Software Testing

Software Testing

(3104313)

Amirkabir University of Technology

Spring 1399-1400

Why do we test software?

- A tester's goal is to **eliminate faults as early as possible**

- Improve quality
- Reduce cost
- Preserve customer satisfaction

Why do we test software?

Software **quality** is increasingly becoming essential to all businesses

- Reliability
- Usability
- Security
- Availability, Scalability, Maintainability, Performance

What purpose(s) does a test serve?

Why Each Test

- What fact is each test trying to **verify**?
- Document test **objectives** and test **requirements**
- How much testing is **enough** and when testing will **complete**
- **Coverage** levels

Verification and Validation

Verification- The process of determining whether the products of a phase of the software development process fulfill the requirements established during the previous phase.

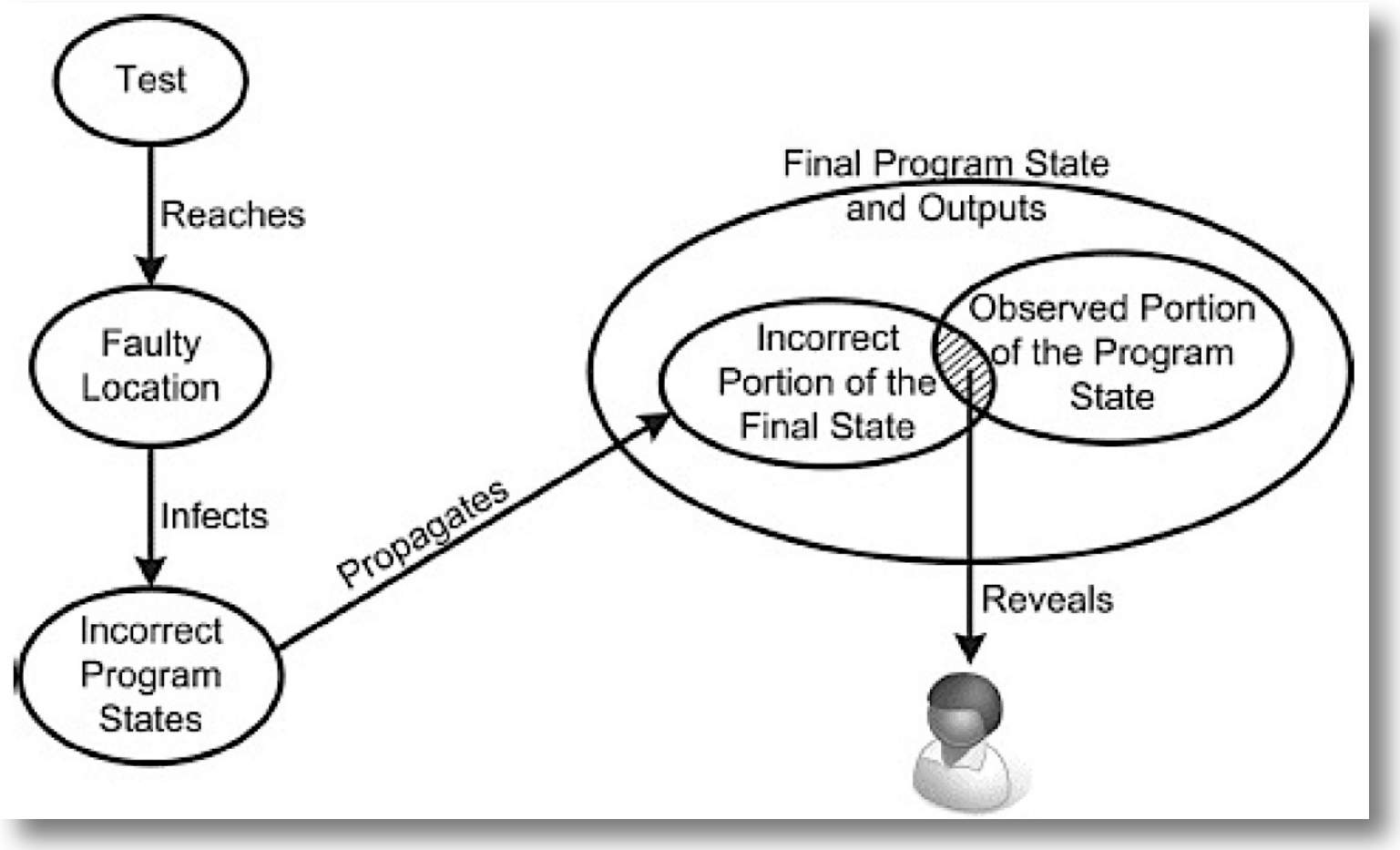
Validation- The process of evaluating software at the end of software development to ensure compliance with intended usage.

Faults, Failure, and Error

Fault: A **static** defect in the software.

Error: An **incorrect internal state** that is the manifestation of some fault.

Failure: **External, incorrect behavior** with respect to the requirements or another description of the expected behavior.



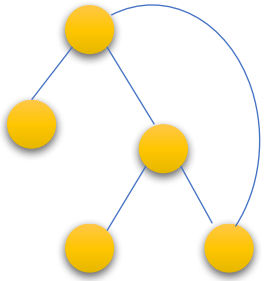
RIPR Model

Reachability, Infection, Propagation, and Revealability

(not X or not Y) and A and B

```
If (x < y)
      z = x - y
else
      z = 2*x
```

A: {0, 1, >1}
B: {600, 700, 800}
C: {swe, cs, isa, infs}



Coverage Criteria

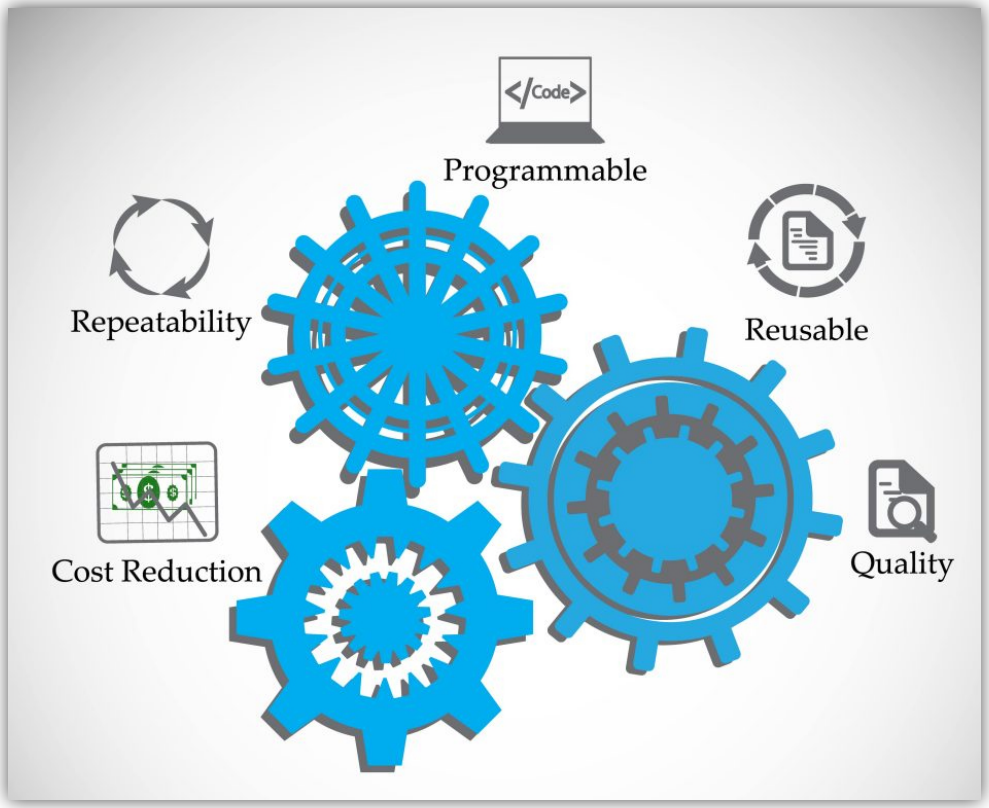
- Coverage criteria give us structured, practical ways to search the input space.
- A coverage criterion is a **rule** or collection of rules that yield **test requirements**.

Model-Driven Test Design

The **Model-Driven Test Design (MDTD)** process breaks testing into a series of small tasks that simplify test generation. Then test designers isolate their task, and work at a higher level of abstraction by using mathematical engineering structures to design test values independently of the details of software or design artifacts, test automation, and test execution.

Test Automation

The use of software to control the **execution** of tests, the **comparison** of actual outcomes to predicted outcomes, the **setting up** of test preconditions, and other test **control** and test **reporting** functions.



Software Testability

The **degree** to which a system or component **facilitates** the establishment of test criteria and the performance of tests to determine whether those criteria have been met.

Observability

How easy it is to observe the behavior of a program in terms of its outputs, effects on the environment and other hardware and software components

Controllability

How easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviors

Test Automation Framework

A test framework provides a standard **design for test scripts**, and should include support for the **test driver**.

What test values to use?

Test frameworks provide very simple ways to **automate** our tests that make testing **efficient as well as effective.**

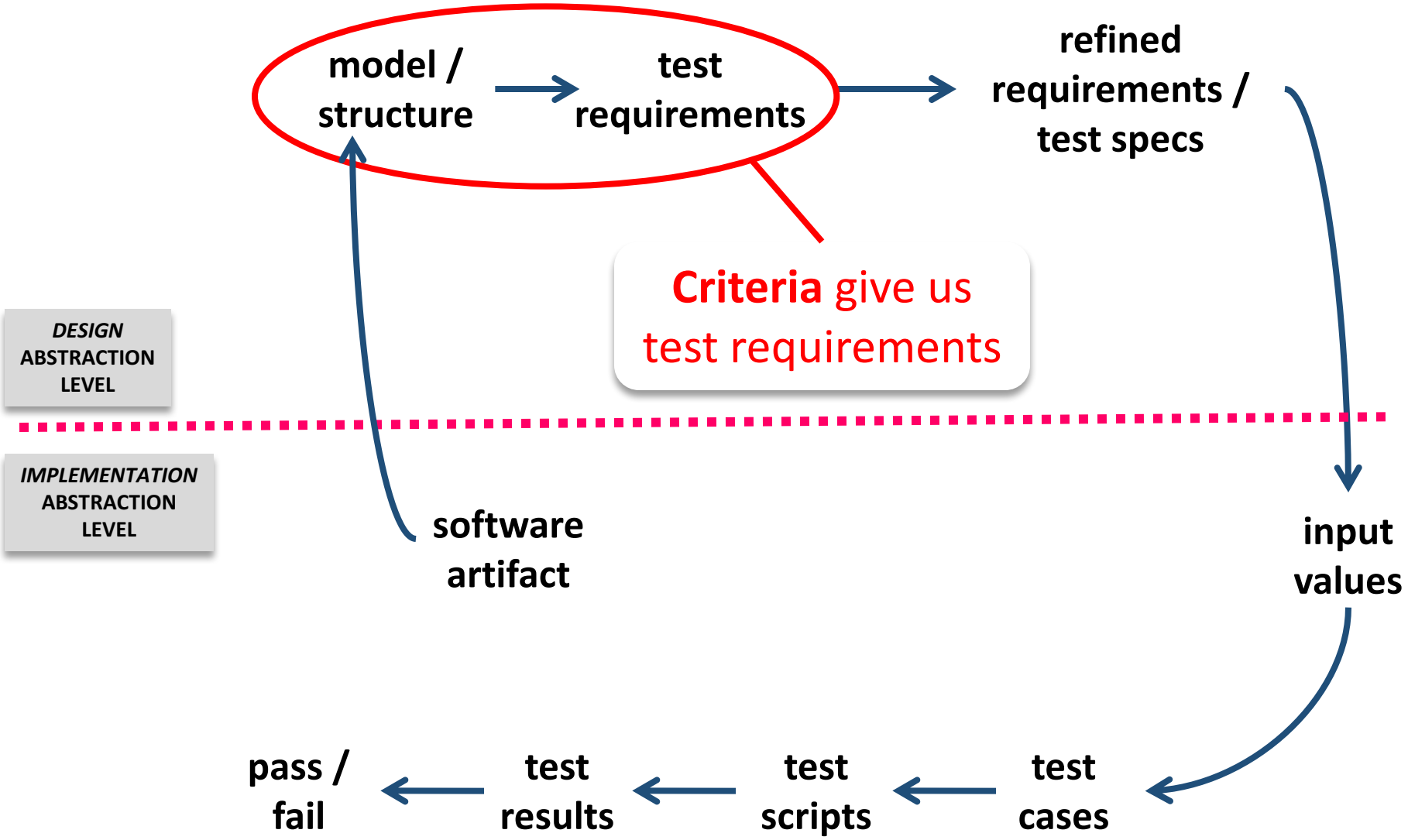
It is **no "silver bullet"** however … it does not solve the hard problem of testing:

"what test values to use?"

This is **test design** … the purpose of **test criteria.**

Criteria-Based Test Design

Model-Driven Test Design



Test Coverage Criteria

A tester's job is **simple!**

Define a model of the software, then **find ways to cover** it

Test Requirements

A specific element of a software artifact that a test case must satisfy or cover.

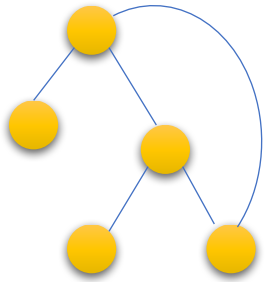
Coverage Criterion

A rule or collection of rules that impose test requirements on a test set.

(not X or not Y) and A and B

```
If (x < y)
      z = x - y
else
      z = 2*x
```

A: {0, 1, >1}
B: {600, 700, 800}
C: {swe, cs, isa, infs}



Four ways to model a software artefact

Input Space Partition Testing

Input Domain

The input domain is defined in terms of the **possible values** that the input parameters can have.

- Method parameters
- Global variables
- Objects representing current state
- User-level inputs
- …

- The input domain is **partitioned** into **regions** (blocks)
- At least **one value** is selected from each region

Modelling the Input Domain

Step 1- Identify testable functions

Step 2- Find all inputs, parameters, & characteristics

Move from impl. level to design abstraction level

Step 3- Model the input domain

Step 4- Apply a test criterion to choose combinations of values

Entirely at the design abstraction level

Step 5- Refine combinations of blocks into test inputs

Back to the implementation abstraction level

Characteristics, Blocks, & Values

- Valid, invalid, special values
- Explore boundaries
- Balance the number of blocks in the characteristics
- Missing blocks
- …

Candidates for characteristics

- Interface-based approach
- Preconditions and postconditions
- Relationships among variables
- Relationship of variables with special values (zero, null, blank, …)
- …

How should we consider **multiple partitions** at the same time?

What **combination of blocks** should we choose values from?

Combination Strategies Criteria

All Combinations Criterion (ACoC)

Test with all combinations of blocks from all characteristics.

- 4 Characteristics: A, B, C, D
- Abstract blocks: A = [a1, a2]; B = [b1, b2]; C = [c1, c2, c3]; D = [d1, d2]

a1 b1 c1 d1	a1 b2 c1 d1	a2 b1 c1 d1	a2 b2 c1 d1
a1 b1 c1 d2	a1 b2 c1 d2	a2 b1 c1 d2	a2 b2 c1 d2
a1 b1 c2 d1	a1 b2 c2 d1	a2 b1 c2 d1	a2 b2 c2 d1
a1 b1 c2 d2	a1 b2 c2 d2	a2 b1 c2 d2	a2 b2 c2 d2
a1 b1 c3 d1	a1 b2 c3 d1	a2 b1 c3 d1	a2 b2 c3 d1
a1 b1 c3 d2	a1 b2 c3 d2	a2 b1 c3 d2	a2 b2 c3 d2

Each Choice Coverage (ECC)

Use at least one value from each block for each characteristic in at least one test case.

- 4 Characteristics: A, B, C, D
- Abstract blocks: A = [a1, a2]; B = [b1, b2]; C = [c1, c2, c3]; D = [d1, d2]

a1 b1 c1 d1
a2 b2 c2 d2
a1 b1 c3 d1

Pair-Wise Coverage (PWC)

A value from each block for each characteristic must be combined with a value from every block for each other characteristic.

- 4 Characteristics: A, B, C, D
- Abstract blocks

  A = [a1, a2];

  B = [b1, b2];

  C = [c1, c2, c3];

  D = [d1, d2]

(a1, b1)	(a2, b1)	(b1, c1)	(b2, c1)	(c1, d1)
(a1, b2)	(a2, b2)	(b1, c2)	(b2, c2)	(c1, d2)
(a1, c1)	(a2, c1)	(b1, c3)	(b2, c3)	(c2, d1)
(a1, c2)	(a2, c2)	(b1, d1)	(b2, d1)	(c2, d2)
(a1, c3)	(a2, c3)	(b1, d2)	(b2, d2)	(c3, d2)
(a1, d1)	(a2, d1)			
(a1, d2)	(a2, d2)			

Base Choice Coverage (BCC)

A base choice block is chosen for each characteristic, and a base test is formed by using the base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant and using each non-base choice in each other characteristic.

- 4 Characteristics: A, B, C, D
- Abstract blocks

  A = [a1, a2];

  B = [b1, b2];

  C = [c1, c2, c3];

  D = [d1, d2]

Base	**a1 b1 c1 d1**
A	**a2** b1 c1 d1
B	a1 **b2** c1 d1
C	a1 b1 **c2** d1
C	a1 b1 **c3** d1
D	a1 b1 c1 **d2**

Multiple Base Choice Coverage (MBCC)

At least one, and possibly more, base choice blocks are chosen for each characteristic, and base tests are formed by using each base choice for each characteristic at least once. Subsequent tests are chosen by holding all but one base choice constant for each base test and using each non-base choice in each other characteristic.

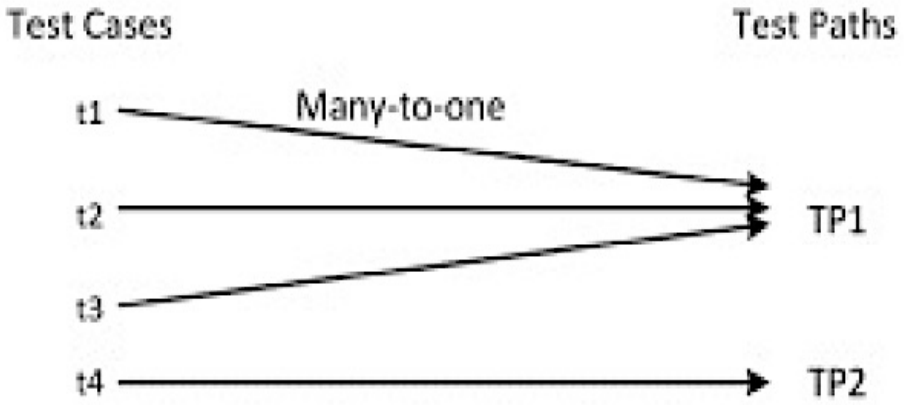
- If *M* base tests and m_i base choices for each characteristic:

$$M + \sum_{i=1}^{Q} (M * (B_i - m_i))$$

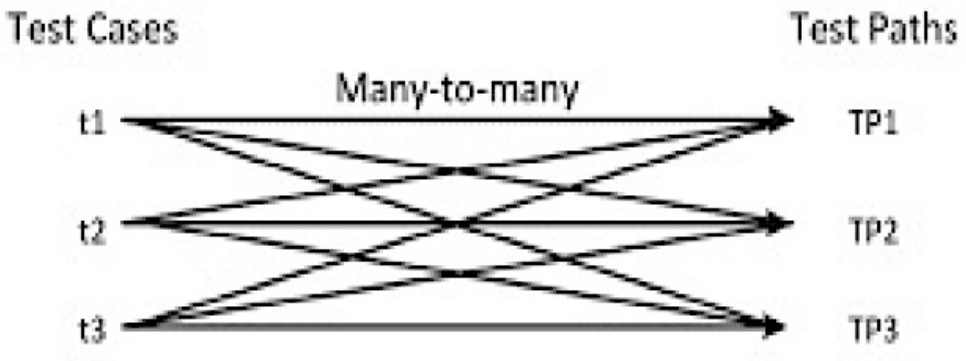
Graph Coverage

Test Cases and Test Paths

Deterministic software–test always executes the same test path.



Non-deterministic software–the same test can execute different test paths



GRAPH COVERAGE CRITERIA

Graph Coverage: Given a set ***TR*** of test requirements for a graph criterion ***C***, a test set ***T*** satisfies ***C*** on graph ***G*** if and only if for every test requirement ***tr*** in ***TR***, there is at least one test path ***p*** in ***path(T)*** such that ***p*** meets ***tr***.

Node & Edge Coverage

Node Coverage (NC)- Test set *T* satisfies node coverage on graph *G* iff for every syntactically reachable node *n* in *N*, there is some path *p* in *path(T)* such that *p* visits *n*.

Node Coverage (NC): *TR* contains each reachable node in *G*.

Edge Coverage (NC): *TR* contains each reachable path of length up to 1, inclusive, in *G*.

Covering Multiple Edges

Edge-Pair Coverage (EPC): *TR* contains each reachable path of length up to 2, inclusive, in *G*.

Complete Path Coverage (CPC): *TR* contains all paths in *G*.

Specified Path Coverage (SPC): *TR* contains a set of *S* test paths, where *S* is supplied as parameter.

How can we handle loops in graphs?

Prime Path Coverage

Prime Path Coverage (PPC): *TR* contains each prime path in *G.*

Touring, sidetrips, and detours

Tour: *A test path p tours subpath q if q is a subpath of p.*

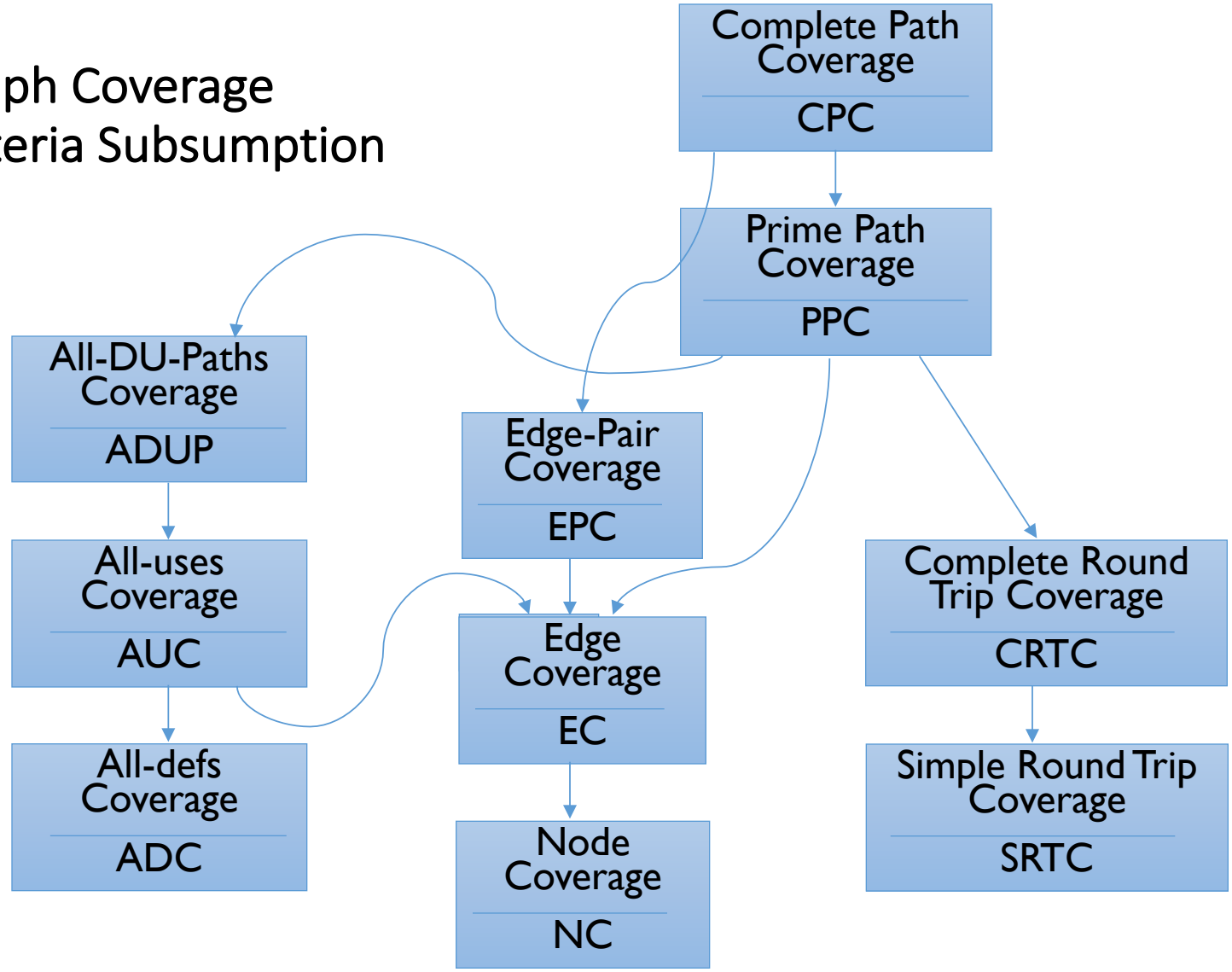
Tour With Sidetrips: A test path p tours subpath q with *sidetrips* iff every edge in q is also in p in the same order

- The tour can include a sidetrip, as long as it comes back to the same node

Tour With Detours: A test path p tours subpath q with *detours* iff every node in q is also in p in the same order

- The tour can include a detour from node *ni*, as long as it comes back to the prime path at a successor of *ni*

Graph Coverage Criteria Subsumption



Graph Coverage Applications

Control Flow Graph (CFG)

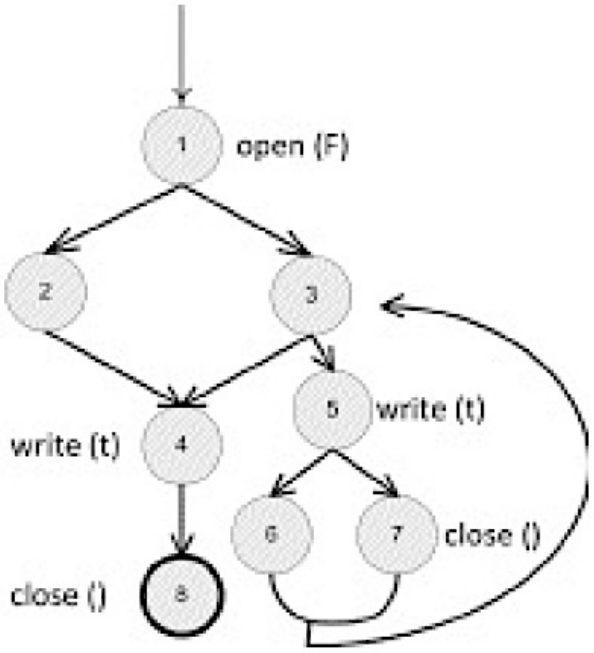
The most common graph for **source code**.

Graph Coverage for Design Elements

- Data abstraction & OO language features
- Testing software **based on design elements** is becoming more important

Graph Coverage for Specifications

Testing Sequencing Constraints

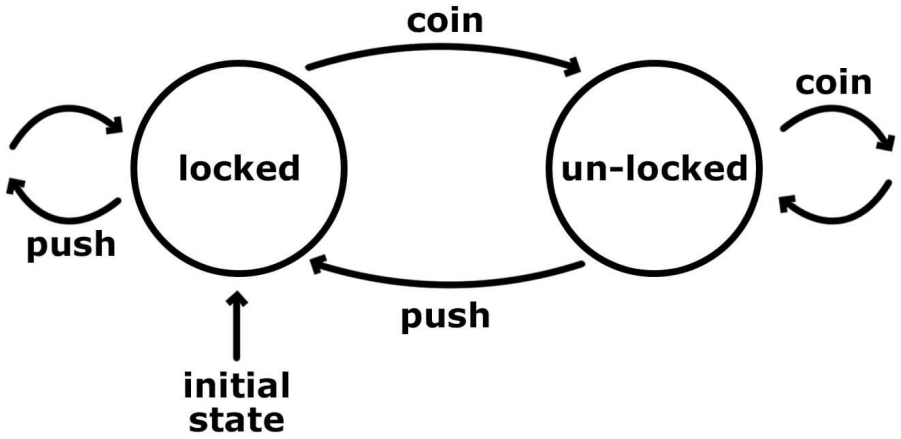


`FileADT`

- `open (String fName)`
- `close (String fName)`
- `write (String textLine)`

1. An `open(F)` must be executed before every `write(t)`
2. An `open(F)` must be executed before every `close()`
3. A `write(t)` must not be executed after a `close()` unless an `open(F)` appears in between
4. A `write(t)` should be executed before every `close()`
5. A `close()` must not be executed after a `close()` unless an `open(F)` appears in between
6. An `open(F)` must not be executed after an `open(F)` `unless` a `close()` appears in between

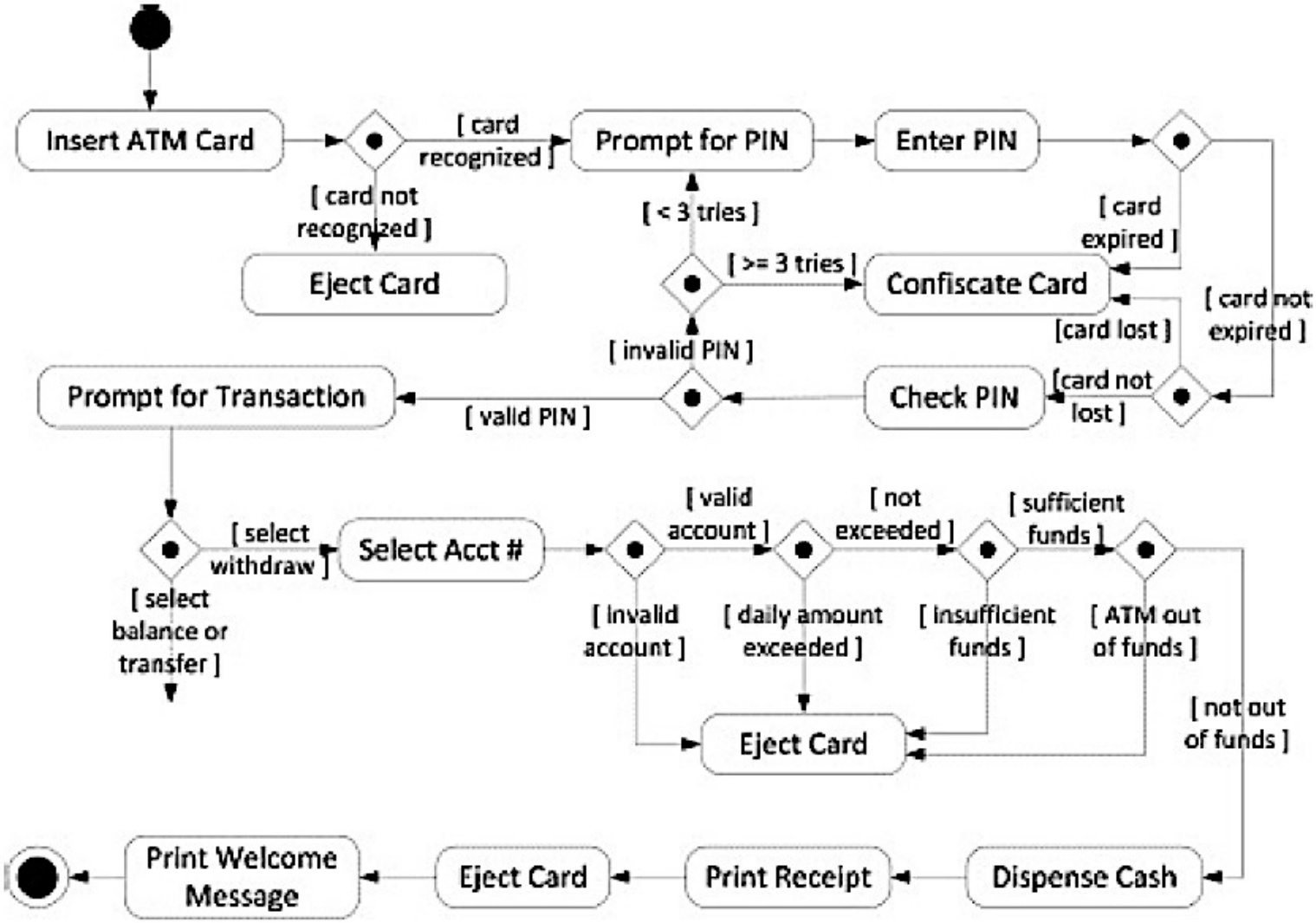
Testing State Behaviour



1. Combining control flow graphs
2. Using the software structure
3. Modeling state variables
4. Using the implicit or explicit specifications

Graph Coverage for Use Cases

UML Activity Diagram



Logic Coverage

Example

```
if ((a > b) || c) && (x < y))
        o.m();
else
        o.n();
```

$((a > b) \vee C) \wedge (x < y)$

```
button2 = true (when gear = park)
```

$gear = park \wedge button2 = true$

```
"pre: stack Not full AND object
reference parameter not null"
```

$\neg stackFull() \wedge newObj \neq null$

Logic Coverage Criteria

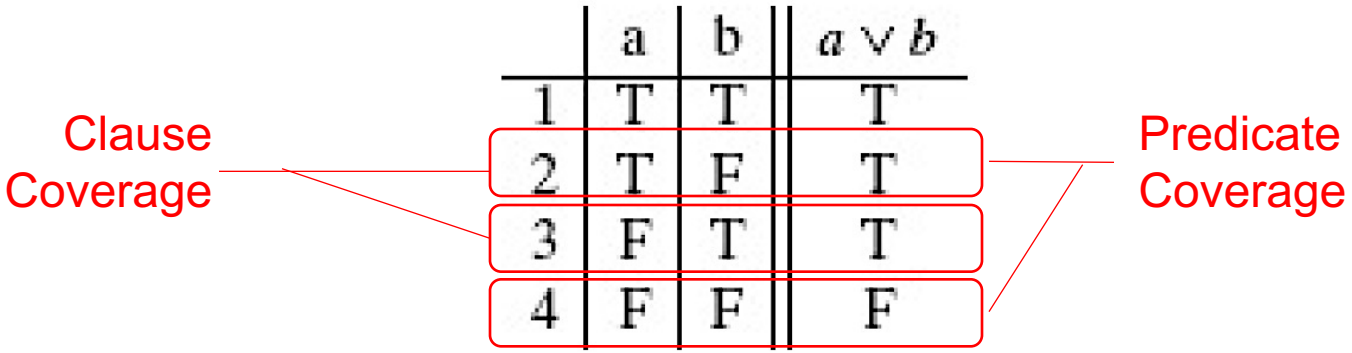
- Develop a **model** of the software as **one or more predicates**
- Require **tests** to satisfy some **combination of clauses**

Predicate & Clause Coverage

- *P* is the set of predicates
- *p* is a single predicate in *P*
- *C* is the set of clauses in *P*
- *c* is a single clause in *C*

Predicate Coverage (PC): For each *p* in *P*, *TR* contains two requirements: *p* evaluates to *true*, and *p* evaluates to *false*

Clause Coverage (CC): For each *c* in *C*, *TR* contains two requirements: *c* evaluates to *true*, and *c* evaluates to *false*.



	a	b	$a \vee b$
1	T	T	T
2	T	F	T
3	F	T	T
4	F	F	F

Problems with PC and CC!

Neither Predicate Coverage nor Clause Coverage subsumes the other.

Combinatorial Coverage (CoC)

Combinatorial Coverage (CoC): For each *p* in *P*, *TR* has test requirements for the clauses in *Cp* to evaluate to each possible combination of truth values.

P = (a & (b | c))

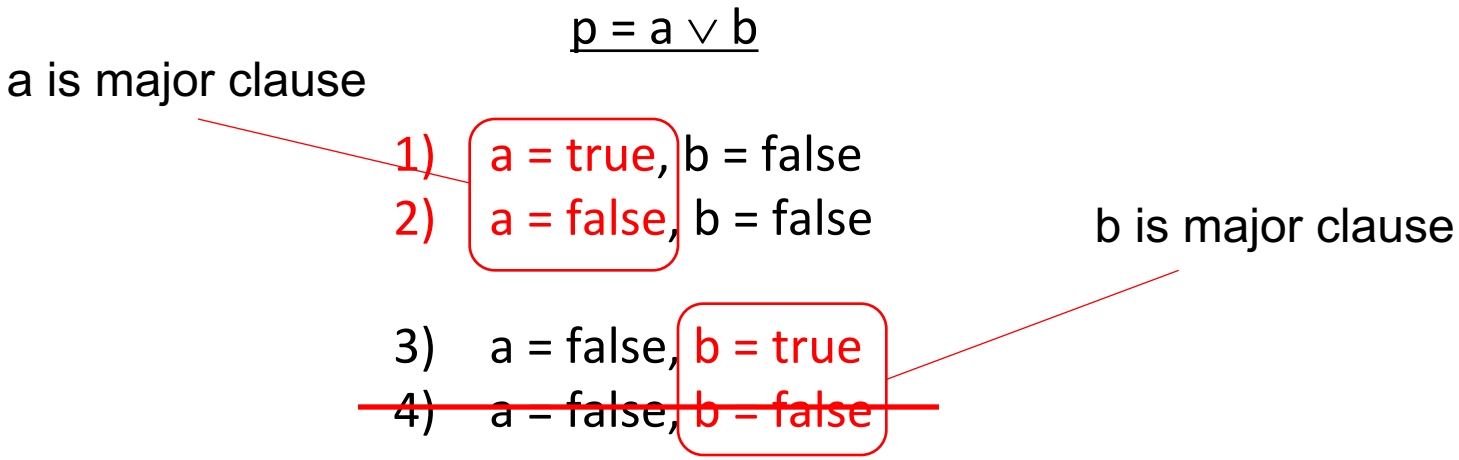
Test 1: (true & (true | true))
Test 2: (false & (false | false))

ACTIVE CLAUSES

To really test the results of a clause, the clause should be the determining factor in the value of the predicate

Active Clause Coverage

Active Clause Coverage (ACC): For each clause c_i in each predicate p, choose values for the other clauses to make c_i active. TR has two requirements for each c_i: c_i evaluates to true and c_i evaluates to false.



Applying
Logic Coverage Criteria

Programs

Structural Logic Coverage

- Predicates are derived directly from decision statements in the programs (if, case, and loop statements).

Finding Values?

- Reachability
- Controllability
- Internal variables

```
public int checkVal(int x) {
   int y = x*2;
   if (x>0)
      if ((x>10 && x<20) || y==50)
         return 1;
   else
      if ((x<-10 && x>-20) || y<-60)
         return 2;
   return 42;
}
```

Logic Coverage for Specification

"list all the wireless mice that either retail for more than $100 or for which the store has more than 20 items. Also list non-wireless mice that retail for more than $50."

```
public static int cal (int month1, int day1, int month2,
                       int day2, int year) {
//***********************************************
// Calculate the number of Days between the two given days in
// the same year.
// preconditions : day1 and day2 must be in same year
//                  1 <= month1, month2 <= 12
//                  1 <= day1, day2 <= 31 // month1 <= month2
//                  The range for year: 1 ... 10000
//***********************************************
```

Logic Coverage for Specification

"lis ... h
the ... or
... more than $50."

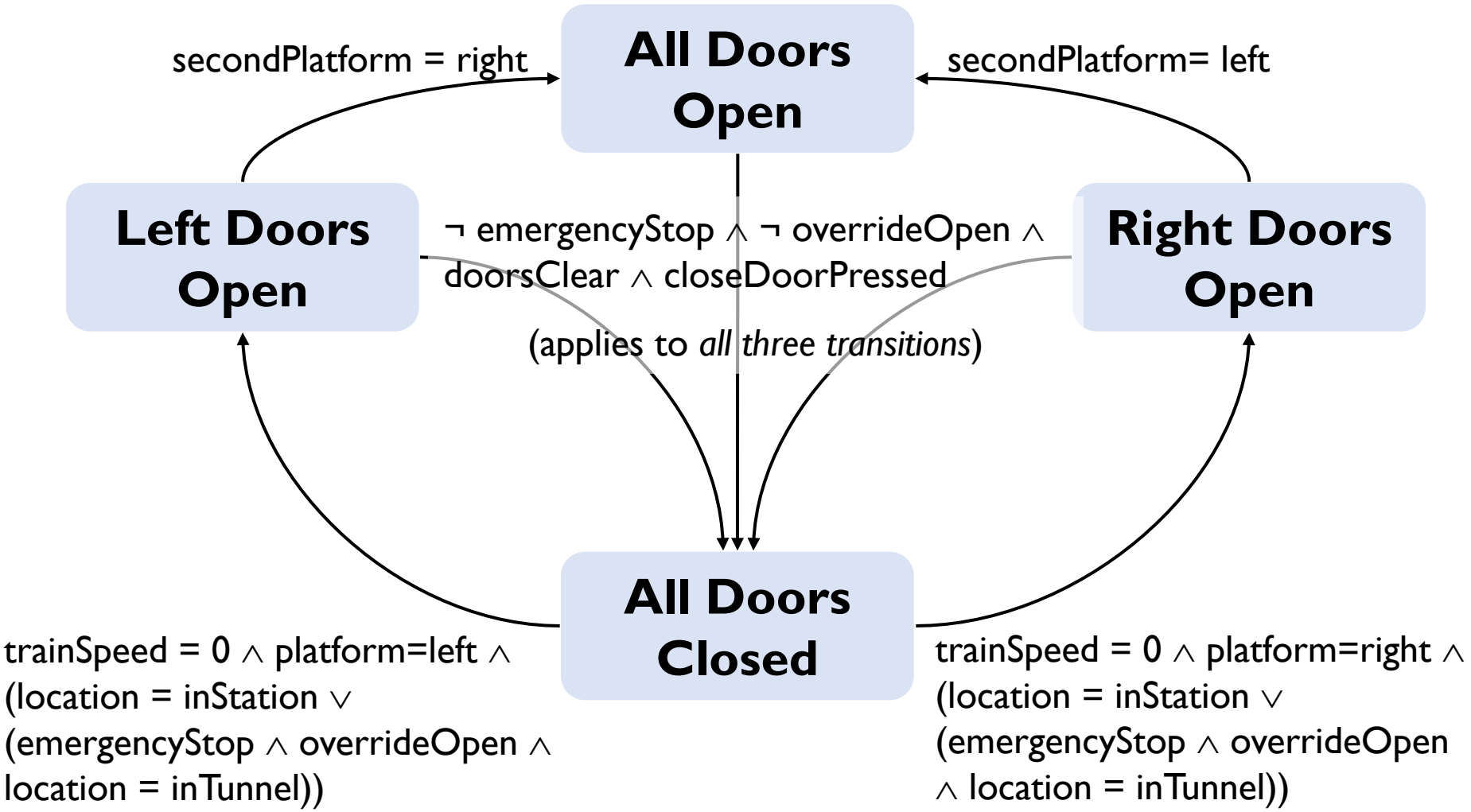
$$((mouseType = wireless) \wedge ((retail > 100) \vee (stock > 20))) \vee (\neg(mouseType = wireless) \wedge (retail > 50))$$

```
public static int cal (int month1, int day1, int month2,
                       int day2, int year) {
//*********************************************
```

$$month1 >= 1 \wedge month1 <= 12 \wedge month2 >= 1 \wedge month2 <= 12 \wedge month1 <= month2$$

$$\wedge day1 >= 1 \wedge day1 <= 31 \wedge day2 >= 1 \wedge day2 <= 31 \wedge year >= 1 \wedge year <= 10000$$

```
//                    1 <= month1, month2 <= 12
//                    1 <= day1, day2 <= 31 // month1 <= month2
//                    The range for year: 1 ... 10000
//*********************************************
```


All Doors Open
secondPlatform = right
secondPlatform= left
Left Doors Open
Right Doors Open
¬ emergencyStop ∧ ¬ overrideOpen ∧ doorsClear ∧ closeDoorPressed
(applies to all three transitions)
All Doors Closed
trainSpeed = 0 ∧ platform=left ∧ (location = inStation ∨ (emergencyStop ∧ overrideOpen ∧ location = inTunnel))
trainSpeed = 0 ∧ platform=right ∧ (location = inStation ∨ (emergencyStop ∧ overrideOpen ∧ location = inTunnel))

Writing Effective Test Oracles

What Should Be Checked?

A **test oracle strategy** is a rule or a set of rules that specify which program states to check.

- **Precision**: **how much** of the output state should be checked.
- **Frequency**: **when and how often** the output state should be checked.

Determining Correct Values?

Test Implementation

Theory is usually further from practice than we wish!

Class Integration Test Order (CITO)

The general **goal** is to integrate and test classes in the order that requires the **least scaffolding**, or additional software.

Test Doubles

- Any kind of pretend object used in place of a real object for testing purposes.
- Implements partial functionality.

- Mocks, Stubs, ….

- The components have **not yet been written**,
- The components do something that we **can't afford** to happen during testing.

Regression Testing

Goldilocks Problem!

Right size, finish in timely manner, efficient, effective, safe, precise, ….

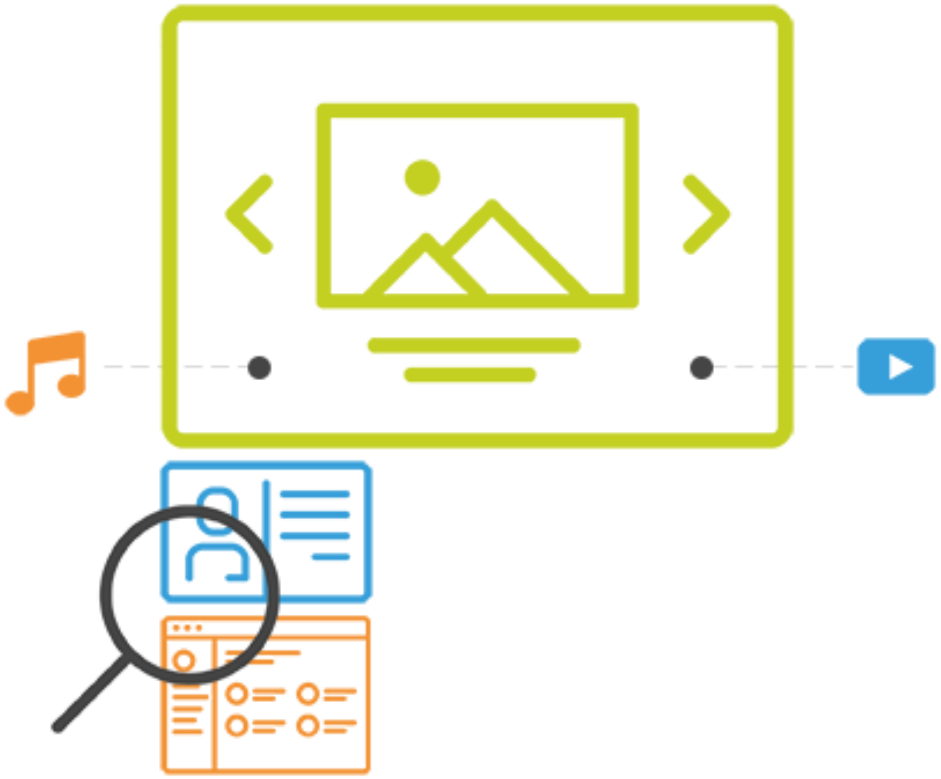
Regression Test Selection

Select a **subset** of the existing tests to retest the new version of the program.

Test Case Prioritisation

Order regression tests such that their execution provides benefits such as earlier detection of faults..

GUI Testing



3. How much testing is enough?

2. Oracle Specification

1. Event Selection

Mutation Testing