Lecture 3 – Coverage Criteria

AAA705: Software Testing and Quality Assurance

Jihyeok Park

PLRG

2026 Fall

Recall

- Random Testing (RT)
  - Geometric distribution – $E(X) = 1/p$, $1 - (1 - p)^n$
  - Residual risk – n clean runs only give $p < \ln(1/\delta)/n$
- Feedback-Directed Random Testing (Randoop)
- Adaptive Random Testing (ART)
  - Distance between inputs, and the 50% upper bound
- Fuzz Testing
  - Generation-based (grammars, Csmith) / mutation-based (bytes, ASTs)
  - **Coverage-guided** fuzzing, and directed fuzzing
  - Sanitizers as oracles, and crash triage

Covered *What*, Exactly?

Last week a fuzzer kept a mutant when it **covered something new**. That "something" was one particular answer: a bitmap of **branches taken**.

Covered *What*, Exactly?

Last week a fuzzer kept a mutant when it **covered something new**. That "something" was one particular answer: a bitmap of **branches taken**.

- A **coverage criterion** needs a **structure** to be defined on – a **graph** or a **logic expression**.

Covered *What*, Exactly?

Last week a fuzzer kept a mutant when it **covered something new**. That "something" was one particular answer: a bitmap of **branches taken**.

- A **coverage criterion** needs a **structure** to be defined on – a **graph** or a **logic expression**.
- It turns that structure into a finite set of **test requirements**, so "how much have we tested?" becomes a **number**.

Covered *What*, Exactly?

Last week a fuzzer kept a mutant when it **covered something new**. That "something" was one particular answer: a bitmap of **branches taken**.

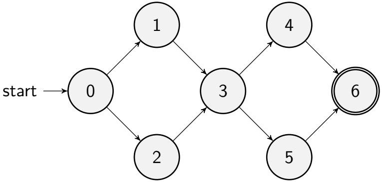
- A **coverage criterion** needs a **structure** to be defined on – a **graph** or a **logic expression**.
- It turns that structure into a finite set of **test requirements**, so "how much have we tested?" becomes a **number**.
- And then the honest question: does a bigger number mean **better tests**? The same question comes back for neural networks.

Contents

Contents

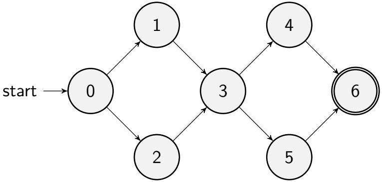
Programs Are Graphs

```
int f(int a, int b) {
  int x = 42;             // 0
  if (a > 0) g();         // 1
  else       h();         // 2
  int y;
  if (b > 0)              // 3
    y = x * 2;            // 4
  else
    y = x - 5;            // 5
  return y;               // 6
}
```



Programs Are Graphs

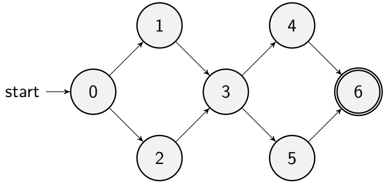
```
int f(int a, int b) {
  int x = 42;             // 0
  if (a > 0) g();         // 1
  else       h();         // 2
  int y;
  if (b > 0)              // 3
    y = x * 2;            // 4
  else
    y = x - 5;            // 5
  return y;               // 6
}
```



- A **control flow graph (CFG)** is one of many: **call graphs**, **finite state machines**, class hierarchies, spec structures.

Programs Are Graphs

```
int f(int a, int b) {
  int x = 42;           // 0
  if (a > 0) g();       // 1
  else       h();       // 2
  int y;
  if (b > 0)            // 3
    y = x * 2;          // 4
  else
    y = x - 5;          // 5
  return y;             // 6
}
```



- A **control flow graph (CFG)** is one of many: **call graphs**, **finite state machines**, class hierarchies, spec structures.
- Define coverage **once** on graphs, and it applies to all of them.

Graphs and Paths

Definition (Graph)

A **graph** $G = (N, E, N_s, N_f)$ consists of a set of **nodes** N, a set of **edges** $E \subseteq N \times N$, a set of **start nodes** $N_s \subseteq N$, and a set of **final nodes** $N_f \subseteq N$.

Graphs and Paths

PLRG

Definition (Graph)

A **graph** $G = (N, E, N_s, N_f)$ consists of a set of **nodes** N, a set of **edges** $E \subseteq N \times N$, a set of **start nodes** $N_s \subseteq N$, and a set of **final nodes** $N_f \subseteq N$.

- A **path** $p = (n_0, \ldots, n_k) \in N^*$ satisfies $(n_i, n_{i+1}) \in E$ for $0 \leq i < k$; its **length** is $|p| = k$; and P_G is the set of all paths in G.

Graphs and Paths

Definition (Graph)

A **graph** $G = (N, E, N_s, N_f)$ consists of a set of **nodes** N, a set of **edges** $E \subseteq N \times N$, a set of **start nodes** $N_s \subseteq N$, and a set of **final nodes** $N_f \subseteq N$.

- A **path** $p = (n_0, \ldots, n_k) \in N^*$ satisfies $(n_i, n_{i+1}) \in E$ for $0 \leq i < k$; its **length** is $|p| = k$; and P_G is the set of all paths in G.
- A **subpath** $q \preceq p$ is a **contiguous** subsequence: $\exists\, 0 \leq i \leq j \leq k.\ q = (n_i, \ldots, n_j)$.

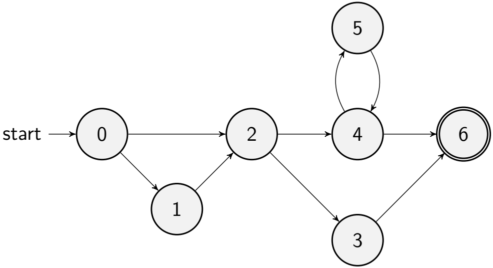
Definition (Graph)

A **graph** $G = (N, E, N_s, N_f)$ consists of a set of **nodes** N, a set of **edges** $E \subseteq N \times N$, a set of **start nodes** $N_s \subseteq N$, and a set of **final nodes** $N_f \subseteq N$.

- A **path** $p = (n_0, \ldots, n_k) \in N^*$ satisfies $(n_i, n_{i+1}) \in E$ for $0 \leq i < k$; its **length** is $|p| = k$; and P_G is the set of all paths in G.
- A **subpath** $q \preceq p$ is a **contiguous** subsequence: $\exists 0 \leq i \leq j \leq k.\ q = (n_i, \ldots, n_j)$.
- A **test path** starts at a start node and ends at a final node ($n_0 \in N_s \wedge n_k \in N_f$) – it is what an **execution** of a test case looks like. We write path(t) for the test path of a test case t, and path(T) for those of a test suite T.

Paths – Example

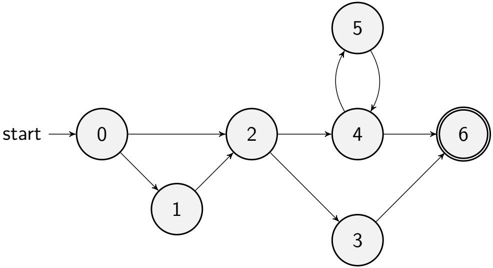
Consider a path $p = [0, 2, 4, 5, 4, 6]$ in the graph G.

$G =$ 

Paths – Example

Consider a path $p = [0, 2, 4, 5, 4, 6]$ in the graph G.

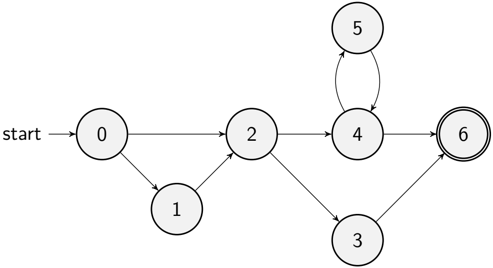
G =



- $|p| = 5$, and p is a **test path** $(0 \in N_s,\ 6 \in N_f)$

Paths – Example

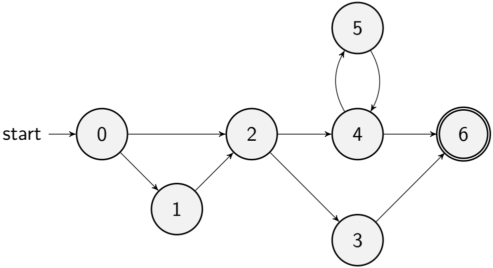
Consider a path $p = [0, 2, 4, 5, 4, 6]$ in the graph G.

$G =$ 

- $|p| = 5$, and p is a **test path** $(0 \in N_s,\ 6 \in N_f)$
- $[2, 4, 5] \preceq p$, but $[2, 4, 6] \not\preceq p$ – **contiguous**, not just in order

Paths – Example

Consider a path $p = [0, 2, 4, 5, 4, 6]$ in the graph G.

$G \quad =$ 

- $|p| = 5$, and p is a **test path** ($0 \in N_s$, $6 \in N_f$)
- $[2, 4, 5] \preceq p$, but $[2, 4, 6] \not\preceq p$ – **contiguous**, not just in order
- p **visits** nodes 0, 2, 4, 5, 6 and edges $(0, 2)$, $(2, 4)$, $(4, 5)$, $(5, 4)$, $(4, 6)$

Graph Coverage Criterion

Definition (Graph Coverage Criterion)

A **graph coverage criterion** $C_G = (R_G, \sim)$ for a graph G is

- a set of **test requirements (TRs)** R_G, and
- a **cover relation** $\sim \subseteq P_G \times R_G$ between paths and test requirements.

Graph Coverage Criterion

Definition (Graph Coverage Criterion)

A **graph coverage criterion** $C_G = (R_G, \sim)$ for a graph G is

- a set of **test requirements (TRs)** R_G, and
- a **cover relation** $\sim \subseteq P_G \times R_G$ between paths and test requirements.

- A test case t **covers** a TR r if its test path does:
  $t \sim r \iff \text{path}(t) \sim r$.

Graph Coverage Criterion

Definition (Graph Coverage Criterion)

A **graph coverage criterion** $C_G = (R_G, \sim)$ for a graph G is

- a set of **test requirements (TRs)** R_G, and
- a **cover relation** $\sim \subseteq P_G \times R_G$ between paths and test requirements.

- A test case t **covers** a TR r if its test path does:
  $t \sim r \iff \mathrm{path}(t) \sim r$.
- A test suite T **satisfies** C_G if it covers every TR.

$$T \vdash C_G \quad \iff \quad \forall r \in R_G.\ \exists t \in T.\ t \sim r$$

Graph Coverage Criterion

Definition (Graph Coverage Criterion)

A **graph coverage criterion** $C_G = (R_G, \sim)$ for a graph G is

- a set of **test requirements (TRs)** R_G, and
- a **cover relation** $\sim \subseteq P_G \times R_G$ between paths and test requirements.

- A test case t **covers** a TR r if its test path does: $t \sim r \iff \text{path}(t) \sim r$.
- A test suite T **satisfies** C_G if it covers every TR.

$$T \vdash C_G \quad \iff \quad \forall r \in R_G.\ \exists t \in T.\ t \sim r$$

- Everything below is a choice of R_G and $\sim$: **structural** criteria use only nodes, edges, and paths; **data-flow** criteria use a graph annotated with **variables**.

Node and Edge Coverage

Definition (Node Coverage (NC))

$R_G = N$, and a path p **covers** a node n if p visits n.

Node and Edge Coverage

Definition (Node Coverage (NC))

$R_G = N$, and a path p **covers** a node n if p visits n.

Definition (Edge Coverage (EC))

$R_G = E$, and a path p **covers** an edge (n, m) if p visits (n, m).

Node and Edge Coverage

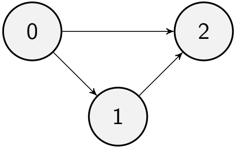
Definition (Node Coverage (NC))

$R_G = N$, and a path p **covers** a node n if p visits n.

Definition (Edge Coverage (EC))

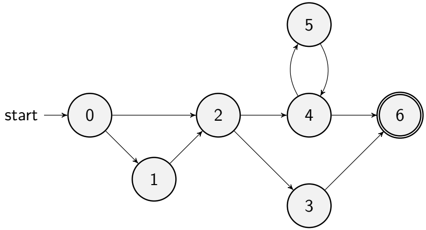
$R_G = E$, and a path p **covers** an edge (n, m) if p visits (n, m).

NC and EC differ only when there is **both** an edge and another subpath between a pair of nodes – an `if` without an `else` is the smallest case: $[0, 2]$ visits every node but misses $(0, 1)$.



Node and Edge Coverage – Example

$G =$

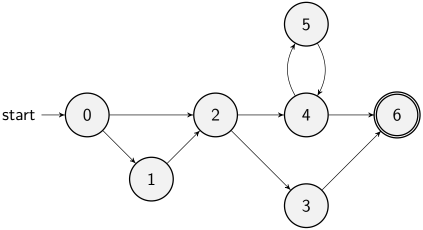


- **Node Coverage (NC)**

$$R_G = \{0, 1, 2, 3, 4, 5, 6\}$$
$$\text{Test Paths} = \{[0, 1, 2, 3, 6], [0, 1, 2, 4, 5, 4, 6]\}$$

Node and Edge Coverage – Example

$G \quad =$



- **Node Coverage (NC)**

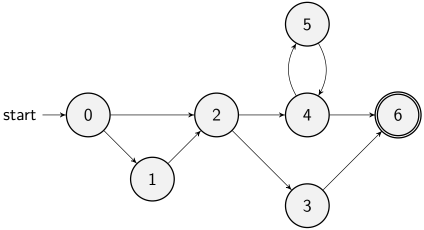
$$R_G = \{0, 1, 2, 3, 4, 5, 6\}$$
$$\text{Test Paths} = \{[0, 1, 2, 3, 6], [0, 1, 2, 4, 5, 4, 6]\}$$

- **Edge Coverage (EC)**

$$R_G = \{(0, 1), (0, 2), (1, 2), (2, 3), (2, 4), (3, 6), (4, 5), (5, 4), (4, 6)\}$$
$$\text{Test Paths} = \{[0, 1, 2, 3, 6], \color{red}{[0, 2, 4, 5, 4, 6]}\}$$

Node and Edge Coverage – Example

$G =$



- **Node Coverage (NC)**

$$R_G = \{0, 1, 2, 3, 4, 5, 6\}$$
$$\text{Test Paths} = \{[0, 1, 2, 3, 6], [0, 1, 2, 4, 5, 4, 6]\}$$

- **Edge Coverage (EC)**

$$R_G = \{(0, 1), (0, 2), (1, 2), (2, 3), (2, 4), (3, 6), (4, 5), (5, 4), (4, 6)\}$$
$$\text{Test Paths} = \{[0, 1, 2, 3, 6], \color{red}{[0, 2, 4, 5, 4, 6]}\}$$

- The NC suite is **not** EC-adequate: it never takes $(0, 2)$.

k-Limiting Path Coverage

Definition (k-Limiting Path Coverage (k-PC))

- $R_G = \{p \in P_G \mid |p| \leq k\}$
- $p \sim q \iff q \preceq p$

k-Limiting Path Coverage

PLRG

Definition (k-Limiting Path Coverage (k-PC))

- $R_G = \{p \in P_G \mid |p| \leq k\}$
- $p \sim q \iff q \preceq p$

- The criteria we just defined are the small cases of this one:

$k = 0:$	paths of length $0 =$ **nodes**	**node coverage (NC)**
$k = 1:$	$+$ paths of length $1 =$ **edges**	**edge coverage (EC)**
$k = 2:$	$+$ pairs of adjacent edges	**edge-pair coverage (EPC)**
$k = \infty:$	every path	**complete path coverage (CPC)**

k-Limiting Path Coverage

Definition (k-Limiting Path Coverage (k-PC))

- $R_G = \{p \in P_G \mid |p| \leq k\}$
- $p \sim q \iff q \preceq p$

- The criteria we just defined are the small cases of this one:

$k = 0$:	paths of length $0 =$ **nodes**	**node coverage (NC)**
$k = 1$:	$+$ paths of length $1 =$ **edges**	**edge coverage (EC)**
$k = 2$:	$+$ pairs of adjacent edges	**edge-pair coverage (EPC)**
$k = \infty$:	every path	**complete path coverage (CPC)**

- Strictly, 1-PC is NC **and** EC together. They coincide as long as every node touches an edge – an **isolated** node is a TR of 1-PC that EC alone would miss.

k-Limiting Path Coverage

PLRG

Definition (k-Limiting Path Coverage (k-PC))

- $R_G = \{p \in P_G \mid |p| \leq k\}$
- $p \sim q \iff q \preceq p$

- The criteria we just defined are the small cases of this one:

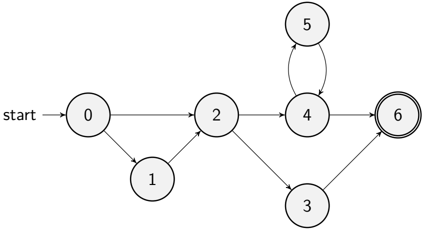
$k = 0$:	paths of length $0 =$ **nodes**	**node coverage (NC)**
$k = 1$:	$+$ paths of length $1 =$ **edges**	**edge coverage (EC)**
$k = 2$:	$+$ pairs of adjacent edges	**edge-pair coverage (EPC)**
$k = \infty$:	every path	**complete path coverage (CPC)**

- Strictly, 1-PC is NC **and** EC together. They coincide as long as every node touches an edge – an **isolated** node is a TR of 1-PC that EC alone would miss.

- So k is a single **knob** trading cost for strength.

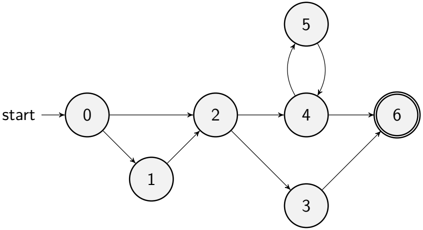
Edge-Pair Coverage – Example

$G =$



Edge-Pair Coverage – Example

G =

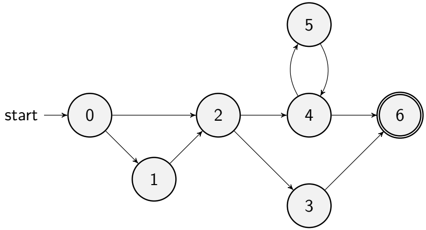


- **11** requirements of length 2 (plus the nodes and edges):

$$[0,1,2],\ [0,2,3],\ [0,2,4],\ [1,2,3],\ [1,2,4],\ [2,3,6],$$
$$[2,4,5],\ [2,4,6],\ [4,5,4],\ [5,4,5],\ [5,4,6]$$

Edge-Pair Coverage – Example

G =

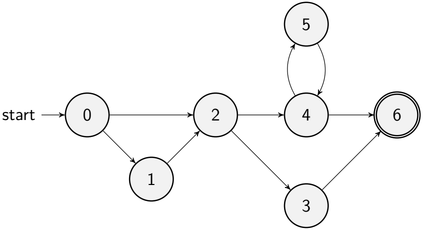


- **11** requirements of length 2 (plus the nodes and edges):

$$[0,1,2],\ [0,2,3],\ [0,2,4],\ [1,2,3],\ [1,2,4],\ [2,3,6],$$
$$[2,4,5],\ [2,4,6],\ [4,5,4],\ [5,4,5],\ [5,4,6]$$

- Test Paths
$= \{[0,1,2,3,6],\ [0,1,2,4,6],\ [0,2,3,6],\ [0,2,4,5,4,5,4,6]\}$

Edge-Pair Coverage – Example

$G =$ 

- **11** requirements of length 2 (plus the nodes and edges):

$$[0,1,2],\ [0,2,3],\ [0,2,4],\ [1,2,3],\ [1,2,4],\ [2,3,6],$$
$$[2,4,5],\ [2,4,6],\ [4,5,4],\ [5,4,5],\ [5,4,6]$$

- Test Paths
  $= \{[0,1,2,3,6],\ [0,1,2,4,6],\ [0,2,3,6],\ [0,2,4,5,4,5,4,6]\}$
- Note $[5,4,5]$: covering it needs $4 \rightarrow 5 \rightarrow 4 \rightarrow 5$, so EPC forces the loop body **twice**.

Loops Break Path Coverage

- One loop makes P_G **infinite**, so CPC is infeasible. Which finitely many paths should stand in for the infinitely many?

Loops Break Path Coverage

- One loop makes P_G **infinite**, so CPC is infeasible. Which finitely many paths should stand in for the infinitely many?

- **Specified path coverage (SPC)** lets the tester list them – and makes the criterion depend on the tester's expertise.

Loops Break Path Coverage

- One loop makes P_G **infinite**, so CPC is infeasible. Which finitely many paths should stand in for the infinitely many?
- **Specified path coverage (SPC)** lets the tester list them – and makes the criterion depend on the tester's expertise.
- Four decades of answers:
  - 1970s: execute each cycle once ($[4, 5, 4]$ above)
  - 1980s: execute each loop exactly once
  - 1990s: execute each loop 0 times, once, more than once
  - 2000s: **prime paths** – get all three for free

Simple and Prime Paths

Definition (Simple Path)

A **simple path** has no repeated node, except that the first and the last may coincide. (No internal loop; a whole loop is a simple path.)

Simple and Prime Paths

PLRG

Definition (Simple Path)

A **simple path** has no repeated node, except that the first and the last may coincide. (No internal loop; a whole loop is a simple path.)

Definition (Prime Path)

A **prime path** is a simple path that is not a subpath of any other simple path – a **maximal** simple path.

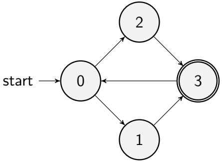
Simple and Prime Paths

Definition (Simple Path)

A **simple path** has no repeated node, except that the first and the last may coincide. (No internal loop; a whole loop is a simple path.)

Definition (Prime Path)

A **prime path** is a simple path that is not a subpath of any other simple path – a **maximal** simple path.



23 simple paths, of which **8** are prime:
$[0, 1, 3, 0]$, $[0, 2, 3, 0]$, $[1, 3, 0, 1]$, $[1, 3, 0, 2]$, $[2, 3, 0, 1]$, $[2, 3, 0, 2]$, $[3, 0, 1, 3]$, $[3, 0, 2, 3]$

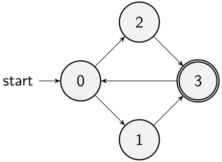
Simple and Prime Paths

Definition (Simple Path)

A **simple path** has no repeated node, except that the first and the last may coincide. (No internal loop; a whole loop is a simple path.)

Definition (Prime Path)

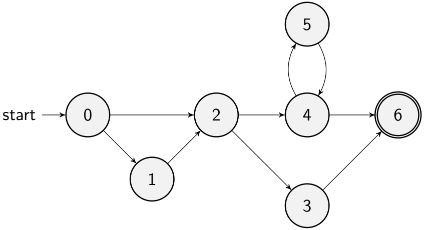
A **prime path** is a simple path that is not a subpath of any other simple path – a **maximal** simple path.



23 simple paths, of which **8** are prime:
$[0, 1, 3, 0]$, $[0, 2, 3, 0]$, $[1, 3, 0, 1]$, $[1, 3, 0, 2]$,
$[2, 3, 0, 1]$, $[2, 3, 0, 2]$, $[3, 0, 1, 3]$, $[3, 0, 2, 3]$

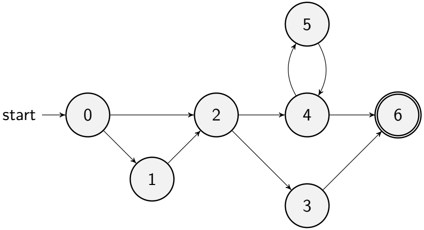
The other 15 are subpaths of these: $[0]$, $[0, 1]$, $[0, 1, 3]$, ...

Prime Paths Handle Loops



38 simple paths,	**9** prime:
[0, 1, 2, 3, 6]	[0, 2, 4, 5]
[0, 1, 2, 4, 5]	[0, 2, 4, 6]
[0, 1, 2, 4, 6]	[4, 5, 4]
[0, 2, 3, 6]	[5, 4, 5]
	[5, 4, 6]

Prime Paths Handle Loops



38 simple paths,

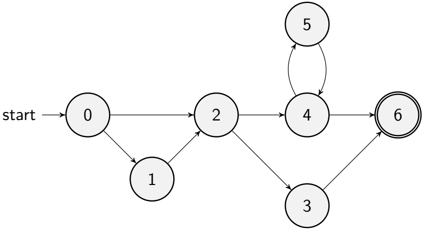
[0, 1, 2, 3, 6]
[0, 1, 2, 4, 5]
[0, 1, 2, 4, 6]
[0, 2, 3, 6]

9 prime:

[0, 2, 4, 5]
[0, 2, 4, 6]
[4, 5, 4]
[5, 4, 5]
[5, 4, 6]

[0, 2, 4, 6] executes the loop **0 times**

Prime Paths Handle Loops



38 simple paths,

[0, 1, 2, 3, 6]
[0, 1, 2, 4, 5]
[0, 1, 2, 4, 6]
[0, 2, 3, 6]

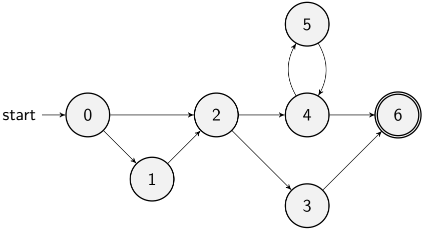
9 prime:

[0, 2, 4, 5]
[0, 2, 4, 6]
[4, 5, 4]
[5, 4, 5]
[5, 4, 6]

[0, 2, 4, 6] executes the loop **0 times**

[4, 5, 4] executes the loop **once**

Prime Paths Handle Loops



38 simple paths,

[0, 1, 2, 3, 6]
[0, 1, 2, 4, 5]
[0, 1, 2, 4, 6]
[0, 2, 3, 6]

9 prime:

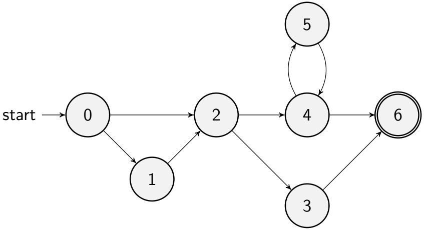
[0, 2, 4, 5]
[0, 2, 4, 6]
[4, 5, 4]
[5, 4, 5]
[5, 4, 6]

[0, 2, 4, 6] executes the loop **0 times**

[4, 5, 4] executes the loop **once**

[5, 4, 5] executes the loop **more than once**

Prime Paths Handle Loops



38 simple paths,

[0, 1, 2, 3, 6]
[0, 1, 2, 4, 5]
[0, 1, 2, 4, 6]
[0, 2, 3, 6]

9 prime:

[0, 2, 4, 5]
[0, 2, 4, 6]
[4, 5, 4]
[5, 4, 5]
[5, 4, 6]

[0, 2, 4, 6] executes the loop **0 times**

[4, 5, 4] executes the loop **once**

[5, 4, 5] executes the loop **more than once**

Prime path coverage needs all three, so it handles loops without a special rule.

Prime Path and Round-Trip Coverage

Definition (Prime Path Coverage (PPC))

$R_G = \{p \in P_G \mid p \text{ is a prime path}\}$, and $p \sim q \iff q \preceq p$.

[1] **[Binder'99]** R. V. Binder. "*Testing Object-Oriented Systems: Models, Patterns, and Tools.*" **[ISSRE'12]** N. E. Holt, R. Torkar, L. Briand, K. Hansen. "*State-Based Testing.*"

Prime Path and Round-Trip Coverage

Definition (Prime Path Coverage (PPC))

$R_G = \{p \in P_G \mid p \text{ is a prime path}\}$, and $p \sim q \iff q \preceq p$.

Not every graph is a CFG. In a **state machine** a cycle is checkable: after `login` $\rightarrow$ `logout`, **no part of the session may be left behind**.[1]

[1] **[Binder'99]** R. V. Binder. "*Testing Object-Oriented Systems: Models, Patterns, and Tools.*" **[ISSRE'12]** N. E. Holt, R. Torkar, L. Briand, K. Hansen. "*State-Based Testing.*"

Prime Path and Round-Trip Coverage

Definition (Prime Path Coverage (PPC))

$R_G = \{p \in P_G \mid p \text{ is a prime path}\}$, and $p \sim q \iff q \preceq p$.

Not every graph is a CFG. In a **state machine** a cycle is checkable: after `login` $\rightarrow$ `logout`, **no part of the session may be left behind**.[1]

Definition (Round-Trip Path)

A **round-trip path** is a prime path that starts and ends at the **same** node.

[1] **[Binder'99]** R. V. Binder. "*Testing Object-Oriented Systems: Models, Patterns, and Tools.*" **[ISSRE'12]** N. E. Holt, R. Torkar, L. Briand, K. Hansen. "*State-Based Testing.*"

Prime Path and Round-Trip Coverage

Definition (Prime Path Coverage (PPC))

$R_G = \{p \in P_G \mid p \text{ is a prime path}\}$, and $p \sim q \iff q \preceq p$.

Not every graph is a CFG. In a **state machine** a cycle is checkable: after `login` $\rightarrow$ `logout`, **no part of the session may be left behind**.[1]

Definition (Round-Trip Path)

A **round-trip path** is a prime path that starts and ends at the **same** node.

- **Complete round trip coverage (CRTC)** requires **every** round-trip path; **simple round trip coverage (SRTC)** requires **one** per node that has any.

[1] **[Binder'99]** R. V. Binder. "*Testing Object-Oriented Systems: Models, Patterns, and Tools.*" **[ISSRE'12]** N. E. Holt, R. Torkar, L. Briand, K. Hansen. "*State-Based Testing.*"

Prime Path and Round-Trip Coverage

Definition (Prime Path Coverage (PPC))

$R_G = \{p \in P_G \mid p \text{ is a prime path}\}$, and $p \sim q \iff q \preceq p$.

Not every graph is a CFG. In a **state machine** a cycle is checkable: after `login` → `logout`, **no part of the session may be left behind**.[1]

Definition (Round-Trip Path)

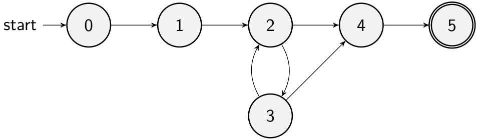
A **round-trip path** is a prime path that starts and ends at the **same** node.

- **Complete round trip coverage (CRTC)** requires **every** round-trip path; **simple round trip coverage (SRTC)** requires **one** per node that has any.
- On a **CFG** a node is not a state you can inspect, so these buy nothing over prime paths.

[1] **[Binder'99]** R. V. Binder. "*Testing Object-Oriented Systems: Models, Patterns, and Tools.*" **[ISSRE'12]** N. E. Holt, R. Torkar, L. Briand, K. Hansen. "*State-Based Testing.*"

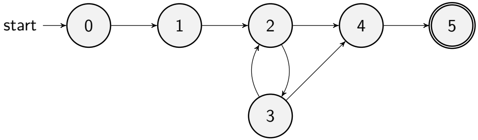
Touring, Sidetrips, and Detours

Prime paths have **no internal loops** – but a real execution may take one on the way. Must we reject it?



Touring, Sidetrips, and Detours

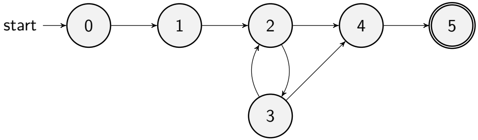
Prime paths have **no internal loops** – but a real execution may take one on the way. Must we reject it?



- p **tours** q if $q \preceq p$. $[0, 1, 2, 4, 5]$ tours $[1, 2, 4]$

Touring, Sidetrips, and Detours

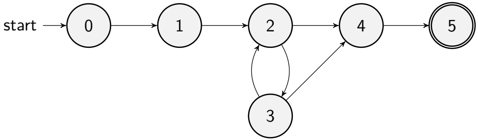
Prime paths have **no internal loops** – but a real execution may take one on the way. Must we reject it?



- p **tours** q if $q \preceq p$. $[0, 1, 2, 4, 5]$ tours $[1, 2, 4]$
- p tours q **with sidetrips** if every **edge** of q is in p in the same order.
  $[0, 1, 2, 3, 2, 4, 5]$

Touring, Sidetrips, and Detours

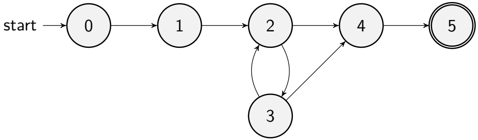
Prime paths have **no internal loops** – but a real execution may take one on the way. Must we reject it?



- p **tours** q if $q \preceq p$. $[0, 1, 2, 4, 5]$ tours $[1, 2, 4]$
- p tours q **with sidetrips** if every **edge** of q is in p in the same order. $[0, 1, 2, 3, 2, 4, 5]$
- p tours q **with detours** if every **node** of q is in p in the same order. $[0, 1, 2, 3, 4, 5]$

Touring, Sidetrips, and Detours

Prime paths have **no internal loops** – but a real execution may take one on the way. Must we reject it?



- p **tours** q if $q \preceq p$. $[0, 1, 2, 4, 5]$ tours $[1, 2, 4]$
- p tours q **with sidetrips** if every **edge** of q is in p in the same order. $[0, 1, 2, \textcolor{red}{3, 2}, 4, 5]$
- p tours q **with detours** if every **node** of q is in p in the same order. $[0, 1, 2, \textcolor{red}{3}, 4, 5]$
- A detour may **skip the edge under test** – $(2, 4)$ is never taken above. That is why sidetrips are the usual compromise.

Infeasibility

- A test requirement is **infeasible** if **no input** satisfies it: dead code, or a subpath that needs contradictory conditions (`if (x > 0)` inside `if (x < 0)`).

Infeasibility

- A test requirement is **infeasible** if **no input** satisfies it: dead code, or a subpath that needs contradictory conditions (`if (x > 0)` inside `if (x < 0)`).

- Deciding feasibility is **undecidable** in general – so "100% coverage" is usually **unreachable**, and we cannot even tell how unreachable.

Infeasibility

- A test requirement is **infeasible** if **no input** satisfies it: dead code, or a subpath that needs contradictory conditions (`if (x > 0)` inside `if (x < 0)`).

- Deciding feasibility is **undecidable** in general – so “100% coverage” is usually **unreachable**, and we cannot even tell how unreachable.

- The **stronger** the criterion, the more infeasible TRs: PPC on a loop whose bound is a constant, EPC on a branch that never repeats. Disallowing sidetrips makes it worse.

Infeasibility

- A test requirement is **infeasible** if **no input** satisfies it: dead code, or a subpath that needs contradictory conditions (`if (x > 0)` inside `if (x < 0)`).

- Deciding feasibility is **undecidable** in general – so "100% coverage" is usually **unreachable**, and we cannot even tell how unreachable.

- The **stronger** the criterion, the more infeasible TRs: PPC on a loop whose bound is a constant, EPC on a branch that never repeats. Disallowing sidetrips makes it worse.

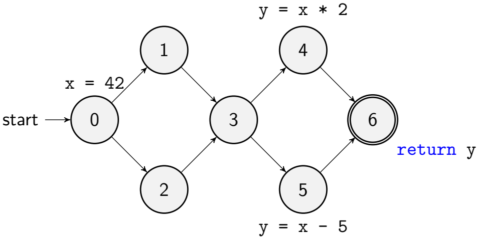
- **In practice**: satisfy what you can **without** sidetrips, then **allow** sidetrips for the rest, then inspect the remainder by hand.

Definitions and Uses

Structural criteria ask **where control went**. Data-flow criteria ask whether a computed **value** was ever **used**.

Definitions and Uses

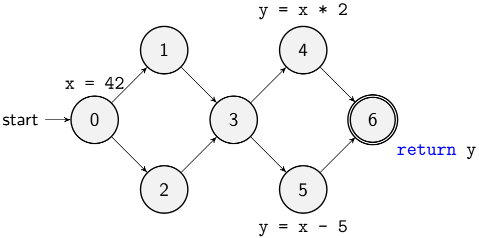
Structural criteria ask **where control went**. Data-flow criteria ask whether a computed **value** was ever **used**.



$$
\begin{array}{ll}
\mathit{def}(0) = \{x\} & \mathit{use}(4) = \{x\} \\
\mathit{def}(4) = \{y\} & \mathit{use}(5) = \{x\} \\
\mathit{def}(5) = \{y\} & \mathit{use}(6) = \{y\}
\end{array}
$$

Definitions and Uses

Structural criteria ask **where control went**. Data-flow criteria ask whether a computed **value** was ever **used**.

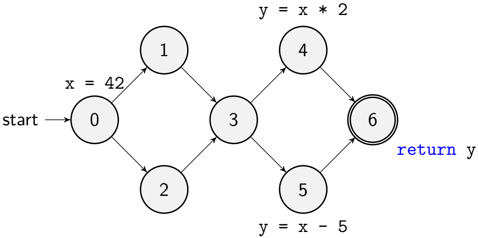


$$
\begin{array}{ll}
\mathit{def}(0) = \{x\} & \mathit{use}(4) = \{x\} \\
\mathit{def}(4) = \{y\} & \mathit{use}(5) = \{x\} \\
\mathit{def}(5) = \{y\} & \mathit{use}(6) = \{y\}
\end{array}
$$

- A **def** of a variable is a location where it is **assigned**; a **use** is a location where its value is **read**.

Definitions and Uses

Structural criteria ask **where control went**. Data-flow criteria ask whether a computed **value** was ever **used**.



$$
\begin{array}{ll}
\mathit{def}(0) = \{x\} & \mathit{use}(4) = \{x\} \\
\mathit{def}(4) = \{y\} & \mathit{use}(5) = \{x\} \\
\mathit{def}(5) = \{y\} & \mathit{use}(6) = \{y\}
\end{array}
$$

- A **def** of a variable is a location where it is **assigned**; a **use** is a location where its value is **read**.
- Node coverage reaches node 4. It never checks that the 42 written at node 0 is the value node 4 reads.

DU-Pairs and DU-Paths

Definition (DU-Pair)

A **du-pair** for x is a pair of locations (l, l') such that x is defined at l and used at l'.

DU-Pairs and DU-Paths

PLRG

Definition (DU-Pair)

A **du-pair** for x is a pair of locations (l, l') such that x is defined at l and used at l'.

Definition (Def-Clear Path)

A path from l to l' is **def-clear** with respect to x if x is **not** redefined on any node or edge strictly between them.

DU-Pairs and DU-Paths

PLRG

Definition (DU-Pair)

A **du-pair** for x is a pair of locations (l, l') such that x is defined at l and used at l'.

Definition (Def-Clear Path)

A path from l to l' is **def-clear** with respect to x if x is **not** redefined on any node or edge strictly between them.

Definition (DU-Path)

A **du-path** for x is a **simple** path from a def of x to a use of x that is **def-clear** with respect to x.

DU-Pairs and DU-Paths

PLRG

Definition (DU-Pair)

A **du-pair** for x is a pair of locations (l, l') such that x is defined at l and used at l'.

Definition (Def-Clear Path)

A path from l to l' is **def-clear** with respect to x if x is **not** redefined on any node or edge strictly between them.

Definition (DU-Path)

A **du-path** for x is a **simple** path from a def of x to a use of x that is **def-clear** with respect to x.

- $du(n, n', x)$ – du-paths from n to n'; $\quad du(n, x) = \bigcup_{n'} du(n, n', x)$

DU-Pairs and DU-Paths

Definition (DU-Pair)

A **du-pair** for x is a pair of locations (l, l') such that x is defined at l and used at l'.

Definition (Def-Clear Path)

A path from l to l' is **def-clear** with respect to x if x is **not** redefined on any node or edge strictly between them.

Definition (DU-Path)

A **du-path** for x is a **simple** path from a def of x to a use of x that is **def-clear** with respect to x.

- $du(n, n', x)$ – du-paths from n to n'; $\quad du(n, x) = \bigcup_{n'} du(n, n', x)$
- A test path **du-tours** a du-path if it tours it **and** the subpath taken stays def-clear – sidetrips are allowed only if they do not redefine x.

Three Data-Flow Criteria

One def, one use, and many paths between them. The three criteria differ exactly on **how many of those paths** you must take.

Three Data-Flow Criteria

One def, one use, and many paths between them. The three criteria differ exactly on **how many of those paths** you must take.

	test requirements R_G	p **covers it by du-touring**
ADC *(all-defs)*	(n, x)	**some** $q \in du(n, x)$
AUC *(all-uses)*	(n, n', x)	**some** $q \in du(n, n', x)$
ADUPC *(all-du-paths)*	**every** du-path q	q itself

only those with at least one du-path ($du(\cdot) \neq \emptyset$)

Three Data-Flow Criteria

One def, one use, and many paths between them. The three criteria differ exactly on **how many of those paths** you must take.

	test requirements R_G	p **covers it by du-touring**
ADC *(all-defs)*	(n, x)	**some** $q \in du(n, x)$
AUC *(all-uses)*	(n, n', x)	**some** $q \in du(n, n', x)$
ADUPC *(all-du-paths)*	**every** du-path q	q itself

only those with at least one du-path ($du(\cdot) \neq \emptyset$)

- ADC: every def reaches **some** use. AUC: every def reaches **every** use it can. ADUPC: and by **every route**.

Three Data-Flow Criteria

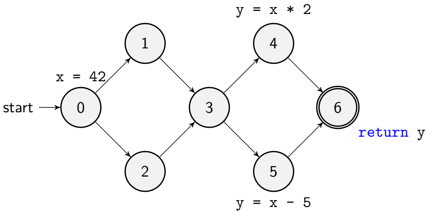
One def, one use, and many paths between them. The three criteria differ exactly on **how many of those paths** you must take.

	test requirements R_G	p **covers it by du-touring**
ADC *(all-defs)*	(n, x)	**some** $q \in du(n, x)$
AUC *(all-uses)*	(n, n', x)	**some** $q \in du(n, n', x)$
ADUPC *(all-du-paths)*	**every** du-path q	q itself

only those with at least one du-path ($du(\cdot) \neq \emptyset$)

- ADC: every def reaches **some** use. AUC: every def reaches **every** use it can. ADUPC: and by **every route**.
- So ADUPC is the **strongest** of the three and ADC the **weakest**. When a tool says "data-flow coverage" it almost always means **AUC** – the other two are too weak and too expensive respectively.

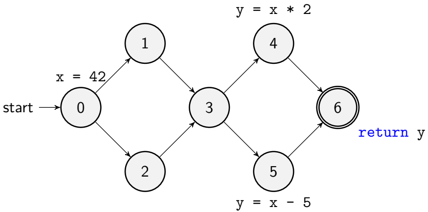
Data-Flow Coverage – Example



- **All-Defs Coverage (ADC)** – one du-path out of each of the three defs

$$R_G = \{(0, x), (4, y), (5, y)\} \qquad \text{Test Paths} = \{[0, 1, 3, 4, 6], [0, 1, 3, 5, 6]\}$$

Data-Flow Coverage – Example



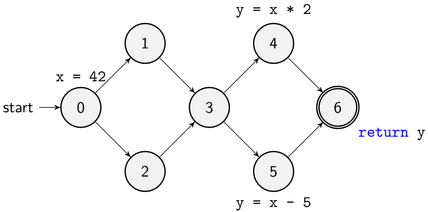
- **All-Defs Coverage (ADC)** – one du-path out of each of the three defs

  $R_G = \{(0, x), (4, y), (5, y)\}$ Test Paths $= \{[0, 1, 3, 4, 6], [0, 1, 3, 5, 6]\}$

- **All-Uses Coverage (AUC)** – every (def, use) pair

  $R_G = \{(0, 4, x), (0, 5, x), (4, 6, y), (5, 6, y)\}$
  Test Paths $= \{[0, 1, 3, 4, 6], [0, 1, 3, 5, 6]\}$

Data-Flow Coverage – Example



- **All-Defs Coverage (ADC)** – one du-path out of each of the three defs

  $R_G = \{(0, x), (4, y), (5, y)\}$ Test Paths $= \{[0, 1, 3, 4, 6], [0, 1, 3, 5, 6]\}$

- **All-Uses Coverage (AUC)** – every (def, use) pair

  $$R_G = \{(0, 4, x), (0, 5, x), (4, 6, y), (5, 6, y)\}$$
  $$\text{Test Paths} = \{[0, 1, 3, 4, 6], [0, 1, 3, 5, 6]\}$$

- **All-DU-Paths Coverage (ADUPC)** – both routes through nodes 1 and 2

  $$R_G = \{[0, 1, 3, 4], [0, 1, 3, 5], [0, 2, 3, 4], [0, 2, 3, 5], [4, 6], [5, 6]\}$$
  $$\text{Test Paths} = \{[0, 1, 3, 4, 6], [0, 1, 3, 5, 6], [0, 2, 3, 4, 6], [0, 2, 3, 5, 6]\}$$

Subsumption

Definition (Subsumption)

C_G **subsumes** C'_G if **every** test suite satisfying C_G also satisfies C'_G.

Subsumption

Definition (Subsumption)

C_G **subsumes** C'_G if **every** test suite satisfying C_G also satisfies C'_G.

- EC subsumes NC; EPC subsumes EC; PPC subsumes EC.
  On the data-flow side, ADUPC subsumes AUC subsumes ADC.

Subsumption

Definition (Subsumption)

C_G **subsumes** C'_G if **every** test suite satisfying C_G also satisfies C'_G.

- EC subsumes NC; EPC subsumes EC; PPC subsumes EC.
  On the data-flow side, ADUPC subsumes AUC subsumes ADC.
- But PPC does **not** subsume EPC: if n has a self-loop, EPC requires $[n, n, m]$, which is not simple and so never a prime path.

Subsumption

PLRG

Definition (Subsumption)

C_G **subsumes** C'_G if **every** test suite satisfying C_G also satisfies C'_G.

- EC subsumes NC; EPC subsumes EC; PPC subsumes EC.
  On the data-flow side, ADUPC subsumes AUC subsumes ADC.
- But PPC does **not** subsume EPC: if n has a self-loop, EPC requires $[n, n, m]$, which is not simple and so never a prime path.
- CRTC and SRTC subsume neither – they see only loops.

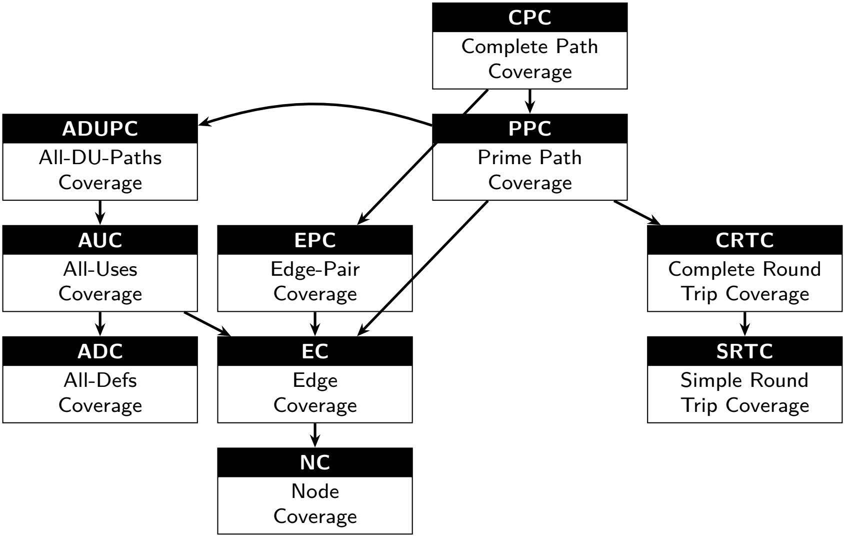
Subsumption

Definition (Subsumption)

C_G **subsumes** C'_G if **every** test suite satisfying C_G also satisfies C'_G.

- EC subsumes NC; EPC subsumes EC; PPC subsumes EC.
  On the data-flow side, ADUPC subsumes AUC subsumes ADC.
- But PPC does **not** subsume EPC: if n has a self-loop, EPC requires $[n, n, m]$, which is not simple and so never a prime path.
- CRTC and SRTC subsume neither – they see only loops.
- Subsumption is about **guarantees**, not about **finding bugs**. A PPC-adequate suite is not automatically better at detecting faults than an EC-adequate one. We come back to this later.

Subsumption Relationships



Contents

Predicates and Clauses

$$p = \underbrace{(a < b)}_{\text{relational}} \lor \underbrace{f(z)}_{\text{boolean function}} \land \underbrace{D}_{\text{variable}} \land \underbrace{(m \geq n \times o)}_{\text{relational}}$$

Predicates and Clauses

$$p \;=\; \underbrace{(a < b)}_{\text{relational}} \lor \underbrace{f(z)}_{\text{boolean function}} \land \underbrace{D}_{\text{variable}} \land \underbrace{(m \geq n \times o)}_{\text{relational}}$$

- A **predicate** is an expression evaluating to a **boolean**; a **clause** is a predicate with **no logical operator** in it. The predicate above has **four** clauses.

Predicates and Clauses

$$p = \underbrace{(a < b)}_{\text{relational}} \vee \underbrace{f(z)}_{\text{boolean function}} \wedge \underbrace{D}_{\text{variable}} \wedge \underbrace{(m \geq n \times o)}_{\text{relational}}$$

- A **predicate** is an expression evaluating to a **boolean**; a **clause** is a predicate with **no logical operator** in it. The predicate above has **four** clauses.

- Operators: $\neg$, $\wedge$, $\vee$, $\oplus$, $\Rightarrow$, $\Leftrightarrow$.

Predicates and Clauses

$$p = \underbrace{(a < b)}_{\text{relational}} \lor \underbrace{f(z)}_{\text{boolean function}} \land \underbrace{D}_{\text{variable}} \land \underbrace{(m \geq n \times o)}_{\text{relational}}$$

- A **predicate** is an expression evaluating to a **boolean**; a **clause** is a predicate with **no logical operator** in it. The predicate above has **four** clauses.
- Operators: $\neg$, $\land$, $\lor$, $\oplus$, $\Rightarrow$, $\Leftrightarrow$.
- Node and edge coverage see **one bit** per `if` – which way it went. Logic coverage looks **inside** the condition.

Predicate and Clause Coverage

PLRG

Notation: P is the set of predicates, C_p the clauses of predicate p.

Predicate and Clause Coverage

Notation: P is the set of predicates, C_p the clauses of predicate p.

Definition (Predicate Coverage (PC))

For each $p \in P$, the TRs are p evaluating to **true** and to **false**. (Also called **decision coverage**; "branch coverage" in tools.)

Predicate and Clause Coverage

Notation: P is the set of predicates, C_p the clauses of predicate p.

Definition (Predicate Coverage (PC))

For each $p \in P$, the TRs are p evaluating to **true** and to **false**. (Also called **decision coverage**; "branch coverage" in tools.)

Definition (Clause Coverage (CC))

For each clause $c \in C_p$, the TRs are c evaluating to **true** and to **false**. (Also called **condition coverage**.)

Predicate and Clause Coverage

Notation: P is the set of predicates, C_p the clauses of predicate p.

Definition (Predicate Coverage (PC))

For each $p \in P$, the TRs are p evaluating to **true** and to **false**. (Also called **decision coverage**; “branch coverage” in tools.)

Definition (Clause Coverage (CC))

For each clause $c \in C_p$, the TRs are c evaluating to **true** and to **false**. (Also called **condition coverage**.)

PC asks **two** questions per predicate; CC asks **two per clause**. Both are cheap, and neither implies the other.

PC and CC – Example

$$p = ((a < b) \vee D) \wedge (m \geq n \times o)$$

PC and CC – Example

$$p = ((a < b) \vee D) \wedge (m \geq n \times o)$$

PC

a	b	D	m	n	o	p
5	10	T	1	1	1	T
10	5	F	1	1	1	F

CC

a	b	D	m	n	o	$(a < b)$	D	$(m \geq n \times o)$
5	10	F	1	1	1	T	F	T
10	5	T	1	2	2	F	T	F

PC and CC – Example

$$p = ((a < b) \vee D) \wedge (m \geq n \times o)$$

PC

a	b	D	m	n	o	p
5	10	T	1	1	1	T
10	5	F	1	1	1	F

CC

a	b	D	m	n	o	$(a < b)$	D	$(m \geq n \times o)$
5	10	F	1	1	1	T	F	T
10	5	T	1	2	2	F	T	F

- Two tests each. This CC suite **happens** to satisfy PC as well (p is `true` then `false`) – but that is luck, not a guarantee.

Neither One Is Enough

- **PC does not subsume CC**: one test pair can flip the predicate while leaving a clause always `false` – and with **short-circuit** evaluation, never even evaluating it.

Neither One Is Enough

- **PC does not subsume CC**: one test pair can flip the predicate while leaving a clause always `false` – and with **short-circuit** evaluation, never even evaluating it.

- **CC does not subsume PC**: this suite covers both values of A and of B, yet $A \vee B$ is `true` in both rows.

A	B	$A \vee B$
T	F	T
F	T	T

Neither One Is Enough

PLRG

- **PC does not subsume CC**: one test pair can flip the predicate while leaving a clause always `false` – and with **short-circuit** evaluation, never even evaluating it.

- **CC does not subsume PC**: this suite covers both values of A and of B, yet $A \vee B$ is `true` in both rows.

A	B	$A \vee B$
T	F	T
F	T	T

- The simplest fix is to require **every combination**.

Combinatorial Coverage

Definition (Combinatorial Coverage (CoC))

For each $p \in P$, the TRs are **all combinations** of truth values of the clauses in C_p.

$$p = ((a < b) \vee D) \wedge (m \geq n \times o)$$

$(a < b)$	D	$(m \geq n \times o)$	p
T	T	T	T
T	T	F	F
T	F	T	T
T	F	F	F
F	T	T	T
F	T	F	F
F	F	T	F
F	F	F	F

Combinatorial Coverage

PLRG

Definition (Combinatorial Coverage (CoC))

For each $p \in P$, the TRs are **all combinations** of truth values of the clauses in C_p.

$$p = ((a < b) \vee D) \wedge (m \geq n \times o)$$

$(a < b)$	D	$(m \geq n \times o)$	p
T	T	T	T
T	T	F	F
T	F	T	T
T	F	F	F
F	T	T	T
F	T	F	F
F	F	T	F
F	F	F	F

2^N tests for N clauses. We want CoC's **guarantee** at CC's **price**.

Determination

Definition (Determination)

A clause $c \in C_p$ – the **major clause** – **determines** p if the remaining **minor clauses** have values such that **changing the value of** c **changes the value of** p.

Determination

PLRG

Definition (Determination)

A clause $c \in C_p$ – the **major clause** – **determines** p if the remaining **minor clauses** have values such that **changing the value of** c **changes the value of** p.

So a pair of tests that differs **only in** c and gives a **different** p is direct evidence that c matters.

Determination

Definition (Determination)

A clause $c \in C_p$ – the **major clause** – **determines** p if the remaining **minor clauses** have values such that **changing the value of** c **changes the value of** p.

So a pair of tests that differs **only in** c and gives a **different** p is direct evidence that c matters.

A	B	$A \vee B$
T	T	T
T	F	T
F	T	T
F	F	F

A	B	$A \wedge B$
T	T	T
T	F	F
F	T	F
F	F	F

A	B	$A \oplus B$
T	T	F
T	F	T
F	T	T
F	F	F

A	B	$A \Leftrightarrow B$
T	T	T
T	F	F
F	T	F
F	F	T

Determination

Definition (Determination)

A clause $c \in C_p$ – the **major clause** – **determines** p if the remaining **minor clauses** have values such that **changing the value of** c **changes the value of** p.

So a pair of tests that differs **only in** c and gives a **different** p is direct evidence that c matters.

A	B	$A \vee B$
T	T	T
T	F	T
F	T	T
F	F	F

A	B	$A \wedge B$
T	T	T
T	F	F
F	T	F
F	F	F

A	B	$A \oplus B$
T	T	F
T	F	T
F	T	T
F	F	F

A	B	$A \Leftrightarrow B$
T	T	T
T	F	F
F	T	F
F	F	T

- A determines $A \vee B$ only when $B = \texttt{false}$, and $A \wedge B$ only when $B = \texttt{true}$.
- A **always** determines $A \oplus B$ and $A \Leftrightarrow B$ – the minor clause cannot mask it.

Active Clause Coverage (ACC)

Definition (Active Clause Coverage (ACC))

For each $p \in P$ and each major clause $c \in C_p$, the TRs are the minor values that make c **determine** p, paired with $c = \texttt{true}$ and $c = \texttt{false}$.

[2] **[SEJ'94]** J. J. Chilenski, S. P. Miller. "*Applicability of Modified Condition/Decision Coverage to Software Testing.*"

Active Clause Coverage (ACC)

PLRG

Definition (Active Clause Coverage (ACC))

For each $p \in P$ and each major clause $c \in C_p$, the TRs are the minor values that make c **determine** p, paired with $c = \texttt{true}$ and $c = \texttt{false}$.

- For $p = A \vee B$:
  1. $A = \texttt{true}$, $B = \texttt{false}$ (A determines p)
  2. $A = \texttt{false}$, $B = \texttt{false}$ (A determines p)
  3. $A = \texttt{false}$, $B = \texttt{true}$ (B determines p)
  4. ~~$A = \texttt{false}$, $B = \texttt{false}$~~ (same row as 2)

  $N + 1$ tests instead of 2^N: **three** rows, not four.

[2] **[SEJ'94]** J. J. Chilenski, S. P. Miller. "*Applicability of Modified Condition/Decision Coverage to Software Testing.*"

Active Clause Coverage (ACC)

Definition (Active Clause Coverage (ACC))

For each $p \in P$ and each major clause $c \in C_p$, the TRs are the minor values that make c **determine** p, paired with $c = \texttt{true}$ and $c = \texttt{false}$.

- For $p = A \vee B$:
  1. $A = \texttt{true}$, $B = \texttt{false}$ (A determines p)
  2. $A = \texttt{false}$, $B = \texttt{false}$ (A determines p)
  3. $A = \texttt{false}$, $B = \texttt{true}$ (B determines p)
  4. $\cancel{A = \texttt{false}, B = \texttt{false}}$ (same row as 2)

  $N + 1$ tests instead of 2^N: **three** rows, not four.

- ACC is better known as **modified condition/decision coverage (MC/DC)**. It is the criterion regulators require for the software that **flies aircraft**.[2]

[2] **[SEJ'94]** J. J. Chilenski, S. P. Miller. "*Applicability of Modified Condition/Decision Coverage to Software Testing.*"

One Definition, Three Criteria

Ambiguity: must the **minor** clauses keep the **same values** when the major clause flips?

$$p = A \vee (B \wedge C), \quad \text{major clause } A$$

A	B	C	p
T	F	T	T
F	F	F	F

is $C =$ `false` allowed here?

[3] **[ISSRE'03]** P. Ammann, J. Offutt, H. Huang. "*Coverage Criteria for Logical Expressions.*"

One Definition, Three Criteria

Ambiguity: must the **minor** clauses keep the **same values** when the major clause flips?

$$p = A \vee (B \wedge C), \quad \text{major clause } A$$

A	B	C	p
T	F	T	T
F	F	F	F

is C = `false` allowed here?

This question confused testers – and certification authorities – for years.

Three answers, three criteria:[3]

[3] **[ISSRE'03]** P. Ammann, J. Offutt, H. Huang. "*Coverage Criteria for Logical Expressions.*"

One Definition, Three Criteria

Ambiguity: must the **minor** clauses keep the **same values** when the major clause flips?

$$p = A \vee (B \wedge C), \quad \text{major clause } A$$

A	B	C	p
T	F	T	T
F	F	F	F

is $C =$ `false` allowed here?

This question confused testers – and certification authorities – for years.

Three answers, three criteria:[3]

- **General** active clause coverage (GACC)
- **Restricted** active clause coverage (RACC)
- **Correlated** active clause coverage (CACC)

[3] **[ISSRE'03]** P. Ammann, J. Offutt, H. Huang. "*Coverage Criteria for Logical Expressions.*"

Why GACC Is Too Weak

Definition (General Active Clause Coverage (GACC))

The minor clauses do **not need** to be the same when the major clause is `true` and when it is `false`.

Why GACC Is Too Weak

Definition (General Active Clause Coverage (GACC))

The minor clauses do **not need** to be the same when the major clause is `true` and when it is `false`.

- GACC does **not subsume predicate coverage**. For $A \Leftrightarrow B$, each clause always determines p, so these two rows satisfy GACC for both A and B – yet p is `true` in both.

A	B	$A \Leftrightarrow B$
T	T	T
F	F	T

Why GACC Is Too Weak

Definition (General Active Clause Coverage (GACC))

The minor clauses do **not need** to be the same when the major clause is `true` and when it is `false`.

- GACC does **not subsume predicate coverage**. For $A \Leftrightarrow B$, each clause always determines p, so these two rows satisfy GACC for both A and B – yet p is `true` in both.

A	B	$A \Leftrightarrow B$
T	T	T
F	F	T

- GACC was meant to be **stronger** than predicate coverage. It is not even as strong. So we need a different rule.

RACC vs. CACC

RACC

A	B	C	$A \wedge (B \vee C)$
T	T	T	T
F	T	T	F
T	T	F	T
F	T	F	F
T	F	T	T
F	F	T	F

CACC

A	B	C	$A \wedge (B \vee C)$
T	T	T	T
T	T	F	T
T	F	T	T
F	T	T	F
F	T	F	F
F	F	T	F

RACC vs. CACC

RACC

A	B	C	$A \wedge (B \vee C)$
T	T	T	T
F	T	T	F
T	T	F	T
F	T	F	F
T	F	T	T
F	F	T	F

CACC

A	B	C	$A \wedge (B \vee C)$
T	T	T	T
T	T	F	T
T	F	T	T
F	T	T	F
F	T	F	F
F	F	T	F

- **RACC** (*restricted*) pairs rows: B and C must be identical across the flip, so only **three** pairs are legal.

RACC vs. CACC

RACC

A	B	C	$A \wedge (B \vee C)$
T	T	T	T
F	T	T	F
T	T	F	T
F	T	F	F
T	F	T	T
F	F	T	F

CACC

A	B	C	$A \wedge (B \vee C)$
T	T	T	T
T	T	F	T
T	F	T	T
F	T	T	F
F	T	F	F
F	F	T	F

- **RACC** (*restricted*) pairs rows: B and C must be identical across the flip, so only **three** pairs are legal.
- **CACC** (*correlated*) only requires p to flip, so **any** top row pairs with **any** bottom row – **nine** choices.

RACC vs. CACC

PLRG

RACC

A	B	C	$A \land (B \lor C)$
T	T	T	T
F	T	T	F
T	T	F	T
F	T	F	F
T	F	T	T
F	F	T	F

CACC

A	B	C	$A \land (B \lor C)$
T	T	T	T
T	T	F	T
T	F	T	T
F	T	T	F
F	T	F	F
F	F	T	F

- **RACC** (*restricted*) pairs rows: B and C must be identical across the flip, so only **three** pairs are legal.
- **CACC** (*correlated*) only requires p to flip, so **any** top row pairs with **any** bottom row – **nine** choices.
- **RACC** is what the aviation industry adopted. **CACC** is the later reading: weaker, easier to satisfy, and still enough to subsume predicate coverage.

Inactive Clause Coverage (ICC)

Definition (Inactive Clause Coverage (ICC))

For each $p \in P$ and each major clause $c \in C_p$, the TRs are the minor values that make c **not determine** p, paired with $c =$ `true` and $c =$ `false`.

Inactive Clause Coverage (ICC)

Definition (Inactive Clause Coverage (ICC))

For each $p \in P$ and each major clause $c \in C_p$, the TRs are the minor values that make c **not determine** p, paired with $c = \texttt{true}$ and $c = \texttt{false}$.

The **dual** of ACC. ACC picks tests where c **must** change the answer; ICC picks tests where c **must not**.

Inactive Clause Coverage (ICC)

Definition (Inactive Clause Coverage (ICC))

For each $p \in P$ and each major clause $c \in C_p$, the TRs are the minor values that make c **not determine** p, paired with $c = \texttt{true}$ and $c = \texttt{false}$.

The **dual** of ACC. ACC picks tests where c **must** change the answer; ICC picks tests where c **must not**.

The **specification** says $p = A \vee B$, but code computes $A \oplus B$:

A	B	$A \vee B$	$A \oplus B$	
T	F	T	T	ACC
F	F	F	F	ACC
T	T	T	F	ICC
F	T	T	T	ICC

Inactive Clause Coverage (ICC)

Definition (Inactive Clause Coverage (ICC))

For each $p \in P$ and each major clause $c \in C_p$, the TRs are the minor values that make c **not determine** p, paired with $c = \texttt{true}$ and $c = \texttt{false}$.

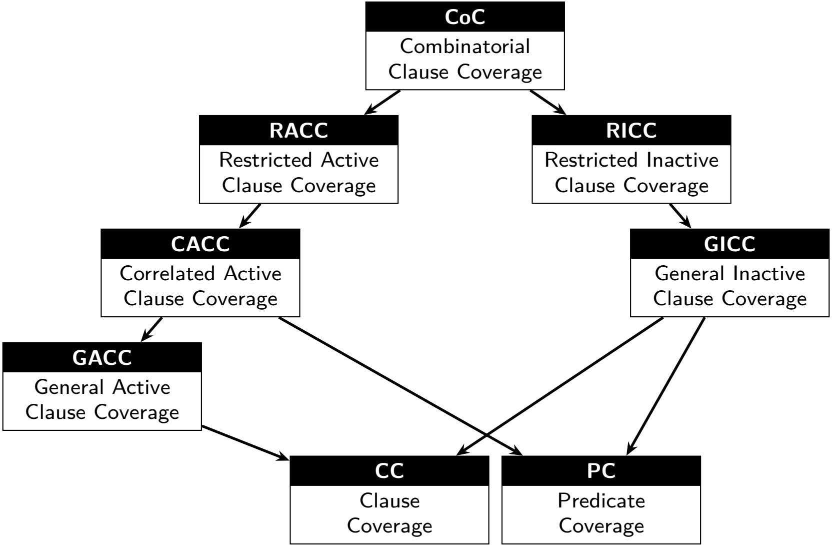
The **dual** of ACC. ACC picks tests where c **must** change the answer; ICC picks tests where c **must not**.

The **specification** says $p = A \vee B$, but code computes $A \oplus B$:

A	B	$A \vee B$	$A \oplus B$	
T	F	T	T	ACC
F	F	F	F	ACC
T	T	T	F	ICC
F	T	T	T	ICC

- Same ambiguity: **GICC** / **RICC**. No “correlated” variant – CACC asks the minor clauses to make p **flip** with c, and under ICC p never flips.

Subsumption Relationships



Contents

From a Criterion to a Number

$$\text{coverage}(T, C_G) \;=\; \frac{|\{r \in R_G \mid \exists t \in T.\ t \sim r\}|}{|R_G|}$$

From a Criterion to a Number

$$\text{coverage}(T, C_G) \;=\; \frac{|\{r \in R_G \mid \exists t \in T.\, t \sim r\}|}{|R_G|}$$

- A criterion is **satisfied** or not; a **percentage** is what people actually report. Every criterion above gives one.

From a Criterion to a Number

$$\text{coverage}(T, C_G) \;=\; \frac{|\{r \in R_G \mid \exists t \in T.\, t \sim r\}|}{|R_G|}$$

- A criterion is **satisfied** or not; a **percentage** is what people actually report. Every criterion above gives one.
- **Infeasible** TRs stay in the denominator, so 100% is usually out of reach – and we do not know what the real maximum is. Teams either remove those TRs by hand or pick a lower target such as 85%.

From a Criterion to a Number

$$\text{coverage}(T, C_G) = \frac{|\{r \in R_G \mid \exists t \in T.\ t \sim r\}|}{|R_G|}$$

- A criterion is **satisfied** or not; a **percentage** is what people actually report. Every criterion above gives one.
- **Infeasible** TRs stay in the denominator, so 100% is usually out of reach – and we do not know what the real maximum is. Teams either remove those TRs by hand or pick a lower target such as 85%.
- The denominator is **not stable**: adding a line changes $|R_G|$, so the percentage moves without anyone writing a test.

From a Criterion to a Number

$$\text{coverage}(T, C_G) \;=\; \frac{|\{r \in R_G \mid \exists t \in T.\, t \sim r\}|}{|R_G|}$$

- A criterion is **satisfied** or not; a **percentage** is what people actually report. Every criterion above gives one.
- **Infeasible** TRs stay in the denominator, so 100% is usually out of reach – and we do not know what the real maximum is. Teams either remove those TRs by hand or pick a lower target such as 85%.
- The denominator is **not stable**: adding a line changes $|R_G|$, so the percentage moves without anyone writing a test.
- Three questions remain: **where** do the numbers come from, **what** do real tools count, and **what does the number mean**?

Instrumentation

To know that node 4 ran, the program must **say so**. Insert probes:

```
int f(int a, int b) {
  cov[0] = 1;  int x = 42;
  if (a > 0) { cov[1] = 1; g(); }
  else       { cov[2] = 1; h(); }
  cov[3] = 1;  int y;
  if (b > 0) { cov[4] = 1; y = x * 2; }
  else       { cov[5] = 1; y = x - 5; }
  cov[6] = 1;  return y;
}
```

[4] **[ISSTA'02]** M. M. Tikir, J. K. Hollingsworth. "*Efficient Instrumentation for Code Coverage Testing.*"

Instrumentation

PLRG

To know that node 4 ran, the program must **say so**. Insert probes:

```
int f(int a, int b) {
  cov[0] = 1;  int x = 42;
  if (a > 0) { cov[1] = 1; g(); }
  else       { cov[2] = 1; h(); }
  cov[3] = 1;  int y;
  if (b > 0) { cov[4] = 1; y = x * 2; }
  else       { cov[5] = 1; y = x - 5; }
  cov[6] = 1;  return y;
}
```

- Probes can go in at the **source**, **IR**, **bytecode**, or **binary**: later is truer to what runs, earlier is easier to map back to lines.

[4] **[ISSTA'02]** M. M. Tikir, J. K. Hollingsworth. "*Efficient Instrumentation for Code Coverage Testing.*"

Instrumentation

To know that node 4 ran, the program must **say so**. Insert probes:

```
int f(int a, int b) {
  cov[0] = 1;  int x = 42;
  if (a > 0) { cov[1] = 1; g(); }
  else       { cov[2] = 1; h(); }
  cov[3] = 1;  int y;
  if (b > 0) { cov[4] = 1; y = x * 2; }
  else       { cov[5] = 1; y = x - 5; }
  cov[6] = 1;  return y;
}
```

- Probes can go in at the **source**, **IR**, **bytecode**, or **binary**: later is truer to what runs, earlier is easier to map back to lines.
- You do not need a probe on every node. If node 6 **post-dominates** node 3, one probe tells you about both.

[4] **[ISSTA'02]** M. M. Tikir, J. K. Hollingsworth. "*Efficient Instrumentation for Code Coverage Testing.*"

Instrumentation

To know that node 4 ran, the program must **say so**. Insert probes:

```
int f(int a, int b) {
  cov[0] = 1;  int x = 42;
  if (a > 0) { cov[1] = 1; g(); }
  else       { cov[2] = 1; h(); }
  cov[3] = 1;  int y;
  if (b > 0) { cov[4] = 1; y = x * 2; }
  else       { cov[5] = 1; y = x - 5; }
  cov[6] = 1;  return y;
}
```

- Probes can go in at the **source**, **IR**, **bytecode**, or **binary**: later is truer to what runs, earlier is easier to map back to lines.
- You do not need a probe on every node. If node 6 **post-dominates** node 3, one probe tells you about both.
- Coverage asks **whether** a block ran, not how often – so a probe can be **removed** once it fires.[4]

[4] **[ISSTA'02]** M. M. Tikir, J. K. Hollingsworth. "*Efficient Instrumentation for Code Coverage Testing.*"

What Real Tools Count

- Everything shipped – **gcov**, **llvm-cov**, **JaCoCo**, **Istanbul** – measures **node or edge** coverage. Prime paths and all-du-paths live in textbooks and research tools, not in your CI.

What Real Tools Count

- Everything shipped – **gcov**, **llvm-cov**, **JaCoCo**, **Istanbul** – measures **node or edge** coverage. Prime paths and all-du-paths live in textbooks and research tools, not in your CI.

- "Branch coverage" in a tool means **predicate coverage** on each decision – not edge coverage of the whole CFG, and not MC/DC.

What Real Tools Count

- Everything shipped – **gcov**, **llvm-cov**, **JaCoCo**, **Istanbul** – measures **node or edge** coverage. Prime paths and all-du-paths live in textbooks and research tools, not in your CI.

- "Branch coverage" in a tool means **predicate coverage** on each decision – not edge coverage of the whole CFG, and not MC/DC.

- A fuzzer goes further: AFL **hashes** edges into 64K buckets, so two edges can share one. It wants a **fast** signal, not an exact one – the criterion follows from the **purpose**.

Contents

100% Coverage Is Not Correctness

```
TEST(FibTest, Smoke) {
  fib(10);            // 100% line and branch coverage of fib
}
```

100% Coverage Is Not Correctness

```
TEST(FibTest, Smoke) {
  fib(10);            // 100% line and branch coverage of fib
}
```

- Coverage measures what was **executed**, never what was **checked**. The **oracle** is a separate problem, and a later topic.

100% Coverage Is Not Correctness

```
TEST(FibTest, Smoke) {
  fib(10);            // 100% line and branch coverage of fib
}
```

- Coverage measures what was **executed**, never what was **checked**. The **oracle** is a separate problem, and a later topic.
- Even with an oracle: a fault must be **reached**, the state must be **infected**, and the corruption must **propagate** to the output. Coverage certifies only the **first**.

100% Coverage Is Not Correctness

PLRG

```
TEST(FibTest, Smoke) {
  fib(10);            // 100% line and branch coverage of fib
}
```

- Coverage measures what was **executed**, never what was **checked**. The **oracle** is a separate problem, and a later topic.
- Even with an oracle: a fault must be **reached**, the state must be **infected**, and the corruption must **propagate** to the output. Coverage certifies only the **first**.
- **Necessary, not sufficient.** So how much does the necessary condition buy?

Then What Is It Good For?

- Bigger suites cover more **and** find more. Control for suite size and the correlation with fault detection is **weak** – and the studies still argue about how weak.[5]

  Low coverage is reliable bad news; high coverage is weak good news.

---

[5] **[ICSE'14]** L. Inozemtseva, R. Holmes. "*Coverage Is Not Strongly Correlated with Test Suite Effectiveness.*" **[ASE'20]** Y. T. Chen et al. "*Revisiting the Relationship between Fault Detection, Test Adequacy Criteria, and Test Set Size.*"

[6] **[FSE'19]** M. Ivanković, G. Petrović, R. Just, G. Fraser. "*Code Coverage at Google.*"

Then What Is It Good For?

- Bigger suites cover more **and** find more. Control for suite size and the correlation with fault detection is **weak** – and the studies still argue about how weak.[5]

  Low coverage is reliable bad news; high coverage is weak good news.

- **As a filter.** Google computes it for a billion lines a day and shows it on the **changed lines** of a code review – never as a quota.[6] Goodhart's law: a measure that becomes a target stops being a good measure.

[5] **[ICSE'14]** L. Inozemtseva, R. Holmes. "*Coverage Is Not Strongly Correlated with Test Suite Effectiveness.*" **[ASE'20]** Y. T. Chen et al. "*Revisiting the Relationship between Fault Detection, Test Adequacy Criteria, and Test Set Size.*"

[6] **[FSE'19]** M. Ivanković, G. Petrović, R. Just, G. Fraser. "*Code Coverage at Google.*"

Then What Is It Good For?

- Bigger suites cover more **and** find more. Control for suite size and the correlation with fault detection is **weak** – and the studies still argue about how weak.[5]

  Low coverage is reliable bad news; high coverage is weak good news.

- **As a filter.** Google computes it for a billion lines a day and shows it on the **changed lines** of a code review – never as a quota.[6] Goodhart's law: a measure that becomes a target stops being a good measure.

- **As a search gradient.** Last week's fuzzer, and techniques still to come. Nobody reports the percentage; it is only a signal.

[5] **[ICSE'14]** L. Inozemtseva, R. Holmes. "*Coverage Is Not Strongly Correlated with Test Suite Effectiveness.*" **[ASE'20]** Y. T. Chen et al. "*Revisiting the Relationship between Fault Detection, Test Adequacy Criteria, and Test Set Size.*"

[6] **[FSE'19]** M. Ivanković, G. Petrović, R. Just, G. Fraser. "*Code Coverage at Google.*"

Then What Is It Good For?

- Bigger suites cover more **and** find more. Control for suite size and the correlation with fault detection is **weak** – and the studies still argue about how weak.[5]

  Low coverage is reliable bad news; high coverage is weak good news.

- **As a filter.** Google computes it for a billion lines a day and shows it on the **changed lines** of a code review – never as a quota.[6] Goodhart's law: a measure that becomes a target stops being a good measure.

- **As a search gradient.** Last week's fuzzer, and techniques still to come. Nobody reports the percentage; it is only a signal.

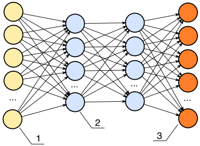
- **Not as a quality measure.** For that, seed faults and count kills – **mutation score**, which we take up later.

[5] **[ICSE'14]** L. Inozemtseva, R. Holmes. "*Coverage Is Not Strongly Correlated with Test Suite Effectiveness.*" **[ASE'20]** Y. T. Chen et al. "*Revisiting the Relationship between Fault Detection, Test Adequacy Criteria, and Test Set Size.*"

[6] **[FSE'19]** M. Ivanković, G. Petrović, R. Just, G. Fraser. "*Code Coverage at Google.*"

Contents

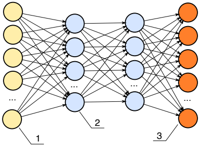
A Program With No Branches



- A DNN classifier is a function $f_\theta : X \to Y$ whose behavior lives in the **weights**, not in control flow. There is nothing to branch on – so cover the **neurons** instead. Write $f_\theta(n, x)$ for the output of neuron n on input x; the neuron is **activated** when that exceeds a threshold t.

[7] **[SOSP'17]** K. Pei, Y. Cao, J. Yang, S. Jana. "*DeepXplore.*" **[ASE'18]** L. Ma et al. "*DeepGauge.*"

A Program With No Branches



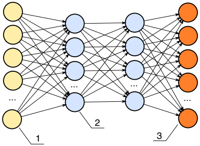
- A DNN classifier is a function $f_\theta : X \to Y$ whose behavior lives in the **weights**, not in control flow. There is nothing to branch on – so cover the **neurons** instead. Write $f_\theta(n, x)$ for the output of neuron n on input x; the neuron is **activated** when that exceeds a threshold t.

Definition (Neuron Coverage (NC))

$$NC(T, t) = \frac{|\{n \in N \mid \exists x \in T.\ f_\theta(n, x) > t\}|}{|N|}$$

[7] **[SOSP'17]** K. Pei, Y. Cao, J. Yang, S. Jana. "*DeepXplore.*" **[ASE'18]** L. Ma et al. "*DeepGauge.*"

A Program With No Branches



- A DNN classifier is a function $f_\theta : X \to Y$ whose behavior lives in the **weights**, not in control flow. There is nothing to branch on – so cover the **neurons** instead. Write $f_\theta(n, x)$ for the output of neuron n on input x; the neuron is **activated** when that exceeds a threshold t.

Definition (Neuron Coverage (NC))

$$NC(T, t) = \frac{|\{n \in N \mid \exists x \in T.\ f_\theta(n, x) > t\}|}{|N|}$$

Plain NC **saturates**, so it was refined by splitting each neuron's training range into k sections (**KMNC**), or looking outside that range (**NBC**), or taking the top k per layer (**TKNC**) – **the same k knob again**.[7]

[7] **[SOSP'17]** K. Pei, Y. Cao, J. Yang, S. Jana. "*DeepXplore.*" **[ASE'18]** L. Ma et al. "*DeepGauge.*"

Does Neuron Coverage Work?

[8] **[FSE'20]** F. Harel-Canada, L. Wang, M. A. Gulzar, Q. Gu, M. Kim. "*Is Neuron Coverage a Meaningful Measure for Testing Deep Neural Networks?*"

[9] **[FSE'20]** S. Yan, G. Tao, X. Liu, J. Zhai, S. Ma, L. Xu, X. Zhang. "*Correlations between Deep Neural Network Model Coverage Criteria and Model Quality.*"

[10] **[ICSE'19]** J. Kim, R. Feldt, S. Yoo. "*Guiding Deep Learning System Testing Using Surprise Adequacy.*"

Does Neuron Coverage Work?

Unfortunately, no!

[8] **[FSE'20]** F. Harel-Canada, L. Wang, M. A. Gulzar, Q. Gu, M. Kim. "*Is Neuron Coverage a Meaningful Measure for Testing Deep Neural Networks?*"

[9] **[FSE'20]** S. Yan, G. Tao, X. Liu, J. Zhai, S. Ma, L. Xu, X. Zhang. "*Correlations between Deep Neural Network Model Coverage Criteria and Model Quality.*"

[10] **[ICSE'19]** J. Kim, R. Feldt, S. Yoo. "*Guiding Deep Learning System Testing Using Surprise Adequacy.*"

Does Neuron Coverage Work?

Unfortunately, no!

- Generate inputs that **maximize** NC and see what you get: **fewer** defects detected, **less natural** inputs, and **more biased** predictions. Higher NC made the suite **worse**.[8]

[8] **[FSE'20]** F. Harel-Canada, L. Wang, M. A. Gulzar, Q. Gu, M. Kim. "*Is Neuron Coverage a Meaningful Measure for Testing Deep Neural Networks?*"

[9] **[FSE'20]** S. Yan, G. Tao, X. Liu, J. Zhai, S. Ma, L. Xu, X. Zhang. "*Correlations between Deep Neural Network Model Coverage Criteria and Model Quality.*"

[10] **[ICSE'19]** J. Kim, R. Feldt, S. Yoo. "*Guiding Deep Learning System Testing Using Surprise Adequacy.*"

Does Neuron Coverage Work?

Unfortunately, no!

- Generate inputs that **maximize** NC and see what you get: **fewer** defects detected, **less natural** inputs, and **more biased** predictions. Higher NC made the suite **worse**.[8]
- Independently, the criteria correlate poorly with model quality.[9]

[8] **[FSE'20]** F. Harel-Canada, L. Wang, M. A. Gulzar, Q. Gu, M. Kim. "*Is Neuron Coverage a Meaningful Measure for Testing Deep Neural Networks?*"

[9] **[FSE'20]** S. Yan, G. Tao, X. Liu, J. Zhai, S. Ma, L. Xu, X. Zhang. "*Correlations between Deep Neural Network Model Coverage Criteria and Model Quality.*"

[10] **[ICSE'19]** J. Kim, R. Feldt, S. Yoo. "*Guiding Deep Learning System Testing Using Surprise Adequacy.*"

Does Neuron Coverage Work?

Unfortunately, no!

- Generate inputs that **maximize** NC and see what you get: **fewer** defects detected, **less natural** inputs, and **more biased** predictions. Higher NC made the suite **worse**.[8]

- Independently, the criteria correlate poorly with model quality.[9]

- A better-founded alternative measures **surprise**: how far an input's activation trace is from the traces seen during training. It correlates with misclassification and gives a usable sampling target.[10]

---

[8] **[FSE'20]** F. Harel-Canada, L. Wang, M. A. Gulzar, Q. Gu, M. Kim. "*Is Neuron Coverage a Meaningful Measure for Testing Deep Neural Networks?*"

[9] **[FSE'20]** S. Yan, G. Tao, X. Liu, J. Zhai, S. Ma, L. Xu, X. Zhang. "*Correlations between Deep Neural Network Model Coverage Criteria and Model Quality.*"

[10] **[ICSE'19]** J. Kim, R. Feldt, S. Yoo. "*Guiding Deep Learning System Testing Using Surprise Adequacy.*"

Does Neuron Coverage Work?

Unfortunately, no!

- Generate inputs that **maximize** NC and see what you get: **fewer** defects detected, **less natural** inputs, and **more biased** predictions. Higher NC made the suite **worse**.[8]
- Independently, the criteria correlate poorly with model quality.[9]
- A better-founded alternative measures **surprise**: how far an input's activation trace is from the traces seen during training. It correlates with misclassification and gives a usable sampling target.[10]
- Same story as code coverage: **a criterion has to be validated, not just defined.**

[8] **[FSE'20]** F. Harel-Canada, L. Wang, M. A. Gulzar, Q. Gu, M. Kim. "*Is Neuron Coverage a Meaningful Measure for Testing Deep Neural Networks?*"

[9] **[FSE'20]** S. Yan, G. Tao, X. Liu, J. Zhai, S. Ma, L. Xu, X. Zhang. "*Correlations between Deep Neural Network Model Coverage Criteria and Model Quality.*"

[10] **[ICSE'19]** J. Kim, R. Feldt, S. Yoo. "*Guiding Deep Learning System Testing Using Surprise Adequacy.*"

The Spec Is a Program

PLRG

```
a = [10, 20, 30];
a.at(-1); // 30
a.at(5);  // undefined
```

23.1.3.1 Array.prototype.at (*index*)

1. Let *O* be ? ToObject(*this* value).
2. Let *len* be ? LengthOfArrayLike(*O*).
3. Let *relativeIndex* be ? ToIntegerOrInfinity(*index*).
4. If *relativeIndex* ≥ 0, then
   a. Let *k* be *relativeIndex*.
5. Else,
   a. Let *k* be *len* + *relativeIndex*.
6. If $k < 0$ or $k \geq len$, return **undefined**.
7. Return ? Get(*O*, ! ToString($\mathbb{F}$(*k*))).

The Spec Is a Program

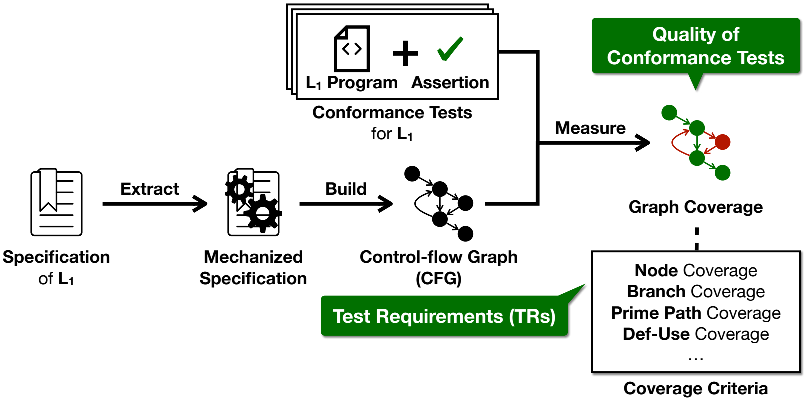
```
a = [10, 20, 30];
a.at(-1); // 30
a.at(5);  // undefined
```

23.1.3.1 Array.prototype.at (*index*)

1. Let *O* be ? ToObject(*this* value).
2. Let *len* be ? LengthOfArrayLike(*O*).
3. Let *relativeIndex* be ? ToIntegerOrInfinity(*index*).
4. If *relativeIndex* ≥ 0, then
   a. Let *k* be *relativeIndex*.
5. Else,
   a. Let *k* be *len* + *relativeIndex*.
6. If $k < 0$ or $k \geq$ *len*, return **undefined**.
7. Return ? Get(*O*, ! ToString($\mathbb{F}(k)$)).

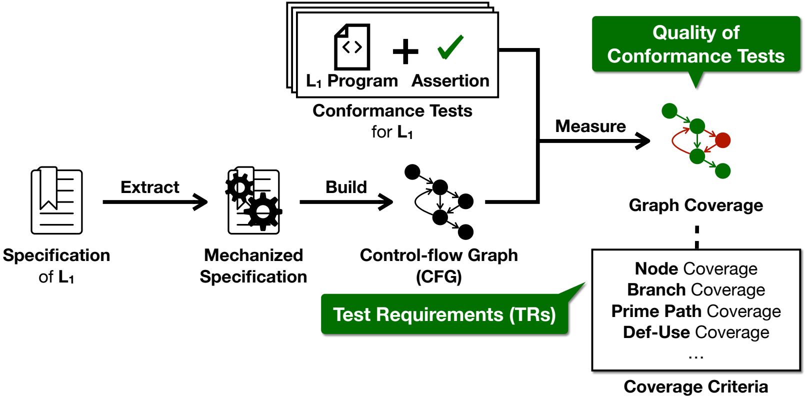
- ECMA-262 defines JavaScript as **pseudocode algorithms**. The two calls take **different paths** through this one: `at(-1)` runs $5 \to 7$, `at(5)` runs $4 \to 6$.

Covering the Specification



[11] **[ICSE'21]** J. Park, S. An, D. Youn, G. Kim, S. Ryu. "*JEST: N+1-Version Differential Testing of Both JavaScript Engines and Specification.*"

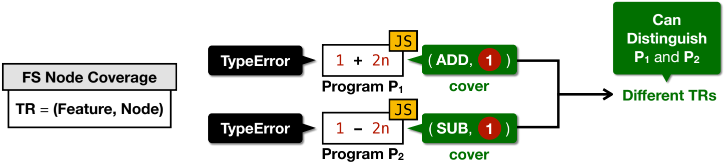
Covering the Specification



- **JEST**: 1,700 programs, **87.70%** of ES11 semantics – **44 engine bugs, 27 spec bugs**.[11]

[11] **[ICSE'21]** J. Park, S. An, D. Youn, G. Kim, S. Ryu. "*JEST: N+1-Version Differential Testing of Both JavaScript Engines and Specification.*"

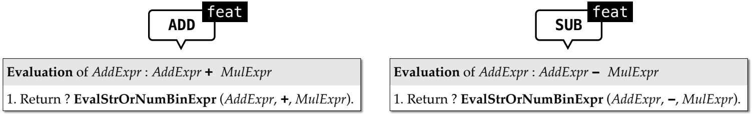
Feature-Sensitive Coverage



- **Feature-Sensitive (FS)** coverage criterion **divides** the given TRs with the **innermost enclosing** language **features**

FS Coverage

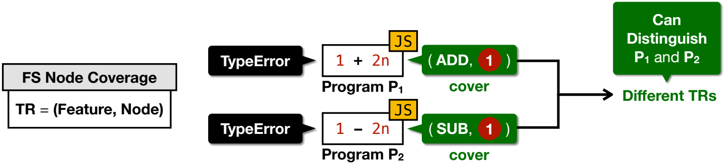
TR = (Feature, given TR)



[12] **[PLDI'23]** J. Park, D. Youn, K. Lee, S. Ryu. "*Feature-Sensitive Coverage for Conformance Testing of Programming Language Implementations.*"

[13] **[TOSEM'26]** K. Lee, S. Kim, J. Park, S. Ryu. "*Selective Feature-Sensitive Coverage for Conformance Testing of Programming Language Implementations.*"

Feature-Sensitive Coverage



- Feature-Sensitive (FS) coverage criterion **divides** the given TRs with the **innermost enclosing** language **features**

FS Coverage

TR = (Feature, given TR)

ADD feat

Evaluation of *AddExpr* : *AddExpr* **+** *MulExpr*

1. Return ? **EvalStrOrNumBinExpr** (*AddExpr*, **+**, *MulExpr*).

SUB feat

Evaluation of *AddExpr* : *AddExpr* **–** *MulExpr*

1. Return ? **EvalStrOrNumBinExpr** (*AddExpr*, **–**, *MulExpr*).

- **143** bugs in eight implementations.[12] Selective FS: **70.7%** fewer tests but preserves 53 of 57 bugs.[13]

[12] **[PLDI'23]** J. Park, D. Youn, K. Lee, S. Ryu. "*Feature-Sensitive Coverage for Conformance Testing of Programming Language Implementations.*"

[13] **[TOSEM'26]** K. Lee, S. Kim, J. Park, S. Ryu. "*Selective Feature-Sensitive Coverage for Conformance Testing of Programming Language Implementations.*"

Summary

1. Graph Coverage
    - Structural Coverage
    - Data-Flow Coverage
    - Subsumption Relationships
2. Logic Coverage
    - Simple Logic Expression Coverage
    - Active Clause Coverage
    - Inactive Clause Coverage
    - Subsumption Relationships
3. Measuring Coverage
4. Is Coverage a Good Criterion?
5. Coverage Beyond Code
    - Neuron Coverage
    - Feature-Sensitive Coverage

Homework #1 – Coverage Measurement

PLRG

- Build a **coverage profiler** that measures which parts of a program are executed by a test suite.
- Please see this document on GitHub:

  `https://github.com/ku-plrg-classroom/js-cov`

- The due date is **23:59 on September 21 (Mon.)**, and please submit it to **LMS**.

Next Lecture

- Search Based Software Testing (SBST)

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