bugs via Live Code Mutation, OOPSLA 2016.

> Two programs $P, Q \in \mathscr{L}$ are *equivalent modulo inputs (EMI) w.r.t.* an input set I common to P and Q (*i.e.*, $I \subseteq \mathsf{dom}(P) \cap \mathsf{dom}(Q)$) iff
>
> $$\forall i \in I \;\; [\![P]\!](i) = [\![Q]\!](i).$$

It defines **EMI mutation operators** not only for dead code but also for **live code** by **profiling** the **valuations** of **live variables**.

Examples: Compilers

C. Sun, V. Le, and Z. Su. Finding Compiler Bugs via Live Code Mutation, OOPSLA 2016.

> Two programs $P, Q \in \mathscr{L}$ are *equivalent modulo inputs (EMI) w.r.t.* an input set I common to P and Q (*i.e.*, $I \subseteq \mathsf{dom}(P) \cap \mathsf{dom}(Q)$) iff
>
> $$\forall i \in I \;\; [\![P]\!](i) = [\![Q]\!](i).$$

It defines **EMI mutation operators** not only for dead code but also for **live code** by **profiling** the **valuations** of **live variables**.

- **Always False Conditional Block (FCB)** – Generate an `if`/`while` statement, whose body is not empty and condition is always `false`.

Examples: Compilers

C. Sun, V. Le, and Z. Su. Finding Compiler Bugs via Live Code Mutation, OOPSLA 2016.

> Two programs $P, Q \in \mathscr{L}$ are *equivalent modulo inputs (EMI) w.r.t.* an input set I common to P and Q (*i.e.*, $I \subseteq \mathsf{dom}(P) \cap \mathsf{dom}(Q)$) iff
>
> $$\forall i \in I \;\; [\![P]\!](i) = [\![Q]\!](i).$$

It defines **EMI mutation operators** not only for dead code but also for **live code** by **profiling** the **valuations** of **live variables**.

- **Always False Conditional Block (FCB)** – Generate an `if`/`while` statement, whose body is not empty and condition is always `false`.
- **Always True Guard (TG)** – Generate an `if` statement, whose condition is always `true` and guard an existing **executed statement**.

Examples: Compilers

C. Sun, V. Le, and Z. Su. Finding Compiler Bugs via Live Code Mutation, OOPSLA 2016.

> Two programs $P, Q \in \mathscr{L}$ are *equivalent modulo inputs (EMI) w.r.t.* an input set I common to P and Q (*i.e.*, $I \subseteq \mathsf{dom}(P) \cap \mathsf{dom}(Q)$) iff
>
> $$\forall i \in I \;\; [\![P]\!](i) = [\![Q]\!](i).$$

It defines **EMI mutation operators** not only for dead code but also for **live code** by **profiling** the **valuations** of **live variables**.

- **Always False Conditional Block (FCB)** – Generate an `if`/`while` statement, whose body is not empty and condition is always `false`.
- **Always True Guard (TG)** – Generate an `if` statement, whose condition is always `true` and guard an existing **executed statement**.
- **Always True Conditional Block (TCB)** – Synthesize an `if` statement, whose body has **side effects** but **reverts** them at the end.

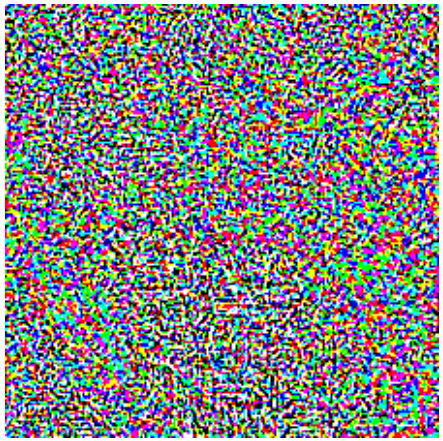
Examples: Deep Neural Networks

I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples, ICLR 2015.

$\boldsymbol{x}$

"panda"
57.7% confidence

$+ .007 \times$



$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$

"nematode"
8.2% confidence

$=$

$\boldsymbol{x} + \epsilon\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$

"gibbon"
99.3 % confidence

Adversarial examples are the inputs that are **designed** to intentionally **mislead** the result of the deep neural networks.

Examples: Deep Neural Networks

X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety verification of deep neural networks, CAV 2017.

automobile to bird automobile to frog automobile to airplane automobile to horse

Fig. 1. Automobile images (classified correctly) and their perturbed images (classified wrongly)

It uses **SMT solvers** to find the **adversarial examples** as **counterexamples** or to **verify** the safety of the deep neural networks.

Metamorphic testing is a surprisingly effective concept for testing deep neural networks (at least so far).

Examples: Object Detection

S. Wang and Z. Su. Metamorphic Object Insertion for Testing Object Detection Systems. ASE 2020.

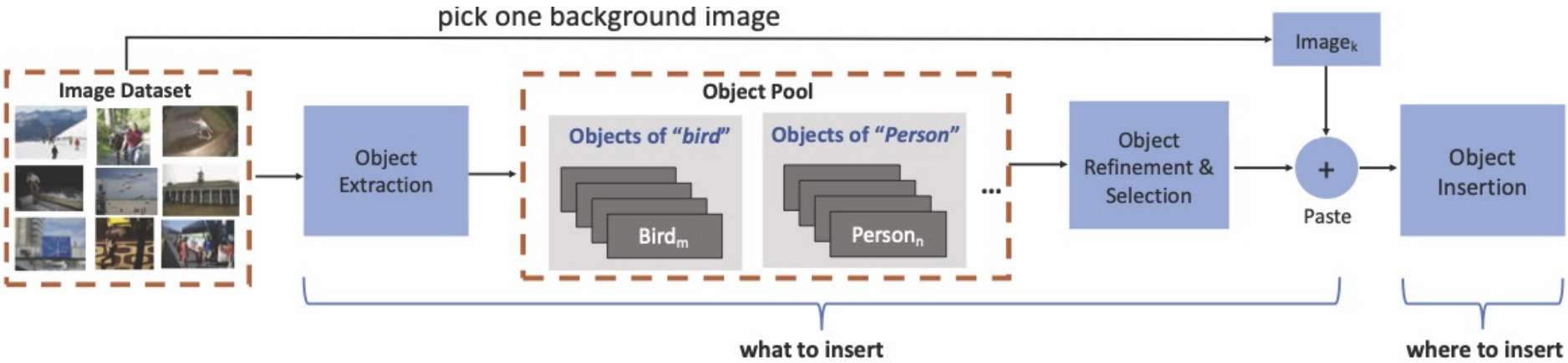
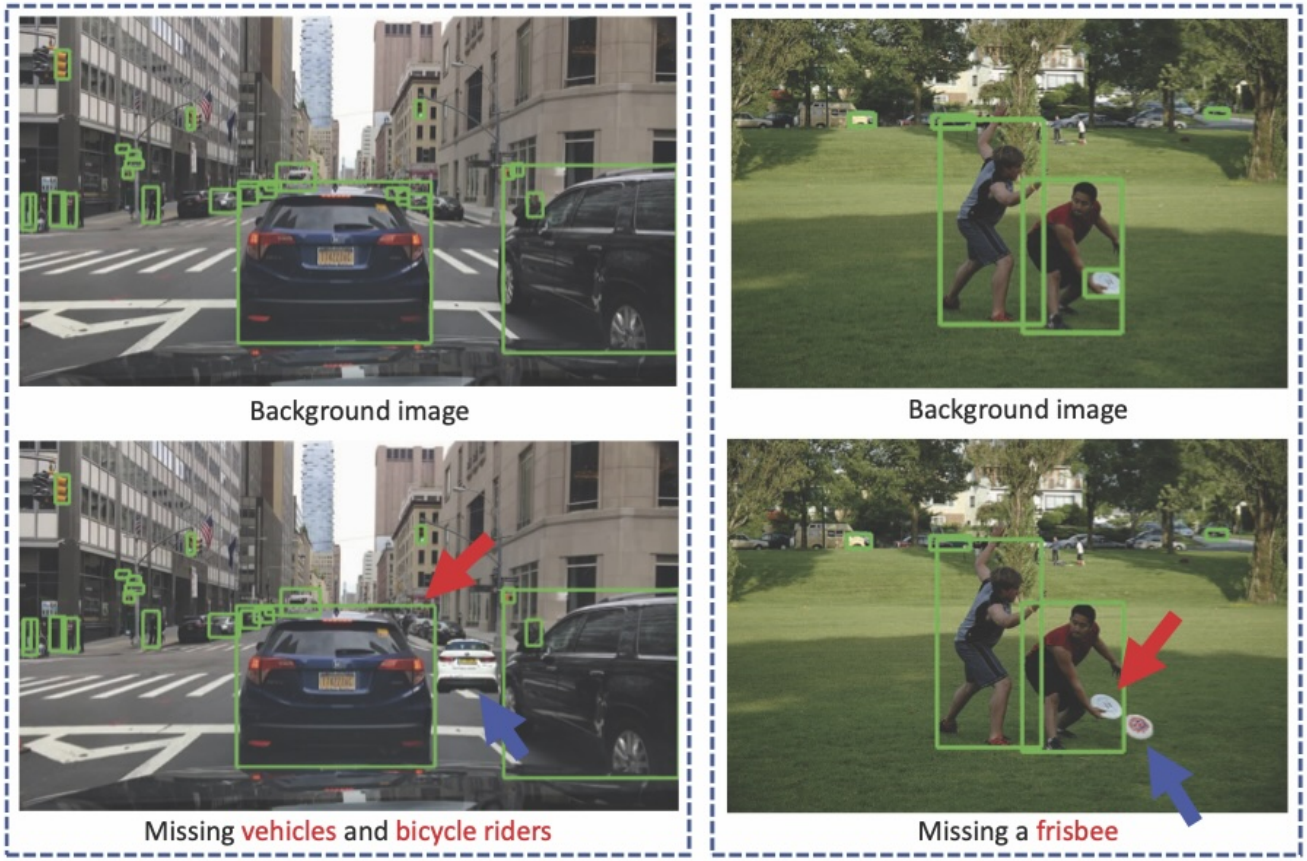


Figure 5: Workflow of MetaOD.

MetaOD utilizes a **metamorphic transformation** that **pastes** objects collected from the original images into the given image.

Examples: Object Detection



The inserted objects are pointed by **blue arrows**, and the inserted objects are resized to the average size of existing objects of the same category.

Learning Metamorphic Relationships

J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei. Search-based inference of polynomial metamorphic relations, ASE 2014.

Learning Metamorphic Relationships

J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei. Search-based inference of polynomial metamorphic relations, ASE 2014.

- Consider a **linear metamorphic relationship** for program $p : X \rightarrow Y$:

$$f(x) = \alpha x + \beta \qquad\qquad c_1 y + c_2 y' + d = 0$$

where $y = p(x)$ and $y' = p(f(x))$.

Learning Metamorphic Relationships

J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei. Search-based inference of polynomial metamorphic relations, ASE 2014.

- Consider a **linear metamorphic relationship** for program $p : X \to Y$:

$$f(x) = \alpha x + \beta \qquad\qquad c_1 y + c_2 y' + d = 0$$

  where $y = p(x)$ and $y' = p(f(x))$.

- For example, $\sin(x) = \sin(\pi - x)$ is a linear metamorphic relationship with $\alpha = -1, \beta = \pi, c_1 = 1, c_2 = -1, d = 0$.

Learning Metamorphic Relationships

J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei. Search-based inference of polynomial metamorphic relations, ASE 2014.

- Consider a **linear metamorphic relationship** for program $p : X \to Y$:

$$f(x) = \alpha x + \beta \qquad\qquad c_1 y + c_2 y' + d = 0$$

  where $y = p(x)$ and $y' = p(f(x))$.

- For example, $\sin(x) = \sin(\pi - x)$ is a linear metamorphic relationship with $\alpha = -1, \beta = \pi, c_1 = 1, c_2 = -1, d = 0$.

- Then, the learning problem is to find the coefficients $\alpha, \beta, c_1, c_2, d$ that satisfy the linear metamorphic relationship.

Learning Metamorphic Relationships

J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei. Search-based inference of polynomial metamorphic relations, ASE 2014.

- Consider a **linear metamorphic relationship** for program $p : X \to Y$:

$$f(x) = \alpha x + \beta \qquad\qquad c_1 y + c_2 y' + d = 0$$

  where $y = p(x)$ and $y' = p(f(x))$.

- For example, $\sin(x) = \sin(\pi - x)$ is a linear metamorphic relationship with $\alpha = -1, \beta = \pi, c_1 = 1, c_2 = -1, d = 0$.

- Then, the learning problem is to find the coefficients $\alpha, \beta, c_1, c_2, d$ that satisfy the linear metamorphic relationship.

- The key idea is a **search-based** approach to find the coefficients using a **particle swarm optimization (PSO)** algorithm.

Recall: Particle Swarm Optimization (PSO)

PLRG

$$x_i^{t+1} = x_i^t + v_i^t$$

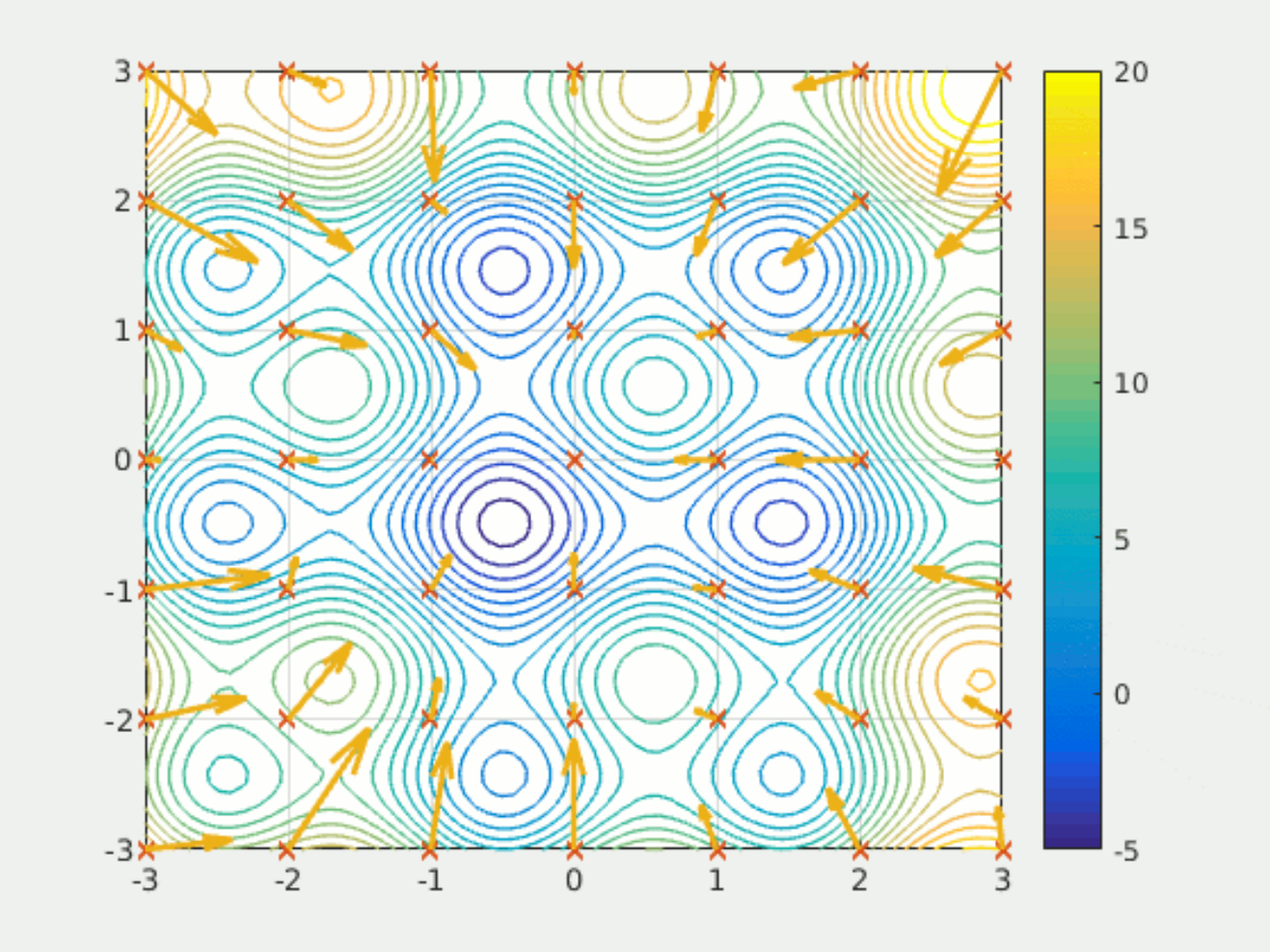
$$v_i^{t+1} = ① \; wv_i^t + ② \; c_1(p_i - x_i^t) + ③ \; c_2(g - x_i^t)$$

- x_i^t – position of the i-th particle at time t
- v_i^t – velocity of the i-th particle at time t
- p_i – best position of the i-th particle (local best)
- g – best position of the entire flock (global best)

It follows the three rules of the flock of birds.

1. Each bird has an inertia to keep flying in the **same direction**.
2. Each bird remembers and has a tendency to return to the **best position** it has ever **visited by itself** (**local best**).
3. Each bird has a tendency to **follow** the known **global best position** in the flock by **communicating** with other birds. (**global best)**

Recall: Particle Swarm Optimization (PSO)



Link

Learning Metamorphic Relationships

J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei. Search-based inference of polynomial metamorphic relations, ASE 2014.

- It defines the **fitness function** for the PSO algorithm as the number of inputs that satisfy the formula with the coefficients with n inputs:

$$\text{fitness} = \sum_{i=1}^{n} (c_1 y_i + c_2 y_i' + d = 0)$$

where x_i is the i-th input and $y_i = p(x_i)$, $y_i' = p(f(x_i))$.

Learning Metamorphic Relationships

J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei. Search-based inference of polynomial metamorphic relations, ASE 2014.

- It defines the **fitness function** for the PSO algorithm as the number of inputs that satisfy the formula with the coefficients with n inputs:

$$\text{fitness} = \sum_{i=1}^{n} (c_1 y_i + c_2 y_i' + d = 0)$$

  where x_i is the i-th input and $y_i = p(x_i)$, $y_i' = p(f(x_i))$.
- When c_1 and c_2 are between $[-\phi, \phi]$ with a **small threshold** ϕ, the algorithm resets them to a new random value to prevent producing the **degenerate solutions**.

Learning Metamorphic Relationships

J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei. Search-based inference of polynomial metamorphic relations, ASE 2014.

- It defines the **fitness function** for the PSO algorithm as the number of inputs that satisfy the formula with the coefficients with n inputs:

$$\text{fitness} = \sum_{i=1}^{n} (c_1 y_i + c_2 y_i' + d = 0)$$

  where x_i is the i-th input and $y_i = p(x_i)$, $y_i' = p(f(x_i))$.
- When c_1 and c_2 are between $[-\phi, \phi]$ with a **small threshold** ϕ, the algorithm resets them to a new random value to prevent producing the **degenerate solutions**.
- We can extend it into a **quadratic metamorphic relationship**:

$$f(x) = \alpha x + \beta \qquad c_1 y^2 + c_2 y y' + c_3 y'^2 + d_1 y + d_2 y' + e = 0$$

  where $y = p(x)$ and $y' = p(f(x))$.

Contents

Differential Testing

- **Differential Testing** is a testing technique that compares the outputs of two or more **similar** programs (or different implementations of same specification) for the **same inputs**.

Differential Testing

- **Differential Testing** is a testing technique that compares the outputs of two or more **similar** programs (or different implementations of same specification) for the **same inputs**.

- Unlike the **metamorphic testing**, differential testing does **not require** the **metamorphic relationships**.

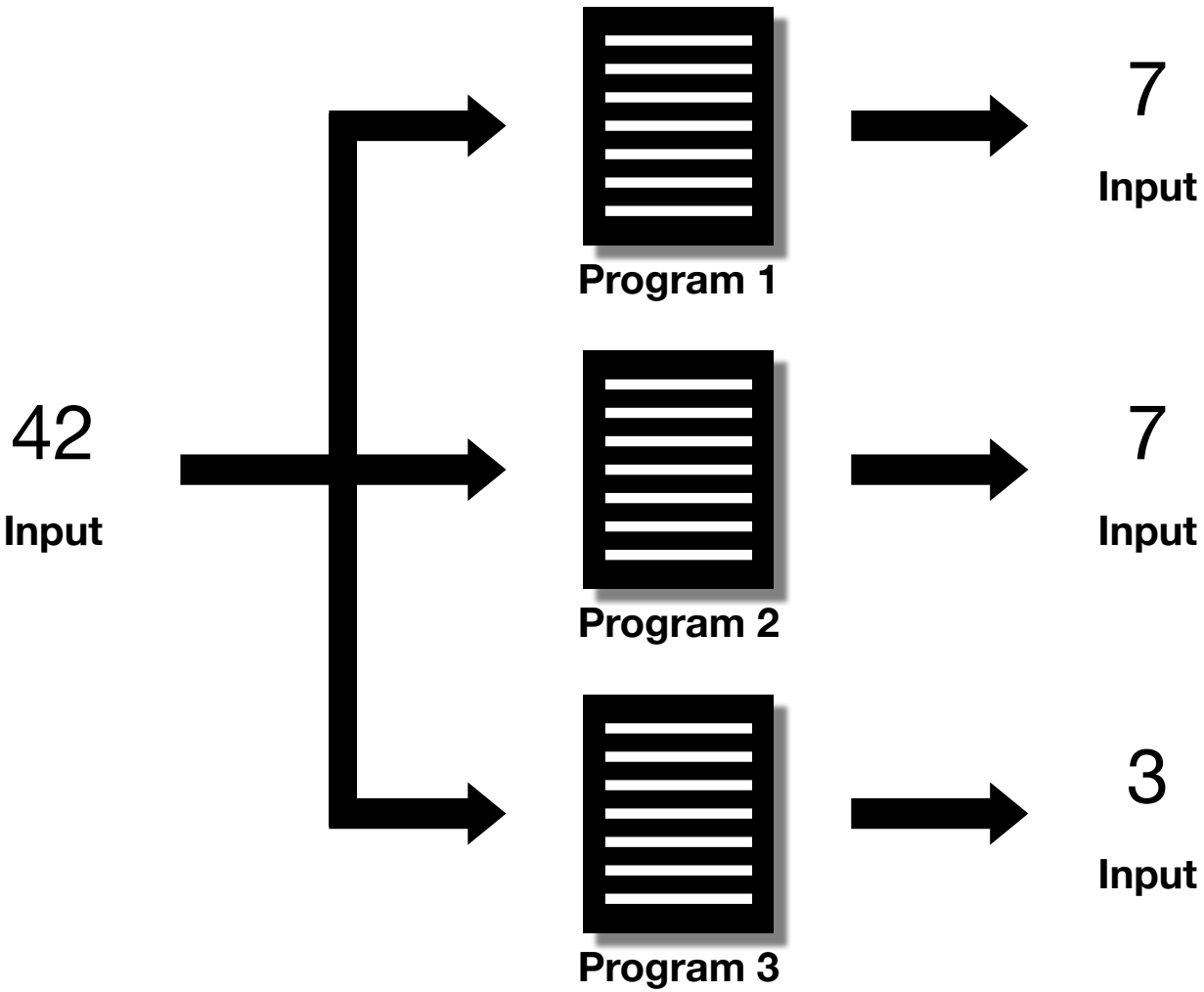
Differential Testing

- **Differential Testing** is a testing technique that compares the outputs of two or more **similar** programs (or different implementations of same specification) for the **same inputs**.
- Unlike the **metamorphic testing**, differential testing does **not require** the **metamorphic relationships**.
- The key idea is to **compare** the outputs of the programs and **report** the **differences** as the **potential bugs**.

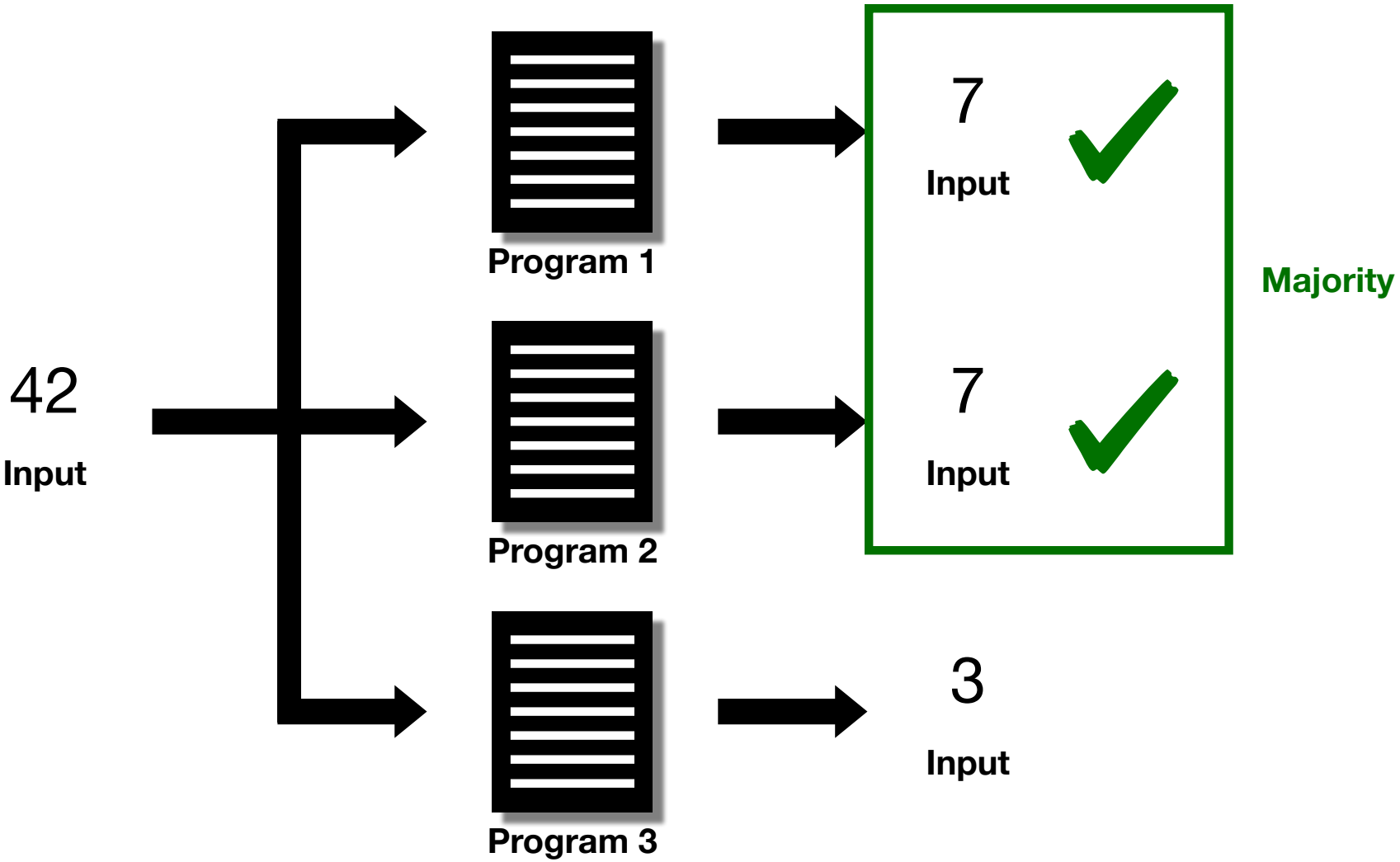
Differential Testing

- **Differential Testing** is a testing technique that compares the outputs of two or more **similar** programs (or different implementations of same specification) for the **same inputs**.
- Unlike the **metamorphic testing**, differential testing does **not require** the **metamorphic relationships**.
- The key idea is to **compare** the outputs of the programs and **report** the **differences** as the **potential bugs**.
- Thus, it utilizes the **cross-reference oracle**, which means that the **majority** of the outputs has a **higher probability** of being **correct**.

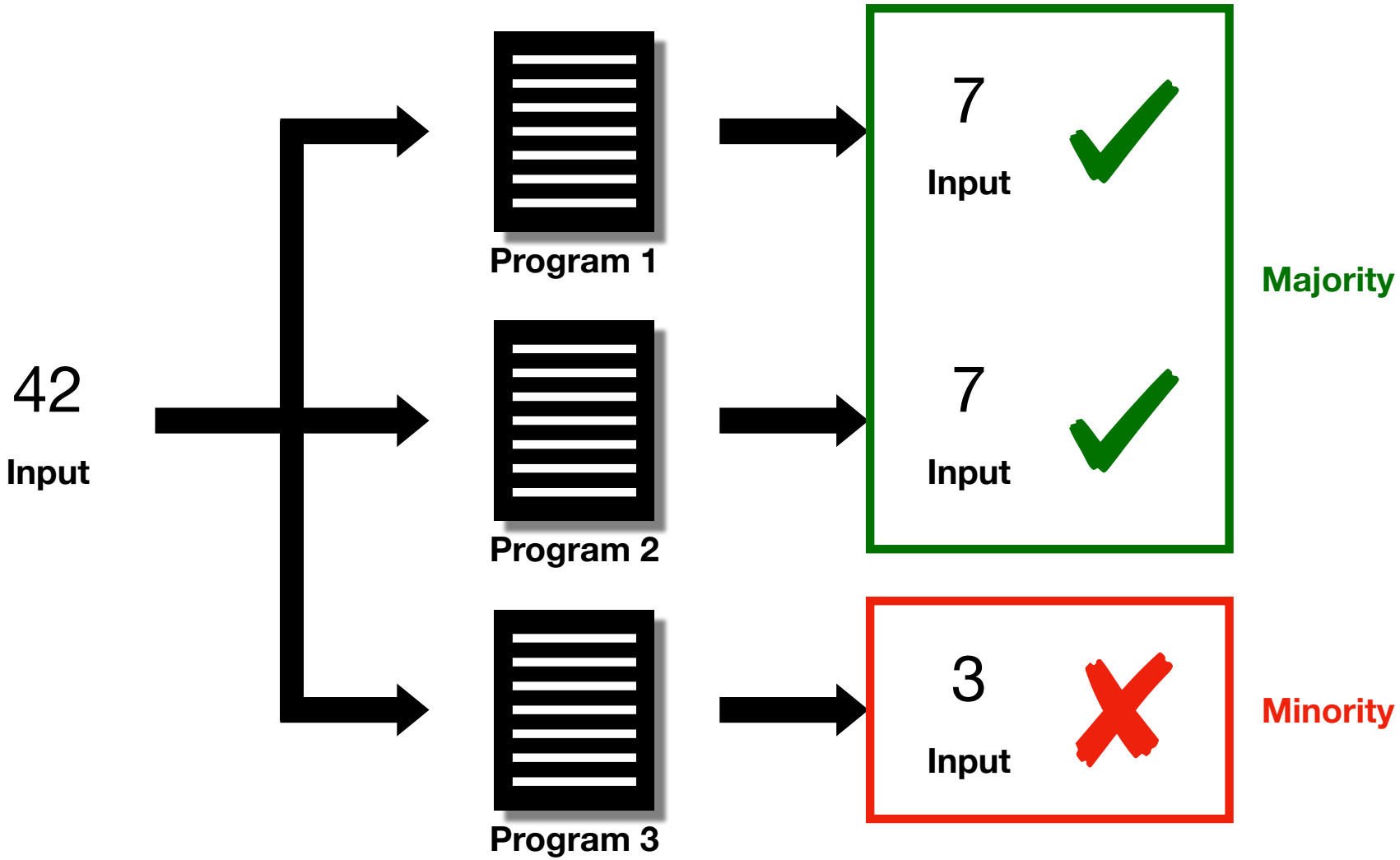
Differential Testing



Differential Testing

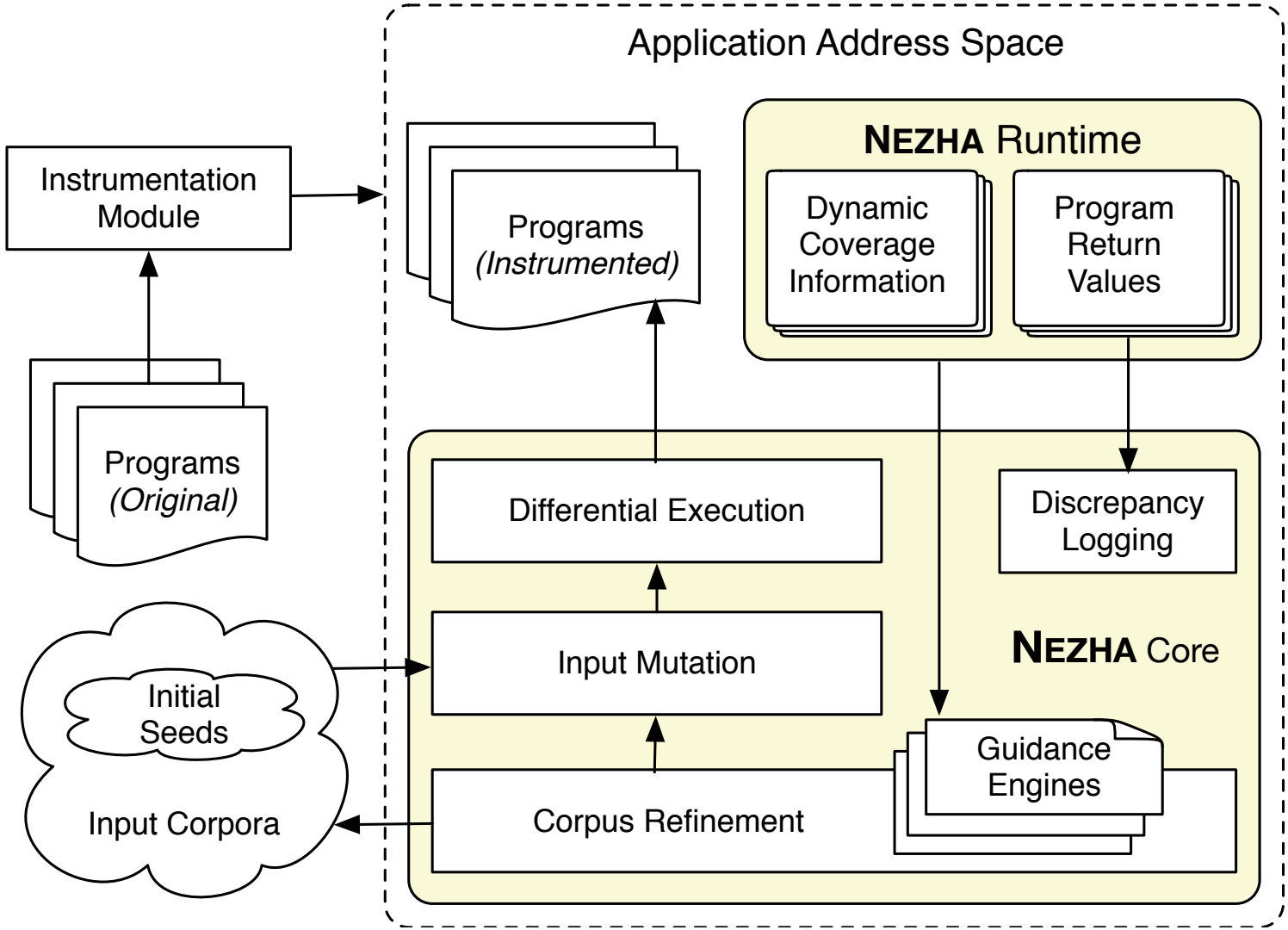


Differential Testing



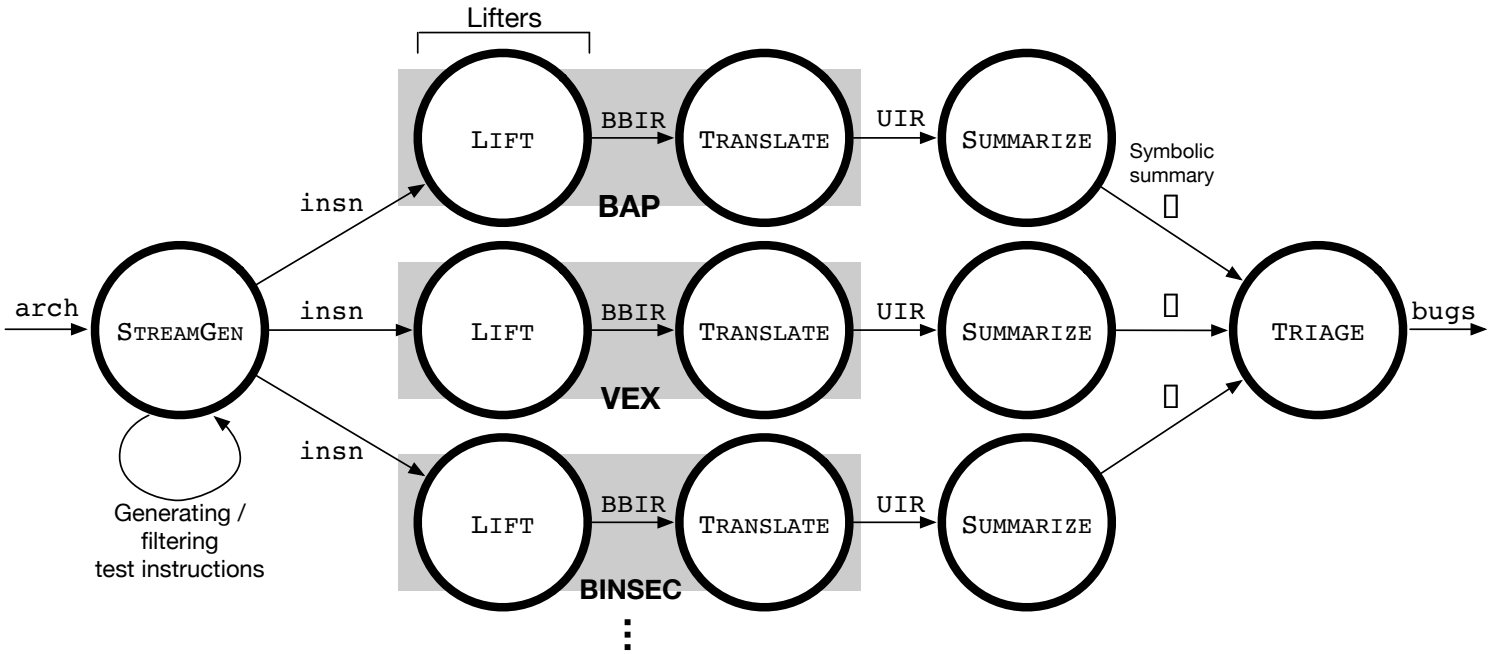
Examples: Nezha

T. Petsios, A. Tang, S. Stolfo, A. Keromytis, and S. Jana. Nezha: Efficient Domain-independent Differential Testing. S&P 2017.



Examples: Binary Lifters

S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. Cha. Testing Intermediate Representations of Binary Analysis. ASE 2017.

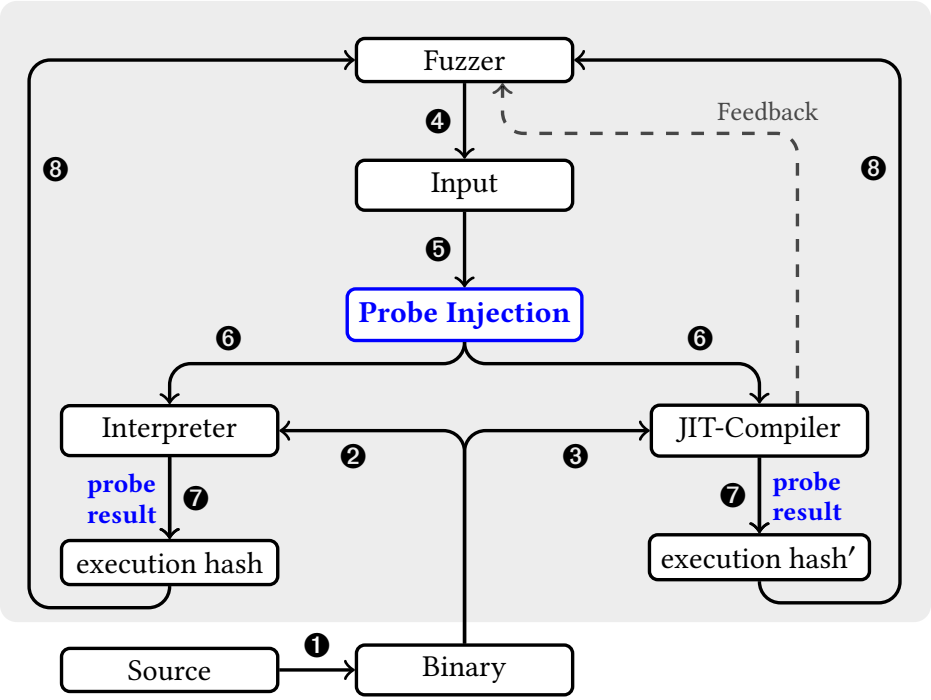


Binary lifters translate a **binary executable** into a high-level **intermediate representation (IR)** as a primary step in binary analysis.

MeanDiff performs a **differential testing** on the **IRs** generated by the **binary lifters** to find the **lifting bugs**.

Examples: JIT Compilers

L. Bernhard, T. Scharnowski, M. Schloegel, T. Blazytko, and T. Holz. JIT-Picking: Differential Testing of JavaScript Engines. CCS 2022.



Just-in-time (JIT) compilation is compilation during program execution to **improve** the **performance** and many JavaScript engines support it.

JIT-Picking performs a **differential testing** between the **results** of programs with and without **JIT compilation** to find the **JIT bugs**.

N+1-version Differential Testing

J. Park, S. An, and S. Ryu. JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification. ICSE 2021.

ECMAScript

GraalVM

QuickJS

moddable

JavaScript Engines

N+1-version Differential Testing

J. Park, S. An, and S. Ryu. JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification. ICSE 2021.

ECMAScript

Conformance

GraalVM

QuickJS

moddable

JavaScript Engines

N+1-version Differential Testing

J. Park, S. An, and S. Ryu. JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification. ICSE 2021.

ECMAScript

Conformance

Conformance

GraalVM

QuickJS

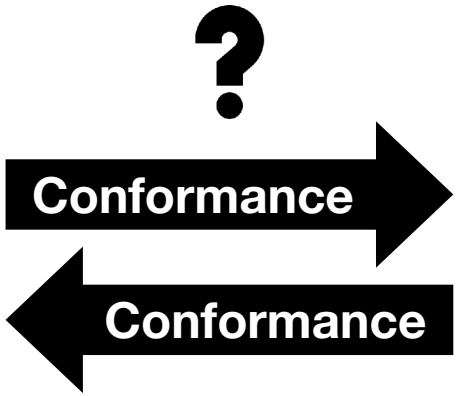
moddable

JavaScript Engines

N+1-version Differential Testing

J. Park, S. An, and S. Ryu. JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification. ICSE 2021.

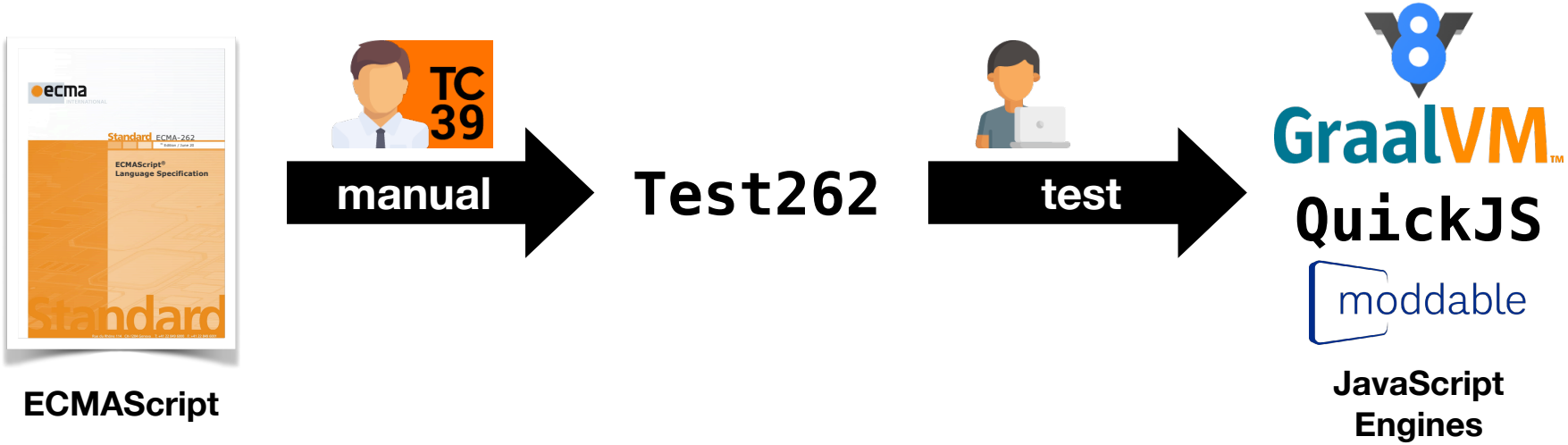
ECMAScript



JavaScript Engines

N+1-version Differential Testing

J. Park, S. An, and S. Ryu. JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification. ICSE 2021.

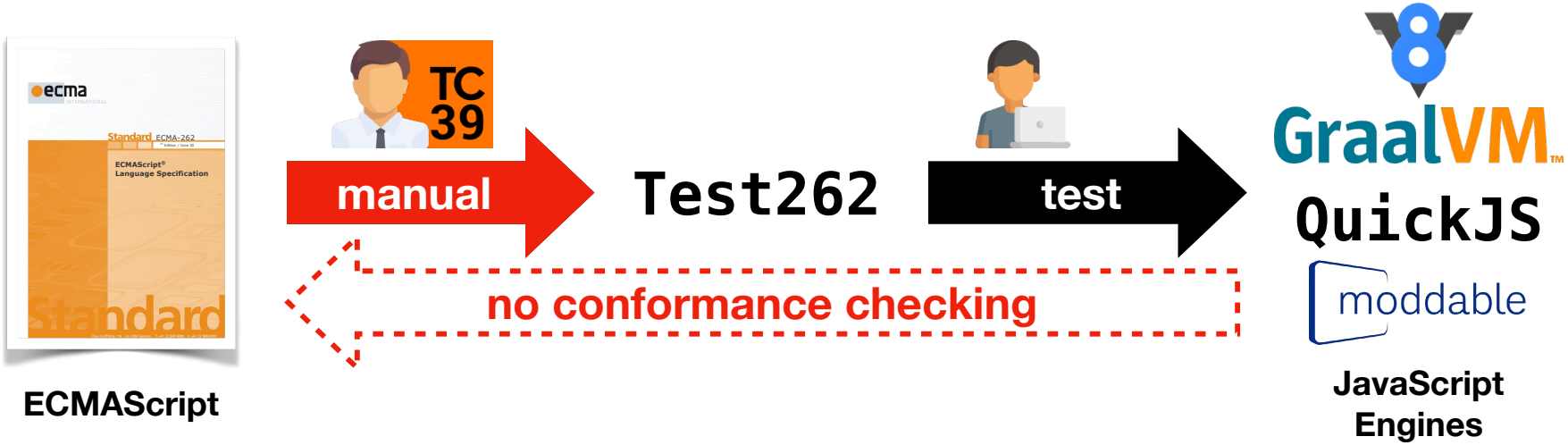


N+1-version Differential Testing

J. Park, S. An, and S. Ryu. JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification. ICSE 2021.

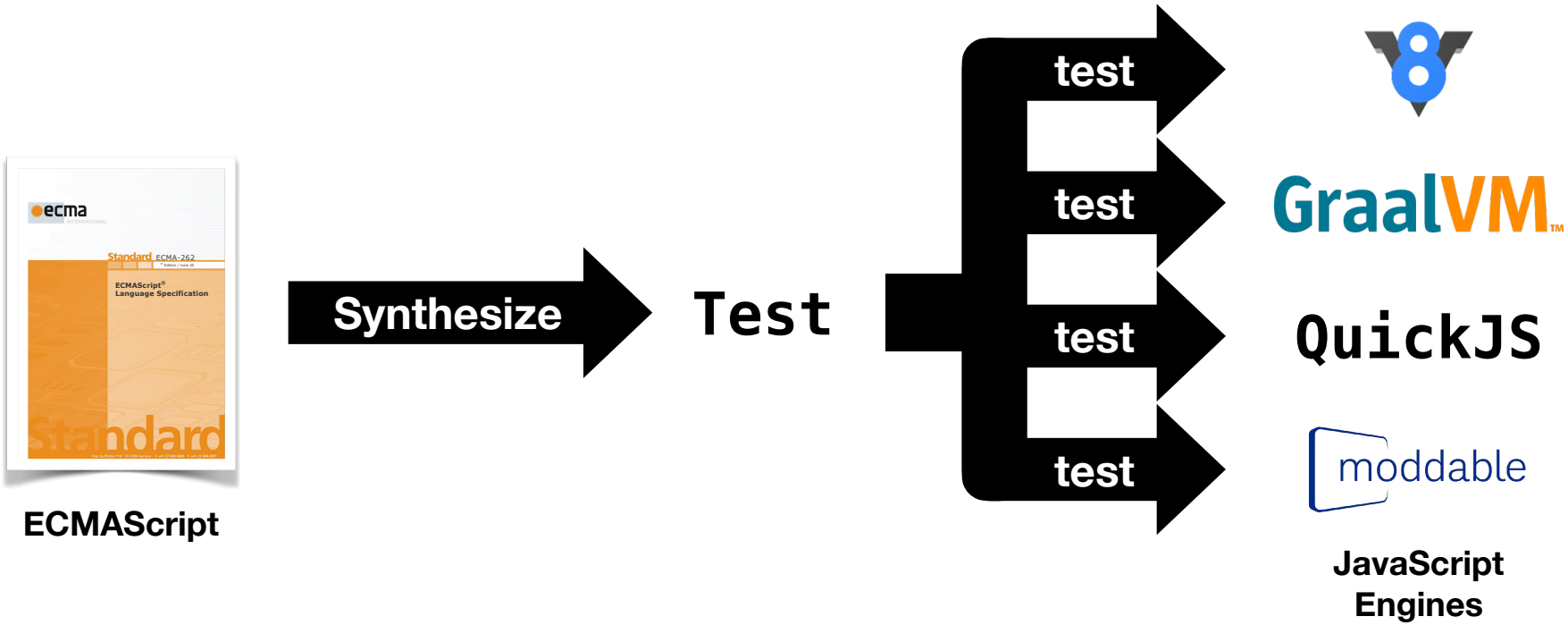
N+1-version Differential Testing

J. Park, S. An, and S. Ryu. JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification. ICSE 2021.



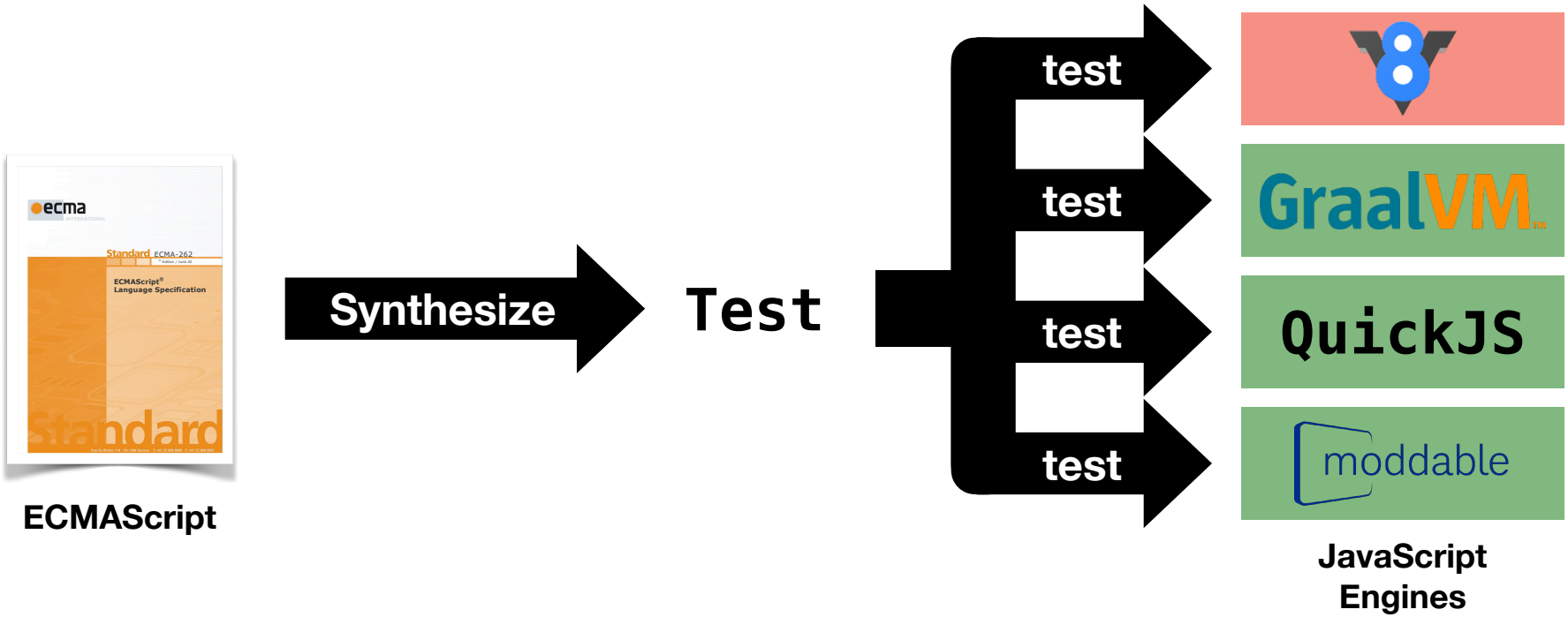
N+1-version Differential Testing

J. Park, S. An, and S. Ryu. JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification. ICSE 2021.



N+1-version Differential Testing

J. Park, S. An, and S. Ryu. JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification. ICSE 2021.



N+1-version Differential Testing

J. Park, S. An, and S. Ryu. JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification. ICSE 2021.

ECMAScript

Synthesize

Test

test

test

test

test

GraalVM

QuickJS

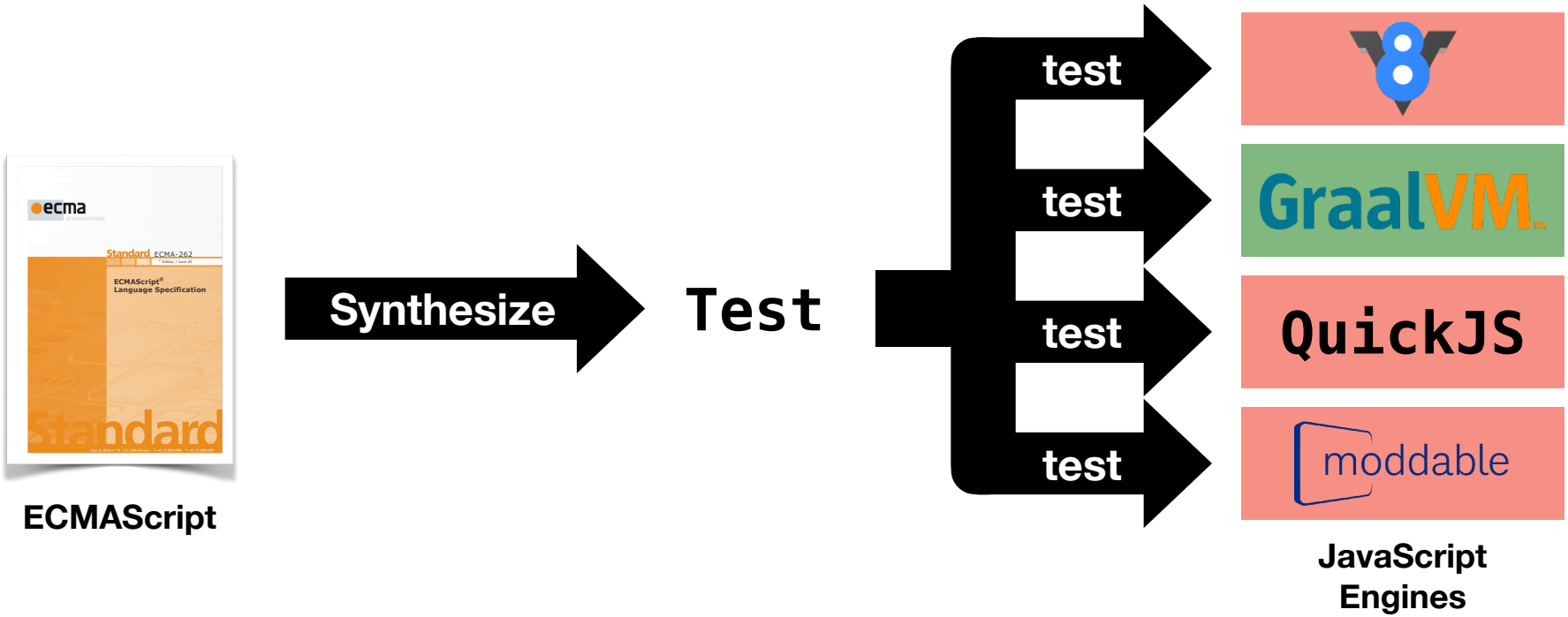
moddable

JavaScript Engines

An engine bug in

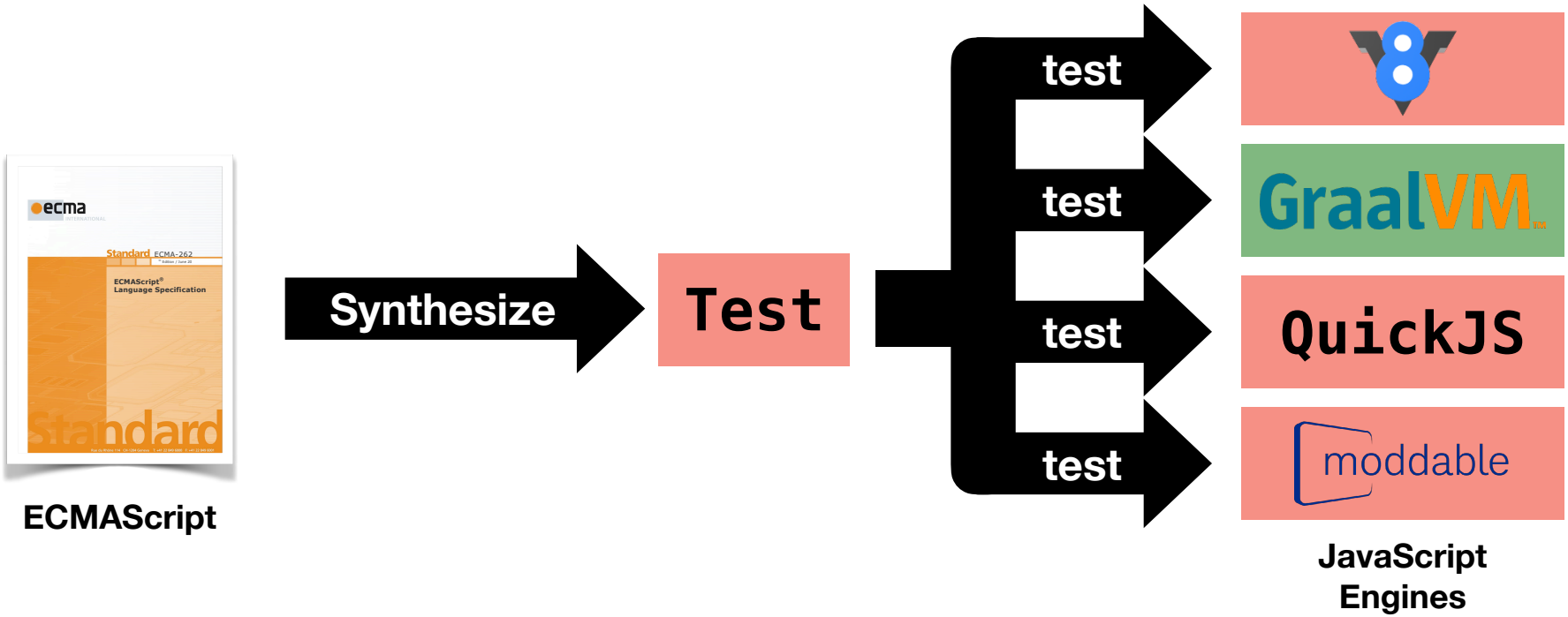
N+1-version Differential Testing

J. Park, S. An, and S. Ryu. JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification. ICSE 2021.



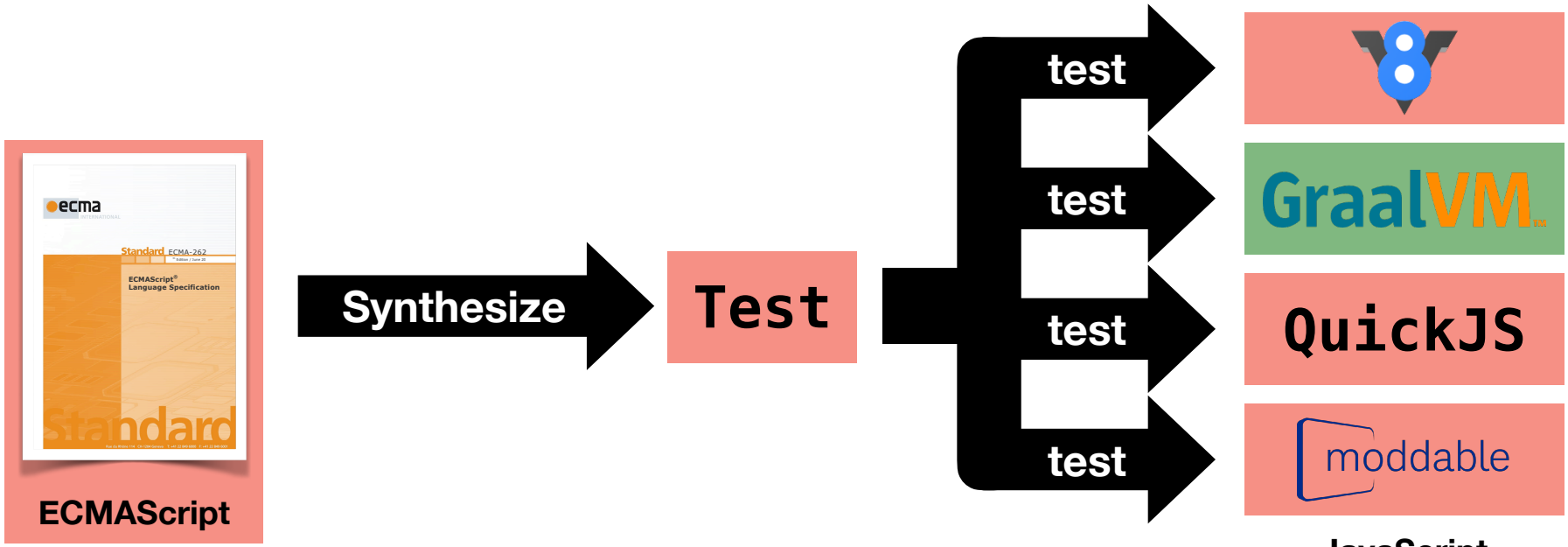
N+1-version Differential Testing

J. Park, S. An, and S. Ryu. JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification. ICSE 2021.



N+1-version Differential Testing

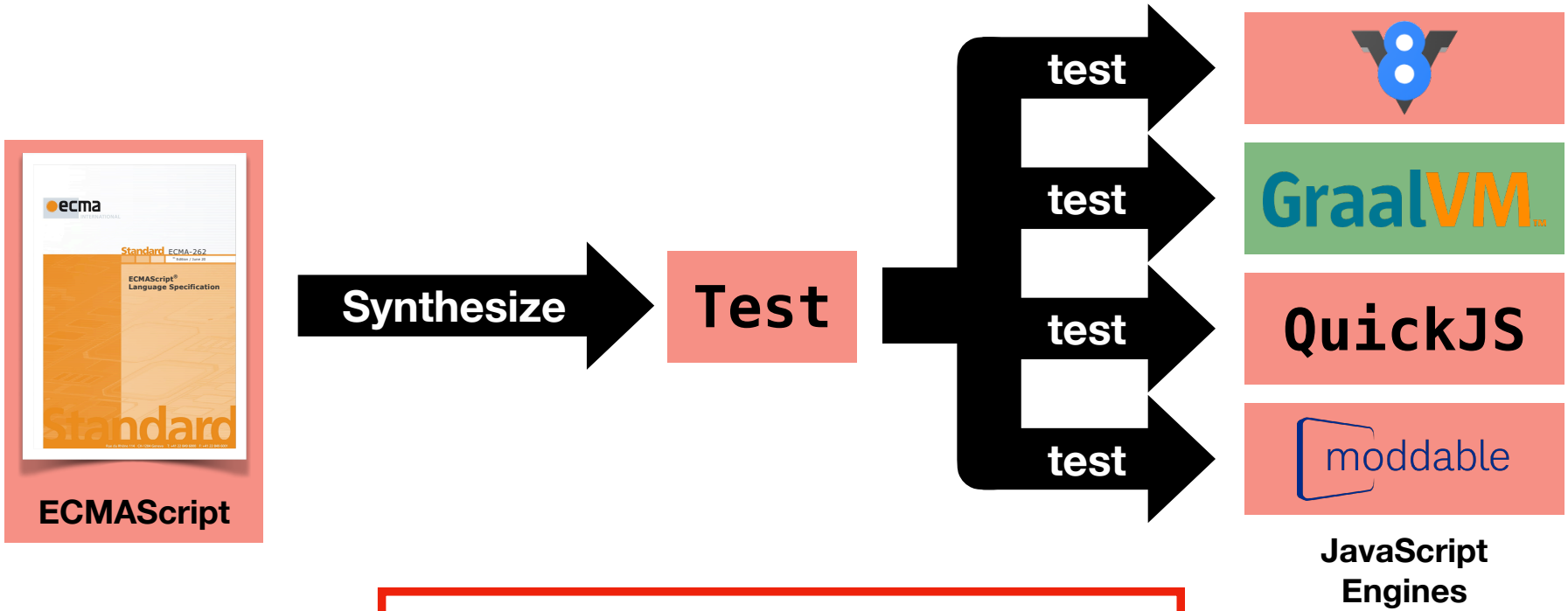
J. Park, S. An, and S. Ryu. JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification. ICSE 2021.



A specification bug in ECMAScript

N+1-version Differential Testing

J. Park, S. An, and S. Ryu. JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification. ICSE 2021.

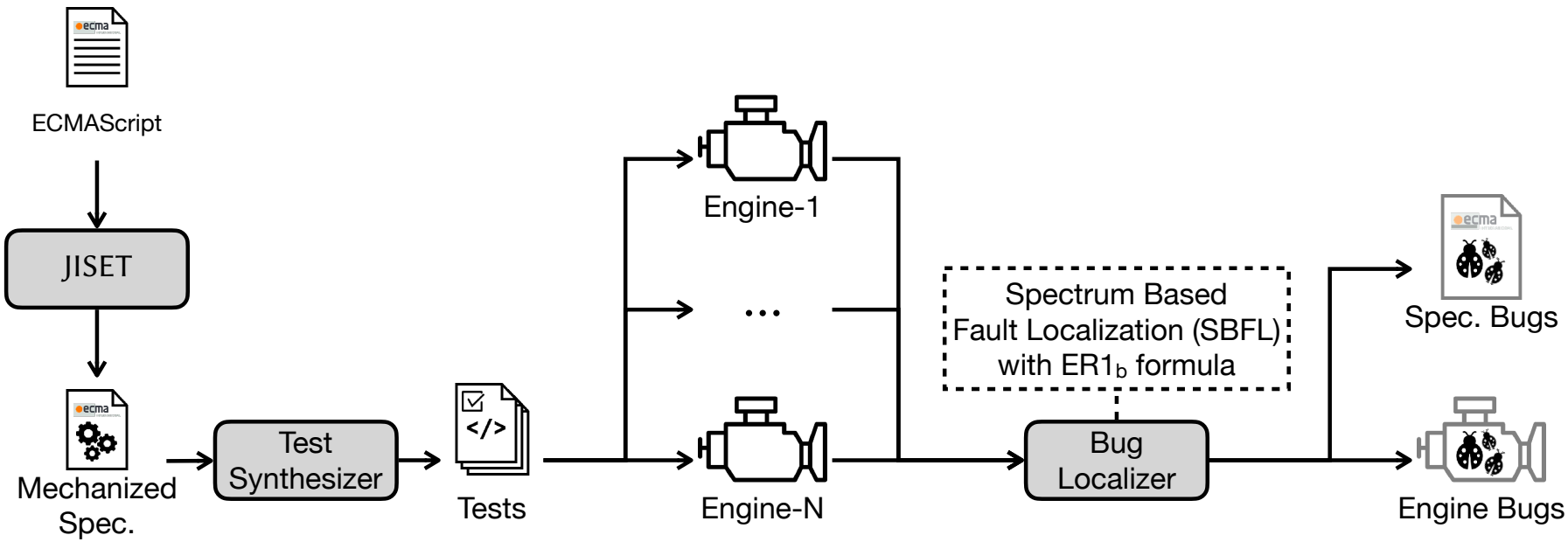


A specification bug in ECMAScript

An engine bug in GraalVM

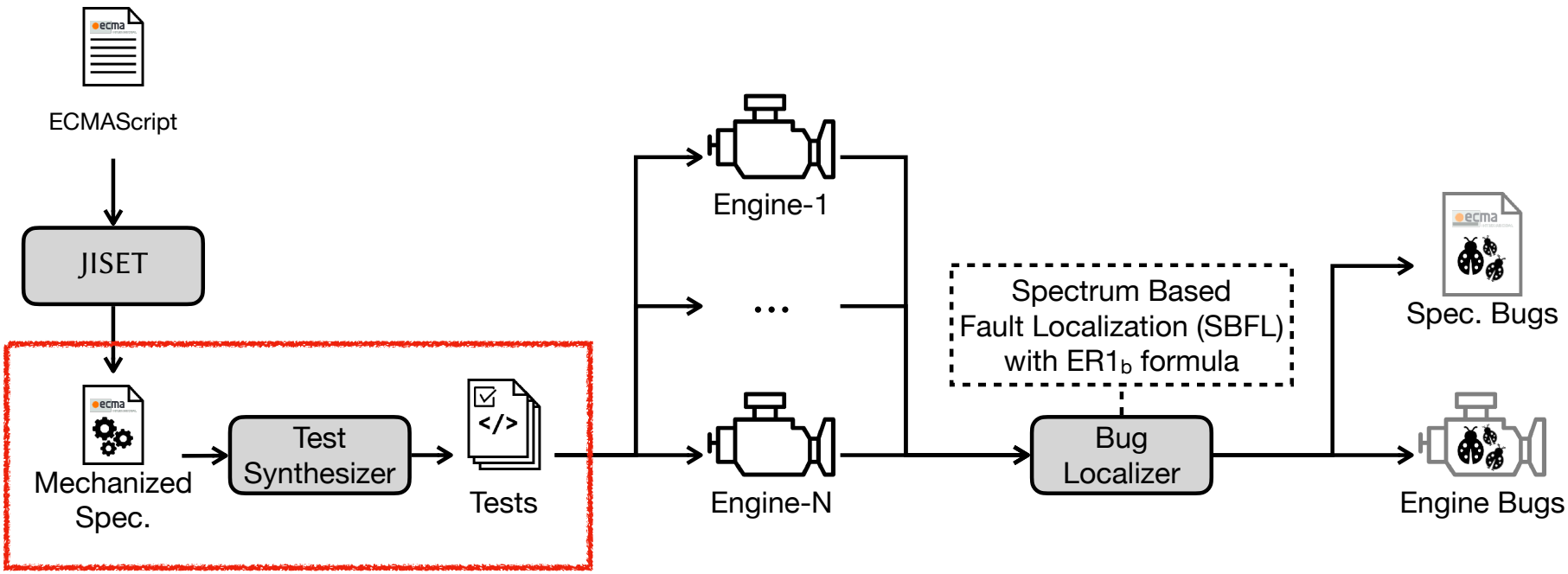
N+1-version Differential Testing

J. Park, S. An, and S. Ryu. JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification. ICSE 2021.



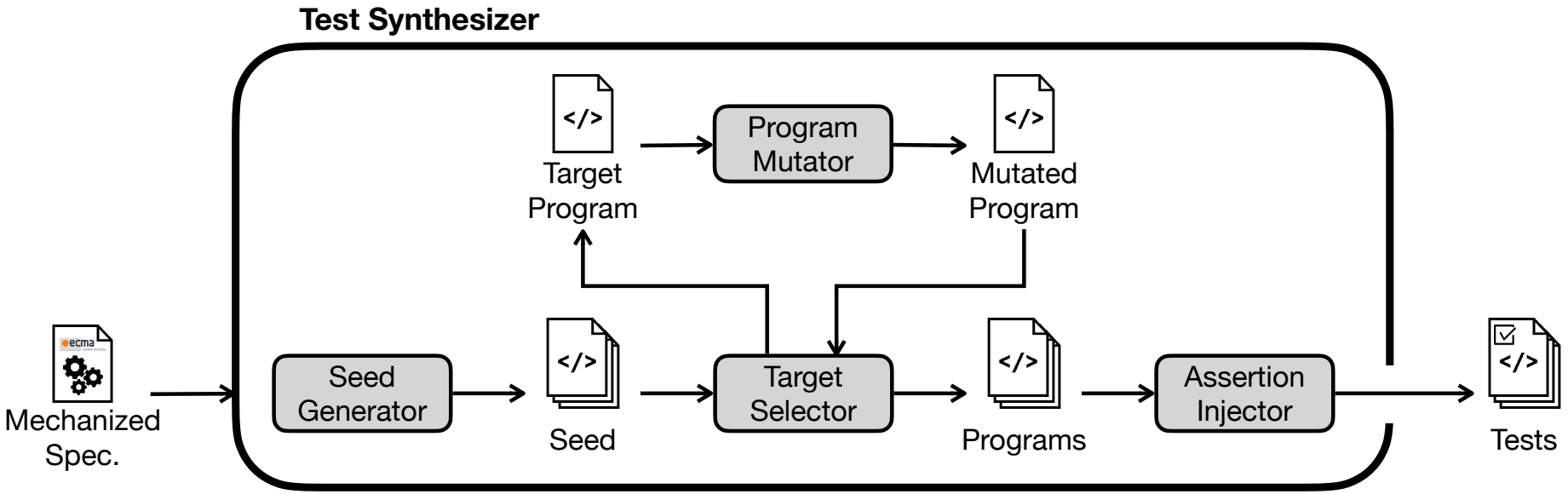
N+1-version Differential Testing

J. Park, S. An, and S. Ryu. JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification. ICSE 2021.



N+1-version Differential Testing

J. Park, S. An, and S. Ryu. JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification. ICSE 2021.



Contents

Property-based Testing

- **Property-based Testing** is a testing technique that tests the programs by specifying the **properties** that the programs should satisfy.

Property-based Testing

PLRG

- **Property-based Testing** is a testing technique that tests the programs by specifying the **properties** that the programs should satisfy.

- The key idea is originally from the **QuickCheck** tool for the **Haskell** programming language.

    - K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. ICFP 2000.

Property-based Testing

PLRG

- **Property-based Testing** is a testing technique that tests the programs by specifying the **properties** that the programs should satisfy.
- The key idea is originally from the **QuickCheck** tool for the **Haskell** programming language.
    - K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. ICFP 2000.
- It requires **property-based oracles**, instead of the input-output pair oracles, to test the programs.

Property-based Testing

- A traditional example-based oracle requires **input-output pairs**.

```
def abs(x):
  if x < 0:
    return -x
  return x

def test_abs():
  assert abs(0) == 0
  assert abs(1) == 1
  assert abs(-1) == 1
```

- A **property-based oracle** requires the **properties** of a given input.

```
def abs(x):
  if x < 0:
    return -x
  return x

def test_abs(x):
  assert abs(x) >= 0
  assert abs(x) == abs(-x)
  assert abs(abs(x)) == abs(x)
```

Tool – Hypothesis for Python

- **Hypothesis** (link) is a property-based testing tool for the **Python** (or Ruby and Java) programming language.

Tool – Hypothesis for Python

- **Hypothesis** (link) is a property-based testing tool for the **Python** (or Ruby and Java) programming language.

- It generates the **random inputs** for the programs and tests the programs with the **properties** that the programs should satisfy.

Tool – Hypothesis for Python

PLRG

- **Hypothesis** (link) is a property-based testing tool for the **Python** (or Ruby and Java) programming language.
- It generates the **random inputs** for the programs and tests the programs with the **properties** that the programs should satisfy.
- It is a **powerful** tool for testing the programs with the **complex properties**.

Tool – Hypothesis for Python

PLRG

- **Hypothesis** (link) is a property-based testing tool for the **Python** (or Ruby and Java) programming language.
- It generates the **random inputs** for the programs and tests the programs with the **properties** that the programs should satisfy.
- It is a **powerful** tool for testing the programs with the **complex properties**.
- It is a **lightweight** tool that can be easily integrated into the existing testing frameworks.

Tool – Hypothesis for Python

PLRG

```
from hypothesis import given, settings, Verbosity
from hypothesis import strategies as st
import unittest

def abs(x):
  if x < 0:
    return -x
  return x

@given(x = st.integers())
@settings(verbosity=Verbosity.verbose)
def test_abs(x):
  assert abs(x) >= 0
  assert abs(x) == abs(-x)
  assert abs(abs(x)) == abs(x)

test_abs()
```

Summary

PLRG

1. Non-Testable Programs
   - Types of Non-Testable Programs
2. Metamorphic Testing
   - Examples: SMT Solvers
   - Examples: Web Browser Debugger
   - Examples: Compilers
   - Examples: Deep Neural Networks
   - Examples: Object Detection
   - Learning Metamorphic Relationships
3. Differential Testing
   - Examples: Nezha
   - Examples: Binary Lifters
   - Examples: JIT Compilers
   - N+1-version Differential Testing
4. Property-based Testing

Next Lecture

- Course Review

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