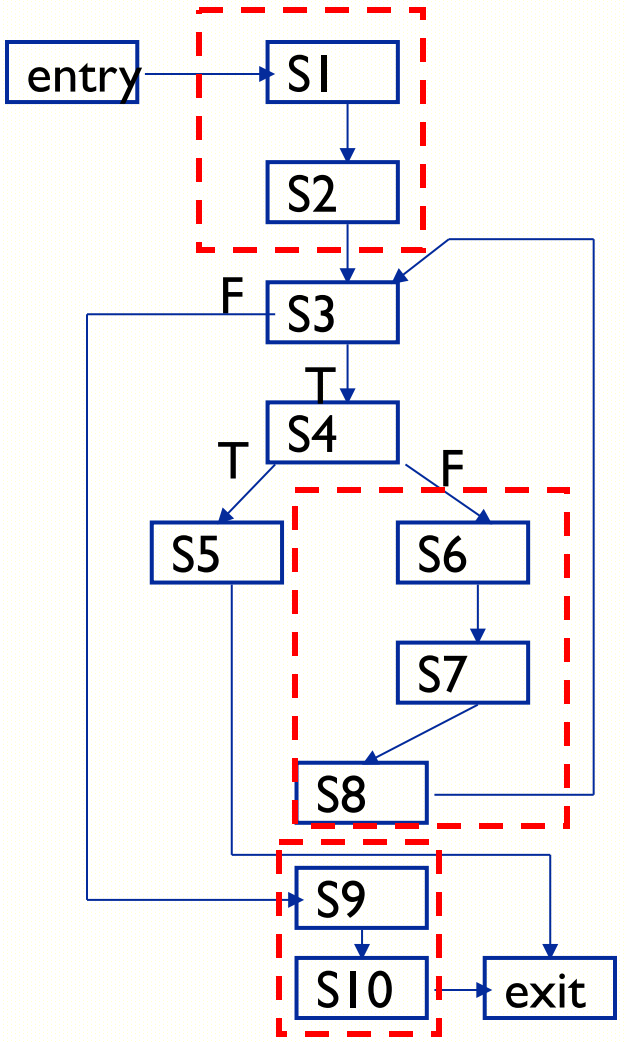
Fundamentals of (Static and Dynamic) Software Verification

Control-flow Analysis
Data-flow Analysis
Static VS Dynamic Analysis
Software Testing

Control Flow Graph

Procedure AVG

```
S1   count = 0
S2   fread(fptr, n)
S3   while (not EOF) do
S4      if (n < 0)
S5         return (error)
        else
S6         nums[count] = n
S7         count ++
        endif
S8         fread(fptr, n)
      endwhile
S9   avg = mean(nums,count)
S10  return(avg)
```

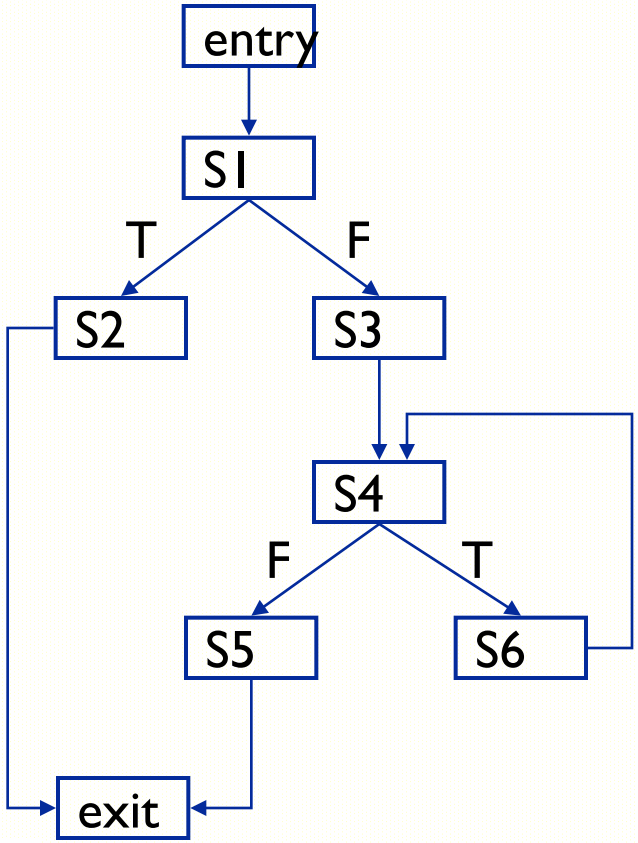


Dominance/Postdominance

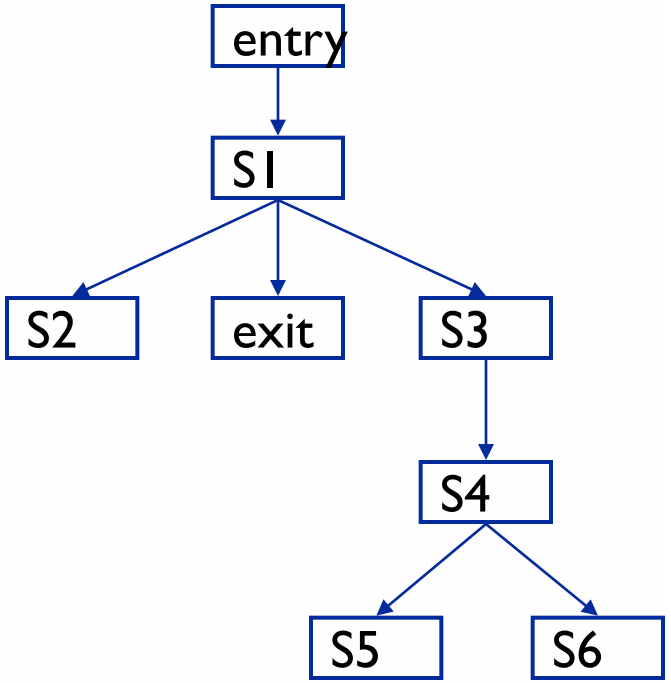
- A node n **dominates** a node m (n dom m) if every path from the entry to m includes n
- A node n **postdominates** a node m (n pdom m) if every path from the m to the exit includes n

Dominance

CFG

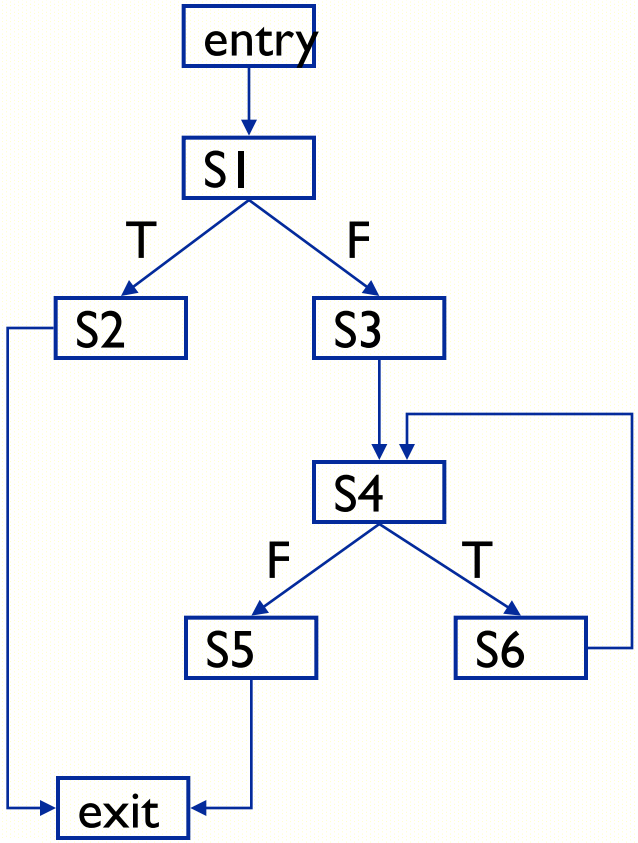


Dominance Tree

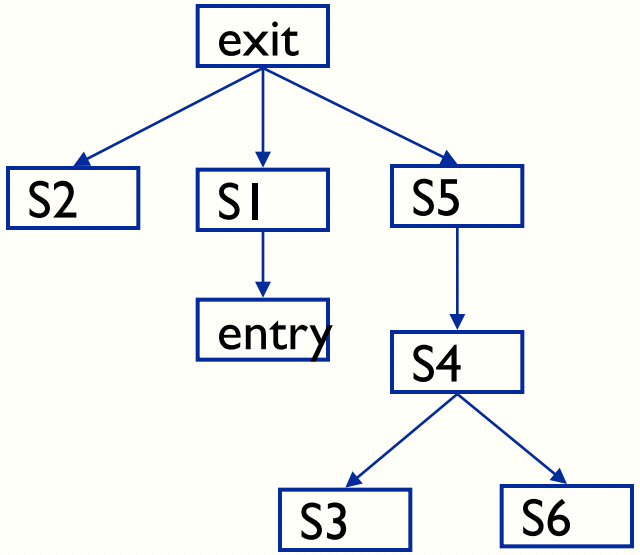


Postdominance

CFG



Postdominance Tree



Basic Data-Flow concepts

- **Definition** and **Use**

  Consider statement X = Y + Z

  - Definitions?
  - Uses?

- **Kill** and **Reach**

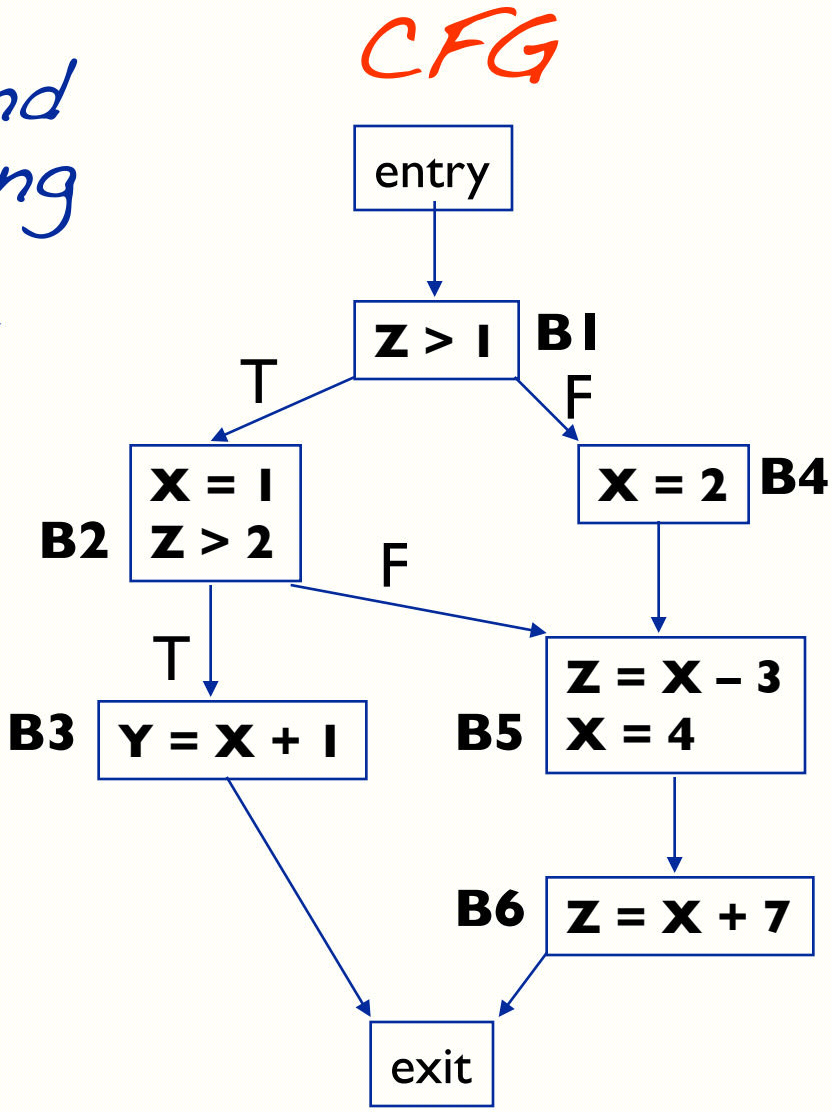
  - A definition d of a variable x is *killed* at a statement s iff s redefines x and the last assignment to x was d
  - A definition d of x *reaches* s if there is at least a path from d to s along which x is not killed (*def-clear path*)

Dependence Analysis

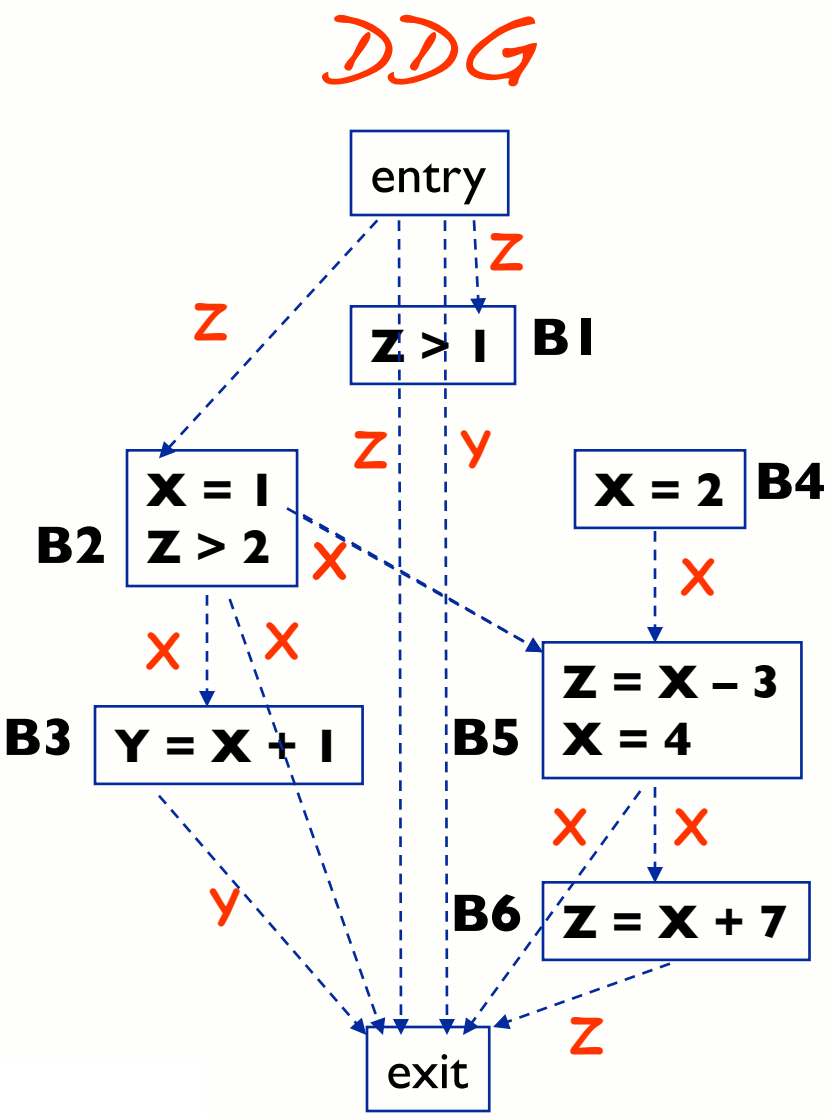
- Important for
  - Optimization (e.g., instruction scheduling)
  - Software engineering
    - Program understanding
    - Reverse engineering
    - Debugging
- Two main kinds of dependences
  - Data related
  - Control related

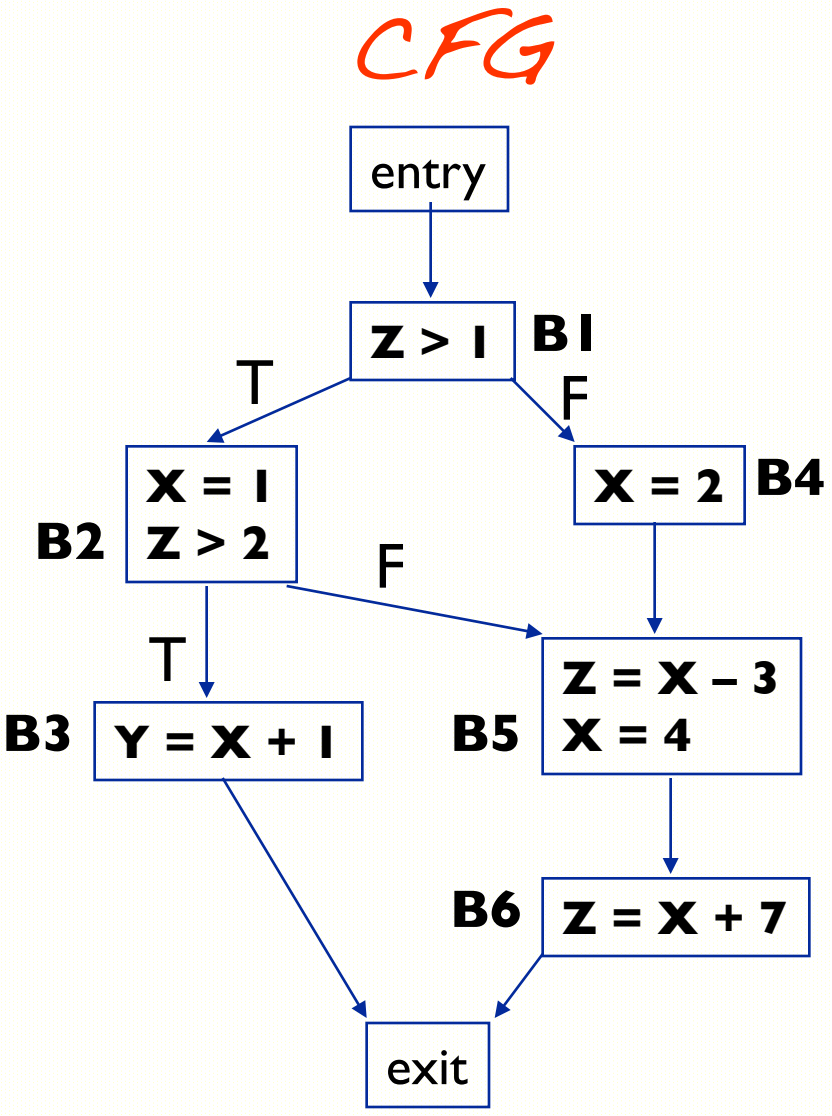
Data-Dependence Graph (DDG)

DDG: one node for every basic block and one edge representing the flow of data between two nodes



Data-Dependence Graph



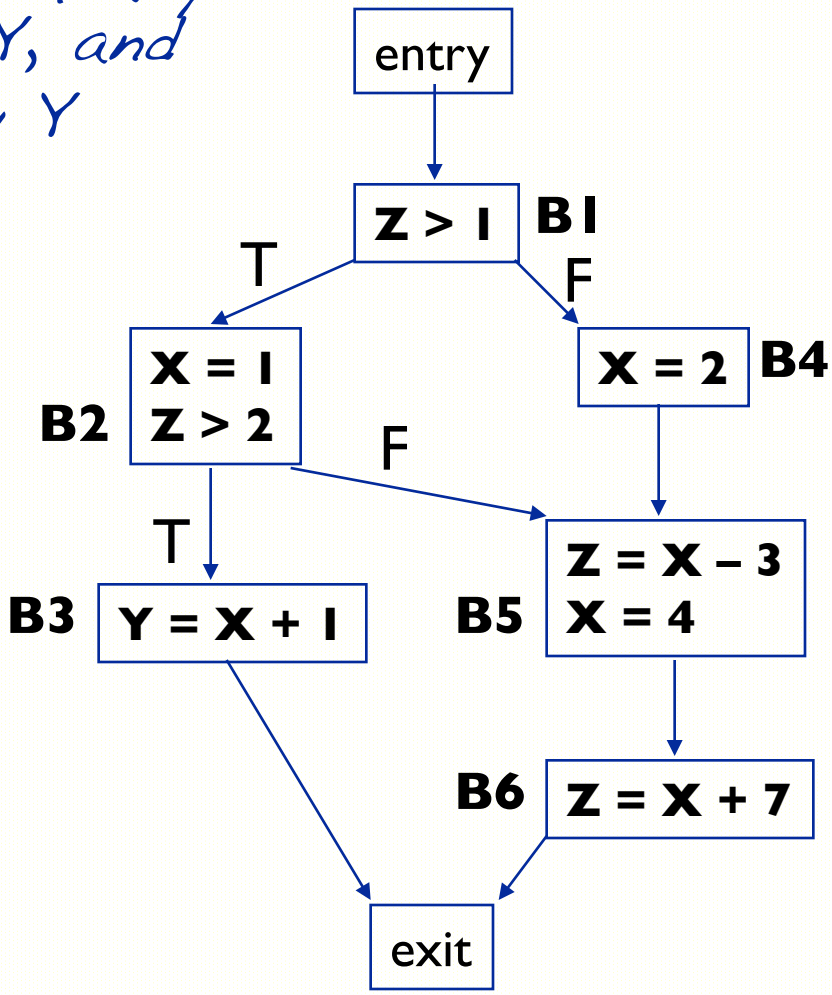


Control-Dependence Graph (CDG)

Y control-dependent on X iff

1. ∃ path P from X to Y with any Z in P postdominated by Y, and
2. X is not postdominated by Y

CFG



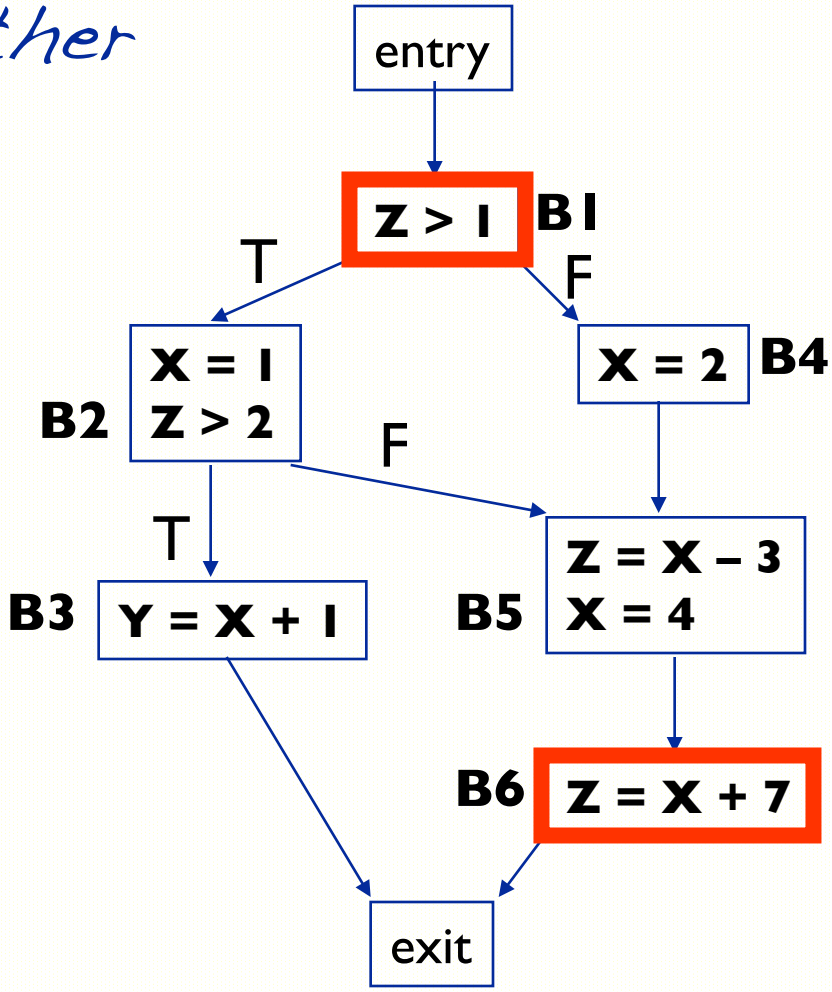
Control-Dependence Graph (CDG)

Intuitively: two edges out of X; traversing one edge always leads to Y, the other may not lead to Y

Dependences:

- B1, exit – entry
- B2 – B1T
- B3 – B2T
- B4 – B1F
- B5 – B2F, B1F
- B6 – B2F, B1F

CFG



Program-Dependence Graph (PDG)

- A PDG for a program P is the combination of the DDG and CDG for P

- A PDG contains nodes representing statements/basic blocks in P and edges representing either control or data dependence between nodes

Static VS Dynamic Analysis

- Static analysis operates on a model of the SW (w/o executing it)
  - Can produce definitive information that holds for all inputs
- Dynamic analysis operates on dynamic information collected by running the SW
  - Produces "sampling information" that holds for the inputs considered
- Combined static and dynamic analyses leverage complementary strengths

Do We Need Dynamic Analysis?

- Imprecision of static analysis
- Need to test for properties of executions (e.g., debugging)
- Need to test assumptions about the environment
- Need to determine performance for the average case
- Need to test for non-functional properties, such as usability
- ...

Examples of Dynamic Analysis

- Testing
- Profiling
- Coverage analysis
- Dynamic-invariant detection
- Assertions
- Dynamic tainting
- Dynamic slicing
- ...

Issues in Dynamic Analysis

- Collecting dynamic data
  - Instrumentation
  - Runtime system
  - Debugging interfaces
- Overhead
- Making sure observations don't change the behavior of the system
- Selecting the "right" inputs

Software Testing

General Concepts

Software Is Buggy

- On average, 1-5 errors per 1KLOC
- Windows 2000
  - 35M LOC
  - 63,000 known bugs at the time of release
  - 2 per 1,000 lines

- For mass market software 100% correct is infeasible, but
- We must verify the SW as much as possible

Failure, Fault, Error

Failure

Observable incorrect behavior of a program. Conceptually related to the behavior of the program, rather than its code.

Fault (bug)

Related to the code. Necessary (not sufficient!) condition for the occurrence of a failure.

Error

Cause of a fault. Usually a human error (conceptual, typo, etc.)

Failure, Fault, Error: Example

```
1.  int double(int param) {
2.      int result;
3.      result = param * param;
4.      return(result);
5.  }
```

A call to double(3) returns 9
Result 9 represents a failure
Such failure is due to the fault at line 3
The error is a typo (hopefully)

Approaches to Verification

- Testing: exercising software to try and generate failures
- Static verification: identify (specific) problems statically, that is, considering all possible executions
- Inspection/review/walkthrough: systematic group review of program text to detect faults
- Formal proof: proving that the program text implements the program specification

TESTING

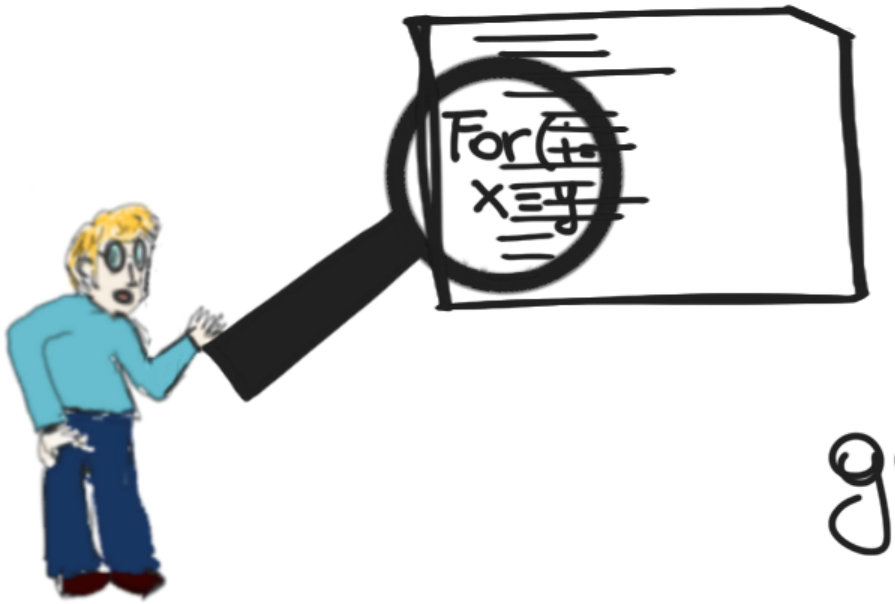
Test case : $\{i \in D, o \in O\}$

Test suite : set of test cases

VERIFICATION

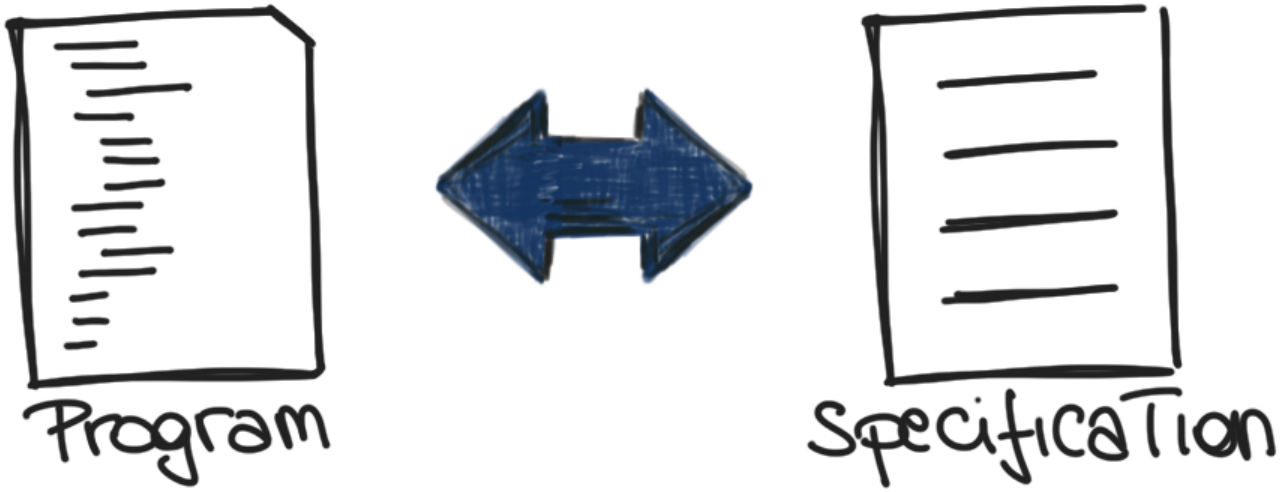
Considers all possible inputs (executions)

INSPECTIONS (AKA
- reviews
- walkthroughs)



Manual
group activity

Formal Proof (of correctness)



Given a formal specification, checks that the code corresponds to such specification

Comparison

Testing

- Pros: no false positives
- Limits: incomplete

Static verification

- P: complete (consider all program behaviors)
- L: false positives, expensive

Inspection

- P: systematic, thorough
- L: informal, subjective

Formal proof (of correctness)

- P: strong guarantees
- L: complex, expensive (requires a spec)

TODAY, QA IS MOSTLY TESTING

"50% of my company employees are Testers, and the rest spends 50% of their time testing"
Who said that?

What is Testing?

Testing == To execute a program with a sample of the input data

- Dynamic technique: program must be executed
- Optimistic approximation:
  - The program under test is exercised with a (very small) subset of all the possible input data
  - We assume that the behavior with any other input is consistent with the behavior shown for the selected subset of input data

Testing Techniques

There are a number of techniques

- Different processes
- Different artifacts
- Different approaches

There are no perfect techniques

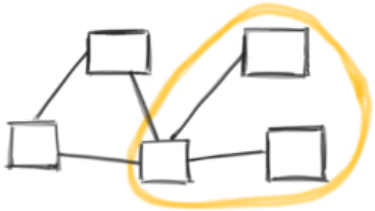
- Testing is a best-effort activity

There is no best technique

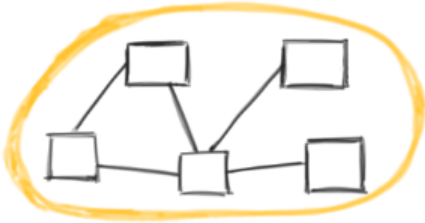
- Different contexts
- Complementary strengths and weaknesses
- Trade-offs

Testing Granularity Levels

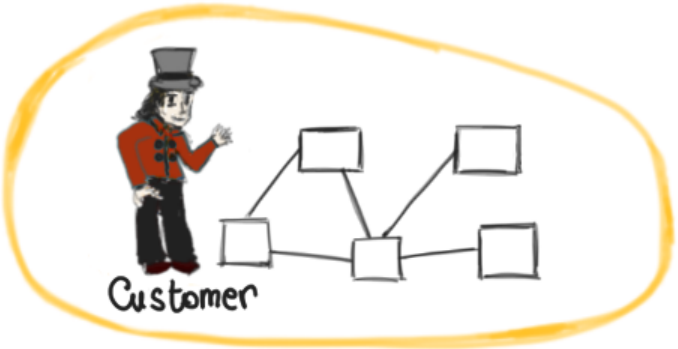
Unit Testing



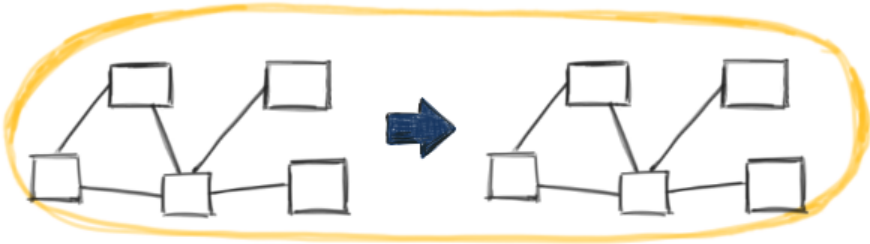
Integration Testing



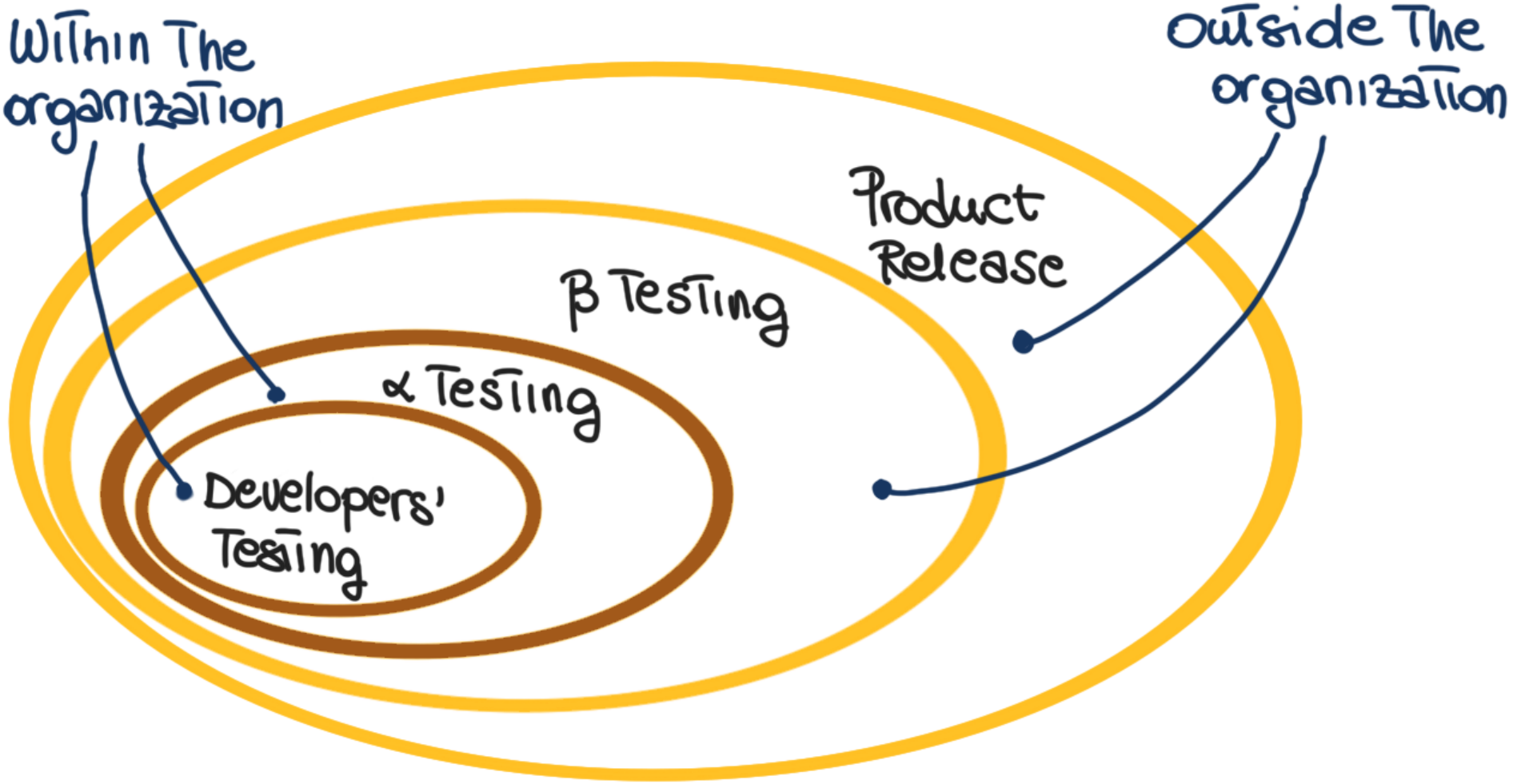
System Testing



Acceptance Testing



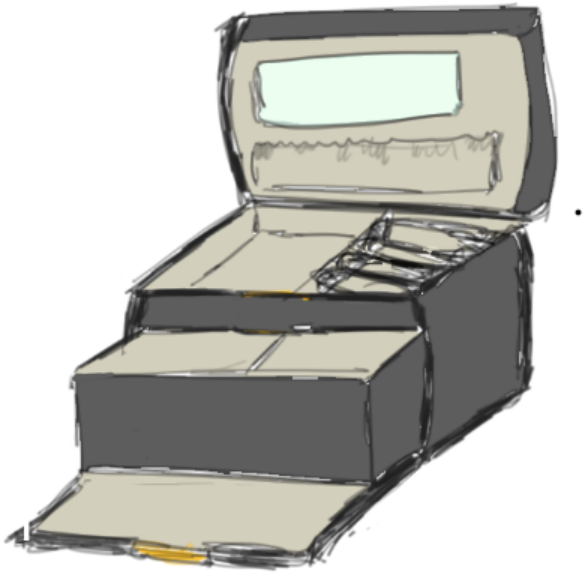
Regression Testing


Within The organization
Outside The organization
Product Release
β Testing
α Testing
Developers' Testing

Functional vs. Structural Testing

BLACK-BOX TESTING

- based on a description of the software (specification)
- cover as much specified behavior as possible
- cannot reveal errors due to implementation details



WHITE-BOX TESTING

- based on the code
- cover as much coded behavior as possible
- cannot reveal errors due to missing paths

BLACK-BOX TESTING EXAMPLE

Specification: inputs an integer and prints it

BLACK-BOX TESTING EXAMPLE

Specification: inputs an integer and prints it

```
1. void printNumBytes( param )
2.     if(param < 1024) printf("%d", param);
3.     else printf("%d KB", param/124);
4. }
```

WHITE-BOX TESTING EXAMPLE

```
1. int fun(int param){
2.     int result;
3.     result = param/2;
4.     return result;
5. }
```

WHITE-BOX TESTING EXAMPLE

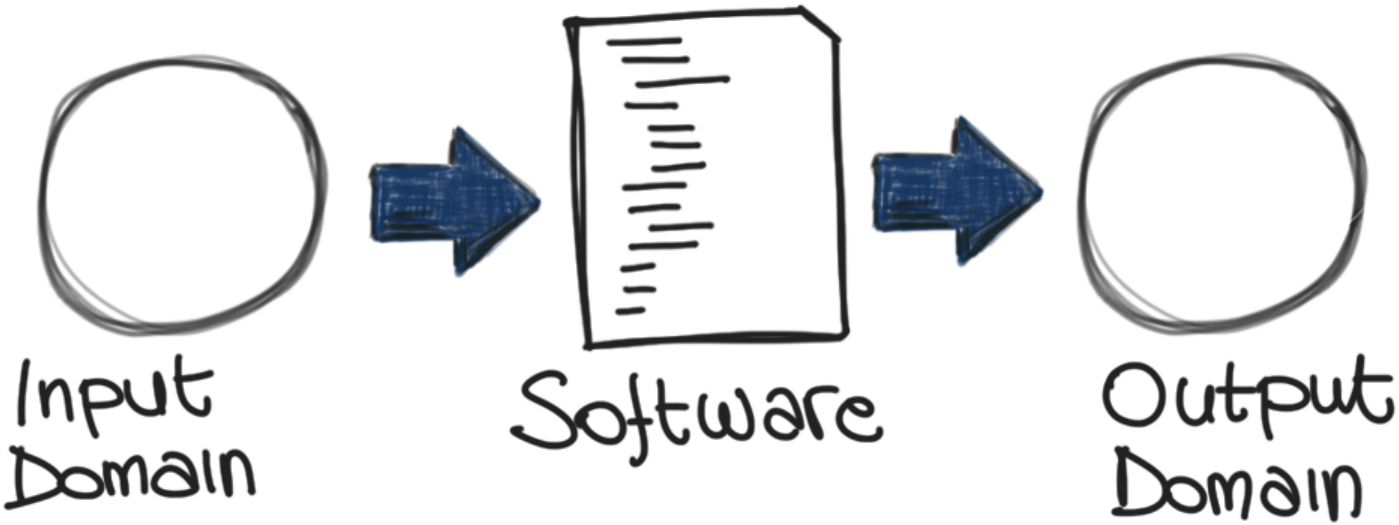
Specification: inputs an integer param and returns half of its value if even, its value otherwise

```
1. int fun(int param){
2.     int result;
3.     result = param/2;
4.     return result;
5. }
```

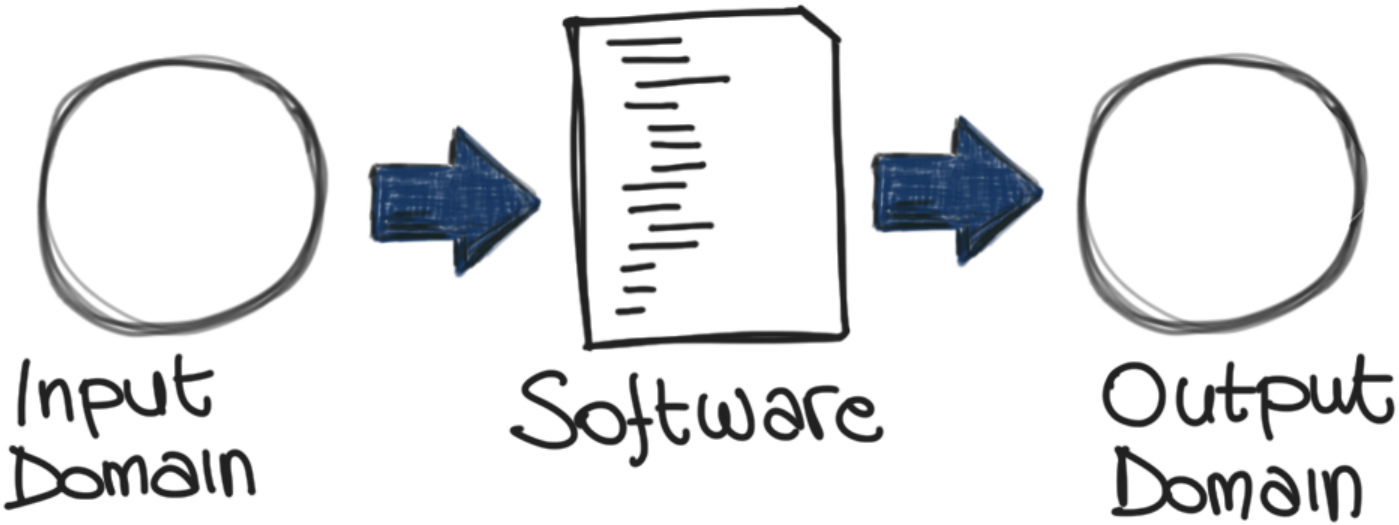
Software Testing

Test-Data Selection

TEST DATA SELECTION



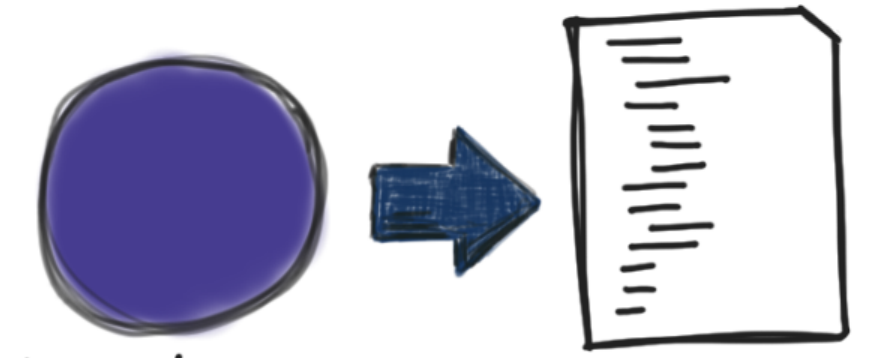
STRAW-MAN IDEA : EXHAUSTIVE TESTING !



STRAW-MAN IDEA: EXHAUSTIVE TESTING!

Considers all possible inputs (executions)

STRAW-MAN IDEA: EXHAUSTIVE TESTING!



Input Domain

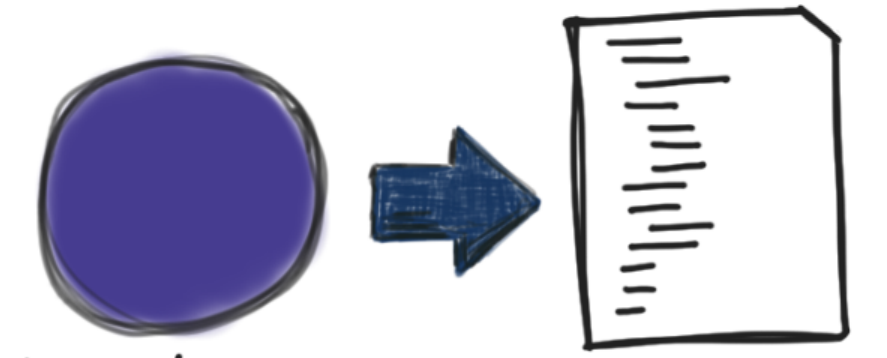
Software

How long would it take to exhaustively test the function

`printSum(int a, int b)`?

[ ]

STRAW-MAN IDEA: EXHAUSTIVE TESTING!



How long would it take to exhaustively test the function

`printSum(int a, int b)`?

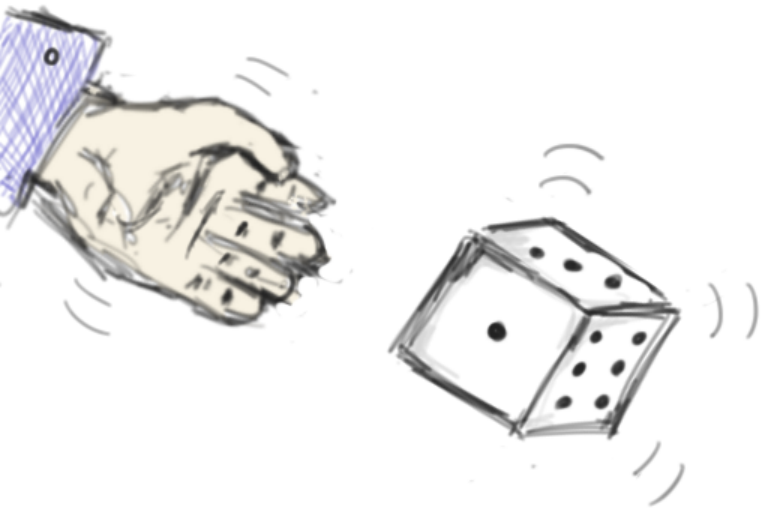
$2^{32} \times 2^{32} = 2^{64} \cong 10^{19}$ tests

1 test per nanosecond (10^9 tests/sec)

$\Rightarrow 10^{10}$ seconds

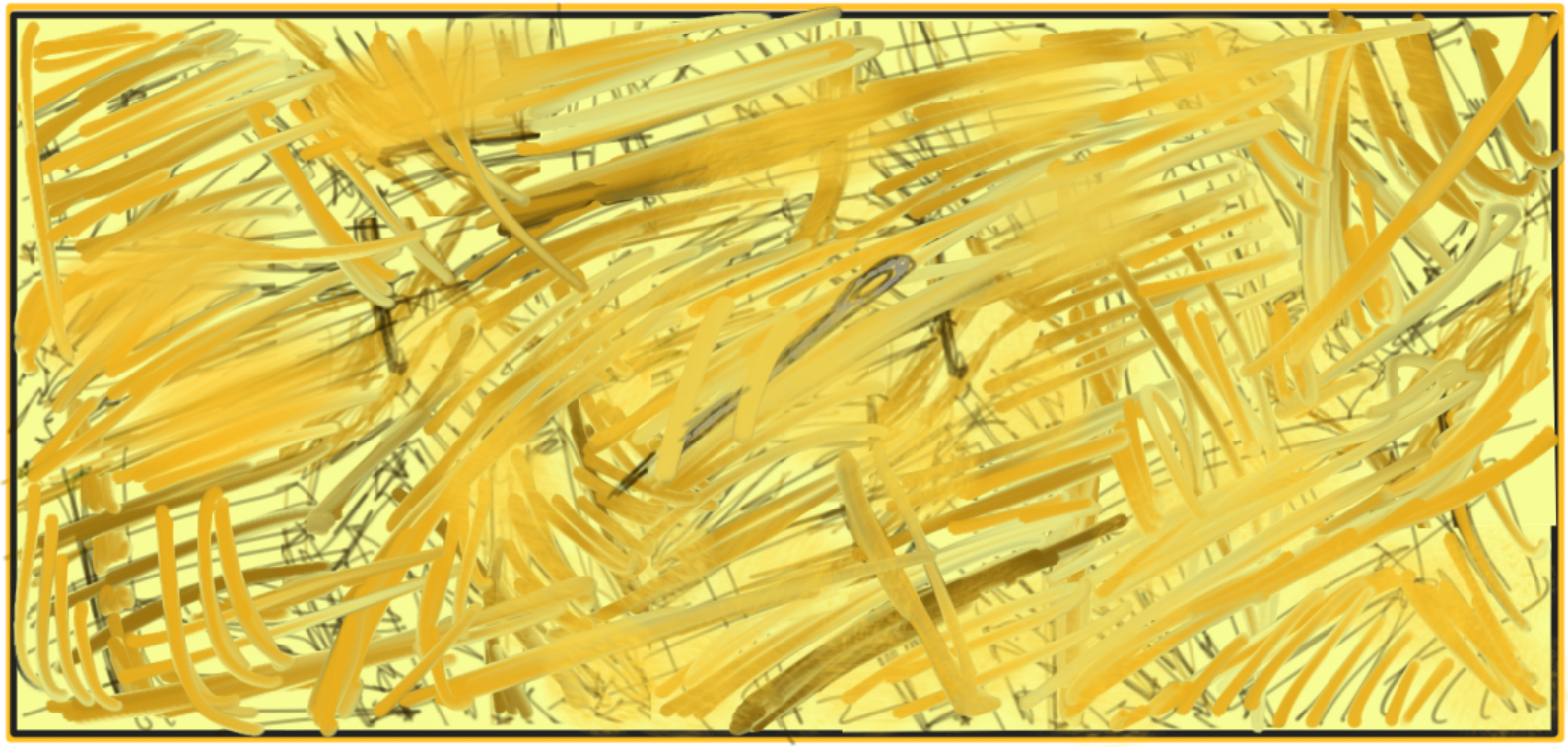
[~600 years]

RANDOM TESTING

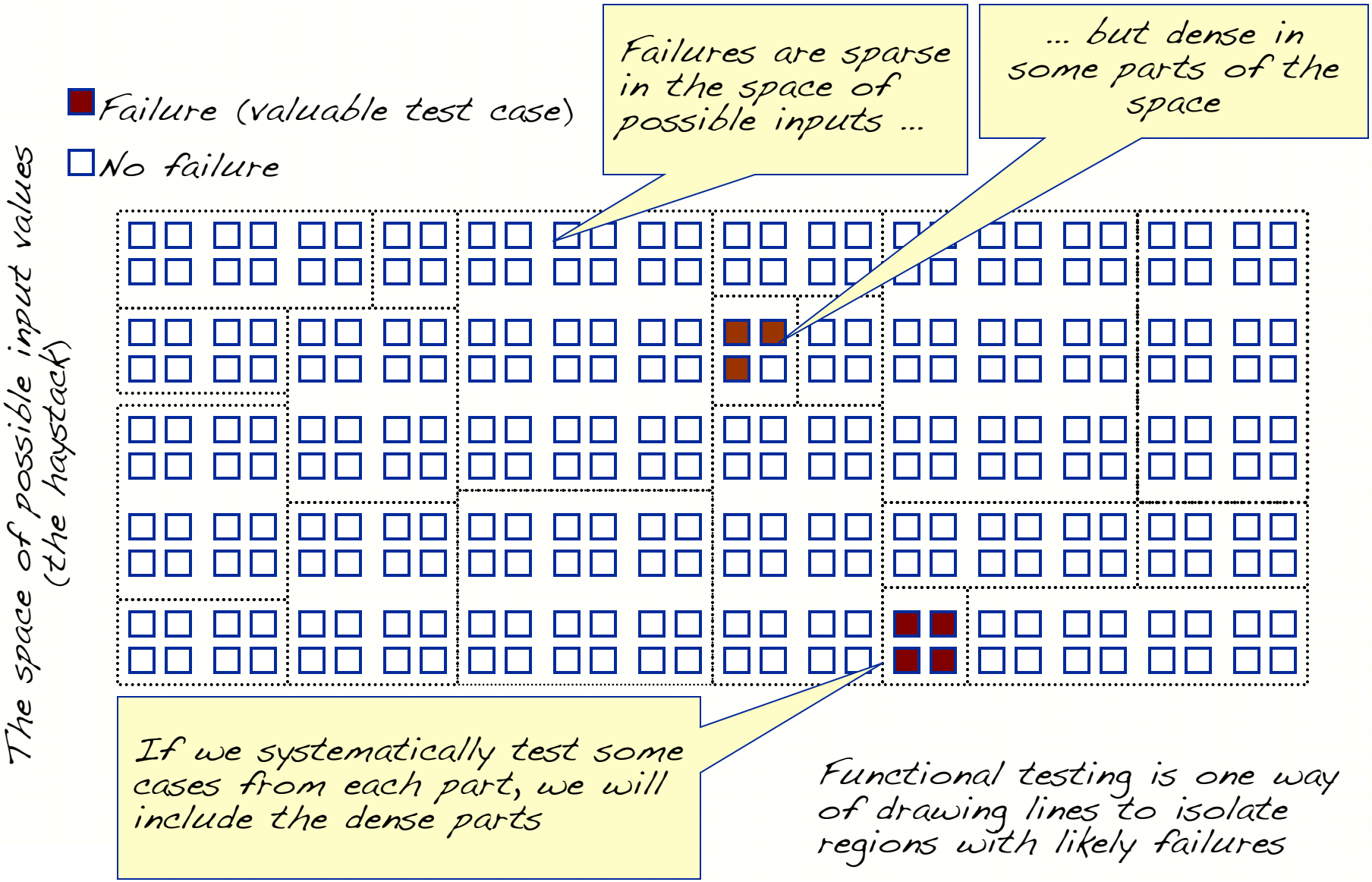


- pick inputs uniformly
- all inputs considered equal
- no designer bias

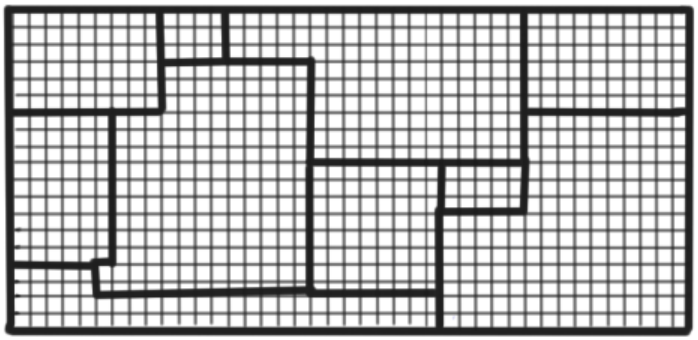
SO WHY NOT RANDOM ?



Systematic Partition Testing



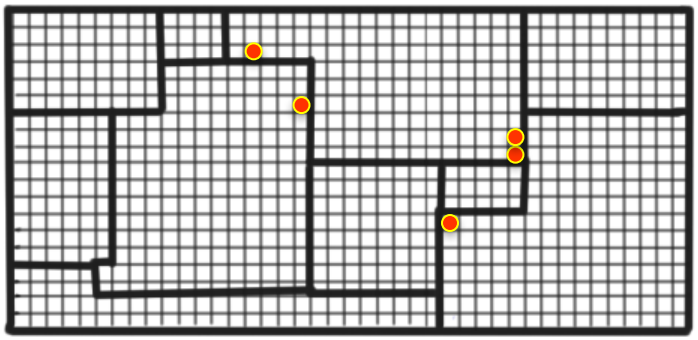
BOUNDARY VALUES



Basic idea

Errors tend to occur at the boundary of a (sub) domain

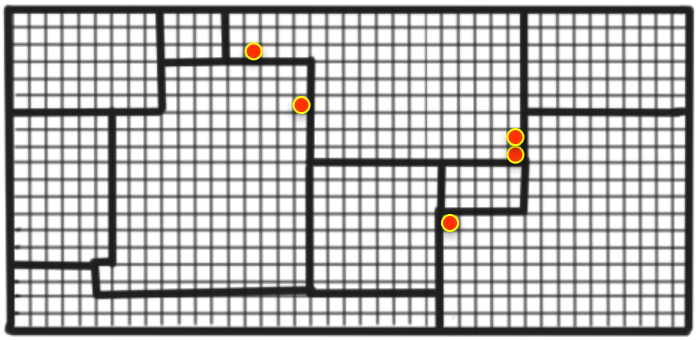
BOUNDARY VALUES



Basic idea

Errors tend to occur at the boundary of a (sub) domain

BOUNDARY VALUES

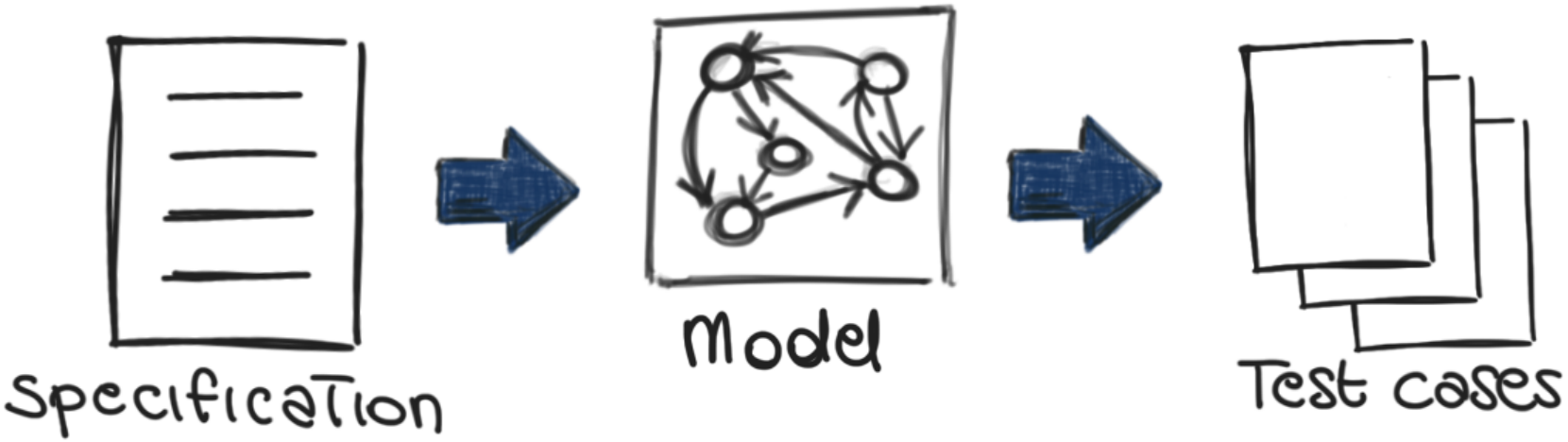


Basic idea

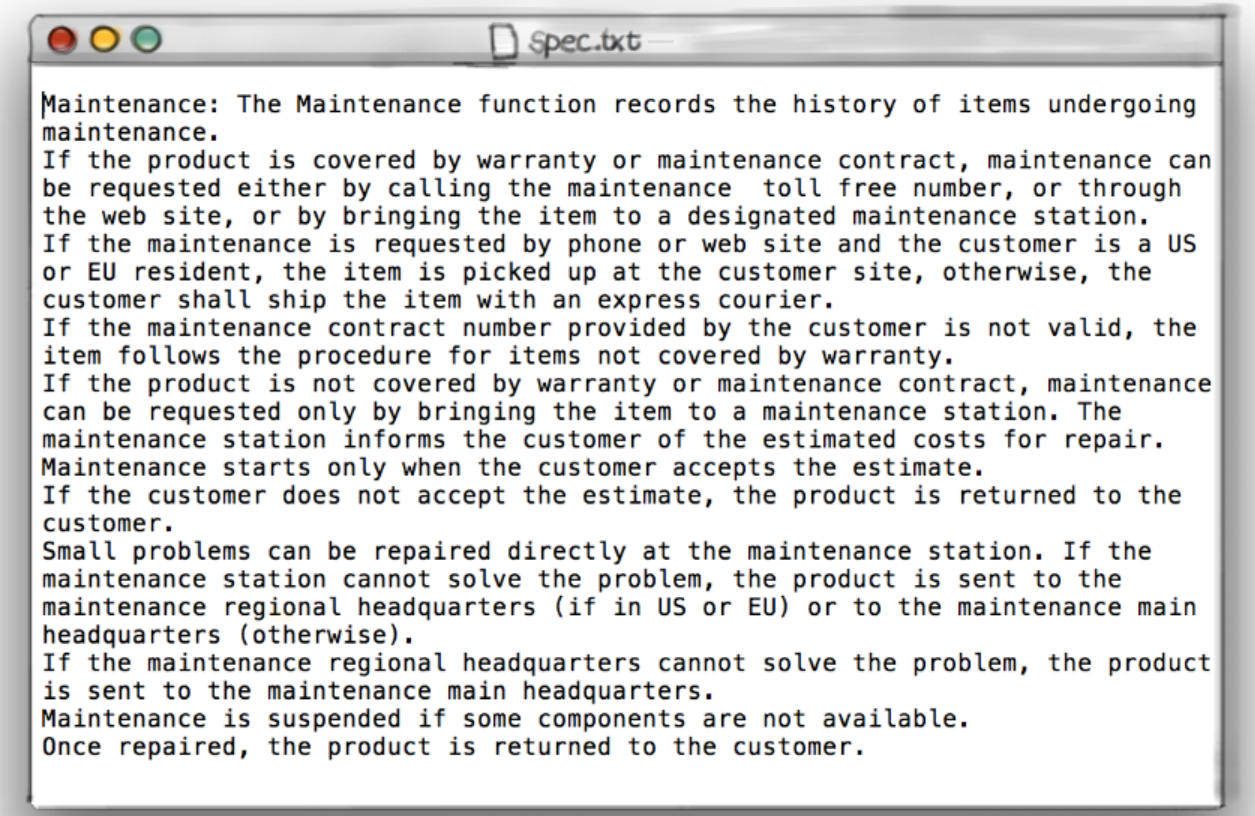
Errors tend to occur at the boundary of a (sub) domain

⇒ Select inputs at these boundaries

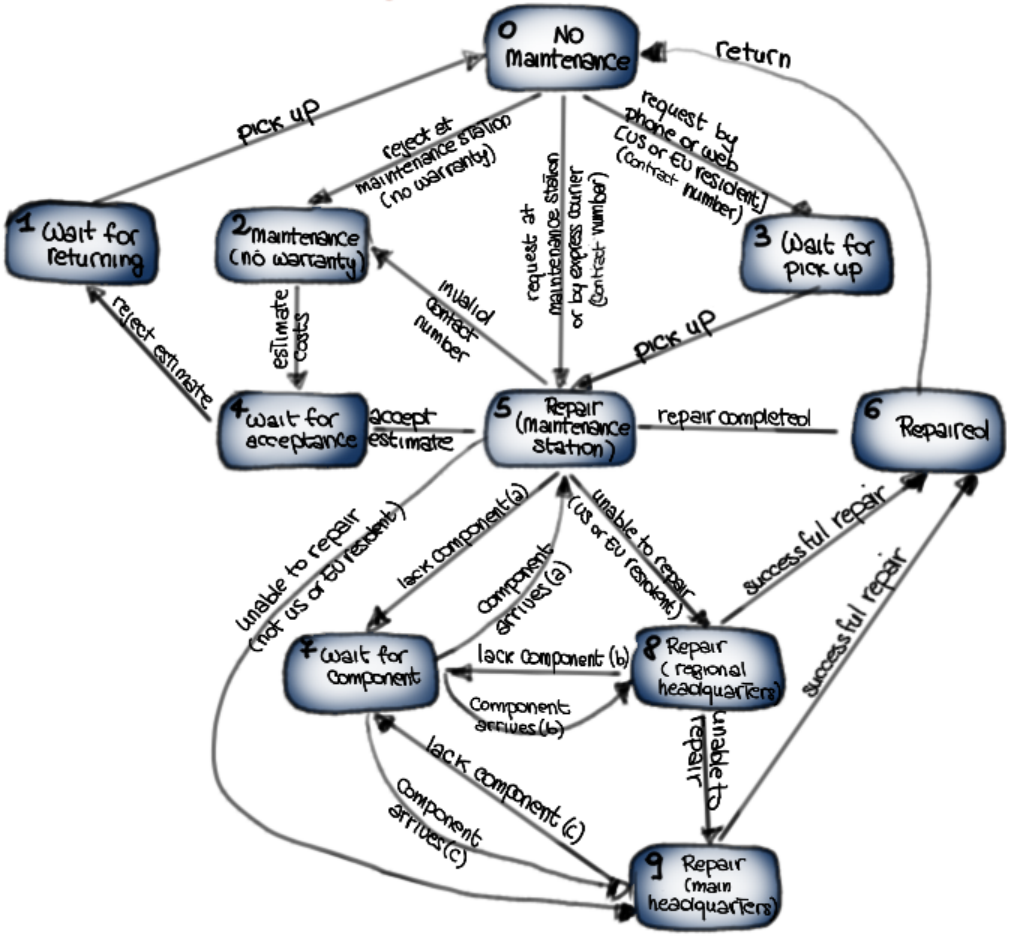
MODEL-BASED TESTING



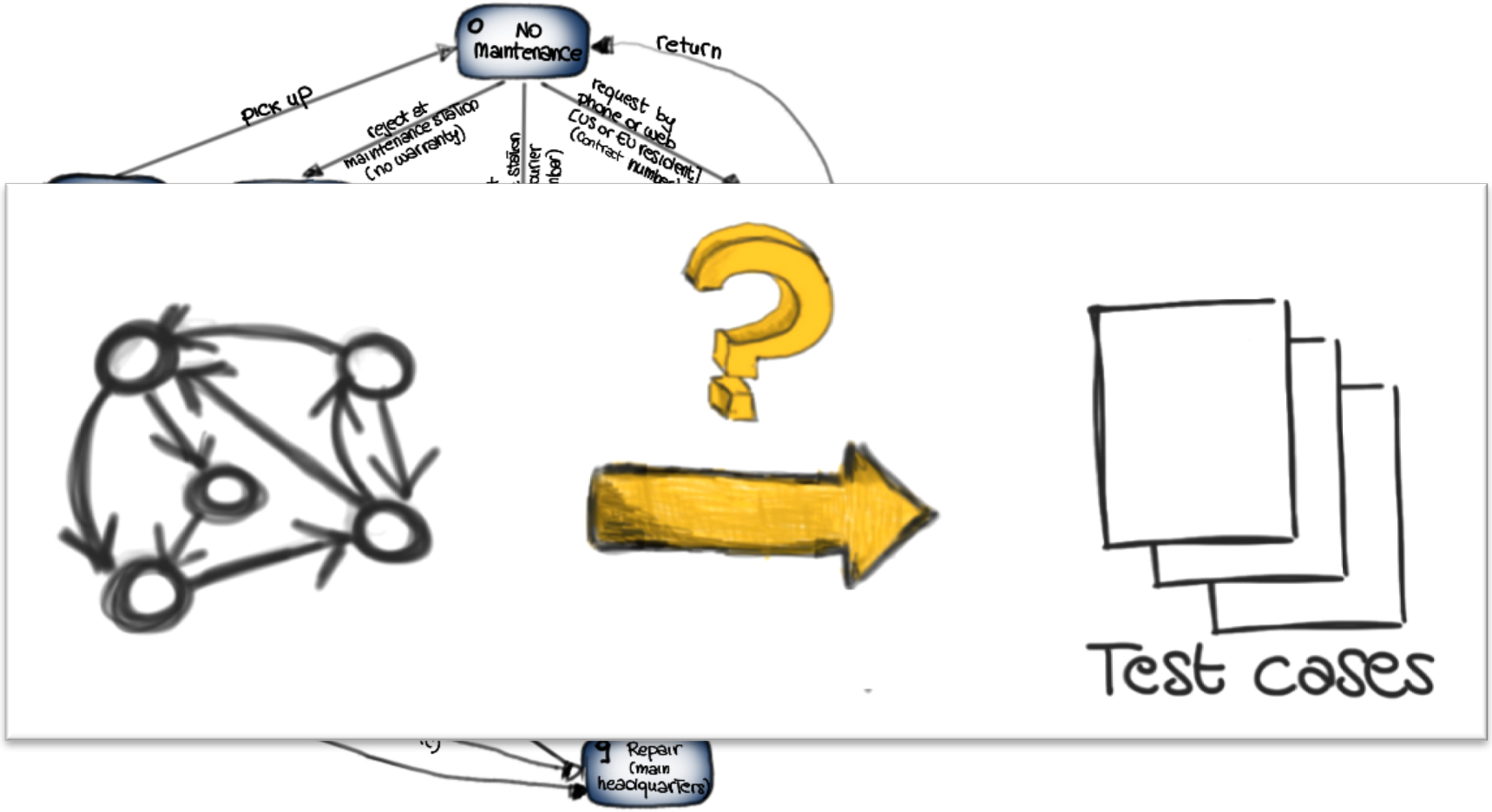
FROM AN INFORMAL SPECIFICATION...



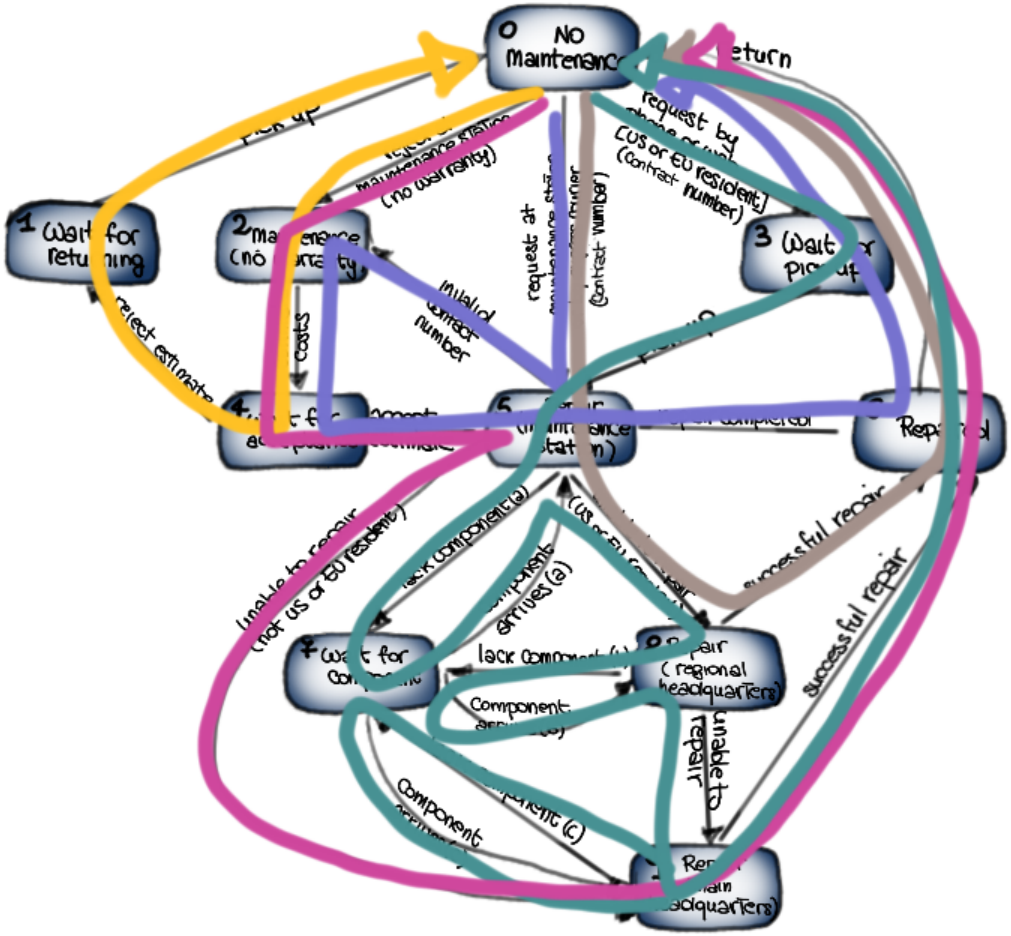
... TO A FINITE STATE MACHINE



TO A SET OF TEST CASES

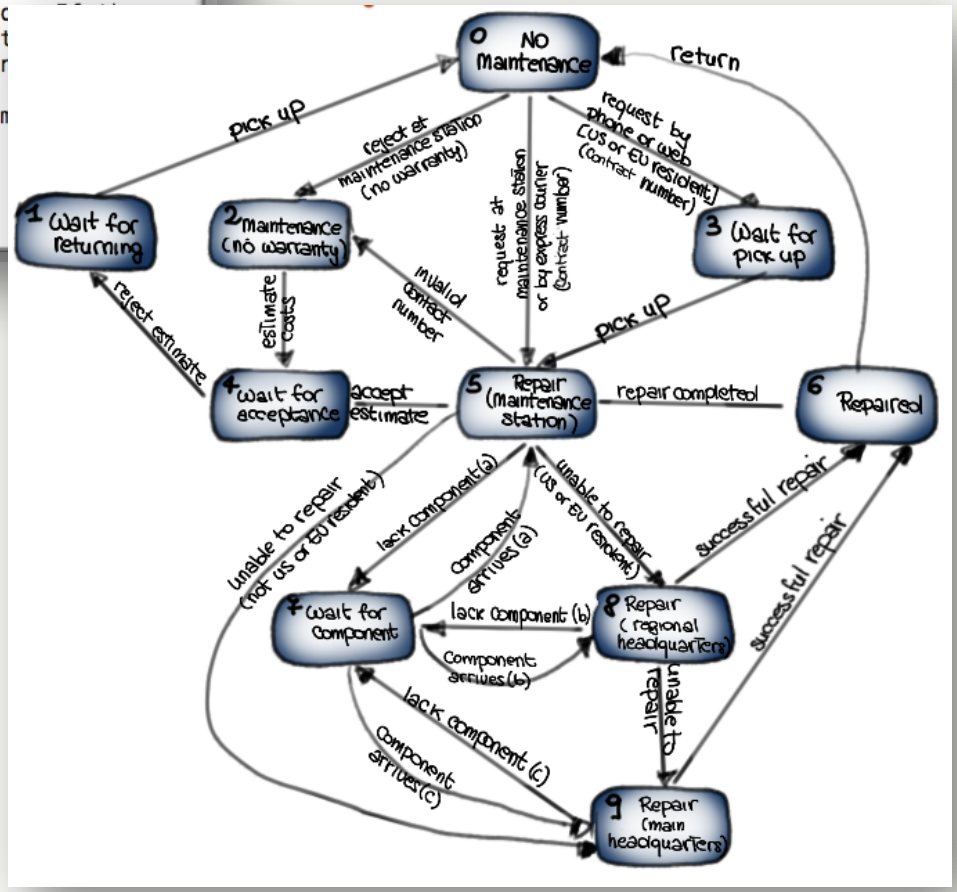


TO A SET OF TEST CASES



spec.txt

Maintenance: The Maintenance function records the history of items undergoing maintenance.
If the product is covered by warranty or maintenance contract, maintenance can be requested either by calling the maintenance toll free number, or through the web site, or by bringing the item to a designated maintenance station.
If the maintenance is requested by phone or web site and the customer is a US or EU resident, the item is picked up at the customer site, otherwise, the customer shall ship the item with an express courier.
If the maintenance contract number provided by the customer is not valid, the item follows the procedure for items not covered by warranty.
If the product is not covered by warranty or maintenance contract, maintenance can be requested only by bringing the item to a maintenance station. The maintenance station informs the customer of the estimated costs for repair.
Maintenance starts only when the customer accepts the estimate.
If the customer does not accept the estimate, the product is returned to the customer.
Small problems can be repaired directly at the maintenance statio
maintenance station cannot solve the problem, the product is sent
maintenance regional headquarters (if in US or EU) or to the mair
headquarters (otherwise).
If the maintenance regional headquarters cannot solve the problem
is sent to the maintenance main headquarters.
Maintenance is suspended if some components are not available.
Once repaired, the product is returned to the customer.



SOME CONSIDERATIONS

Applicability

- very general approach
- in UML, state machine are readily available

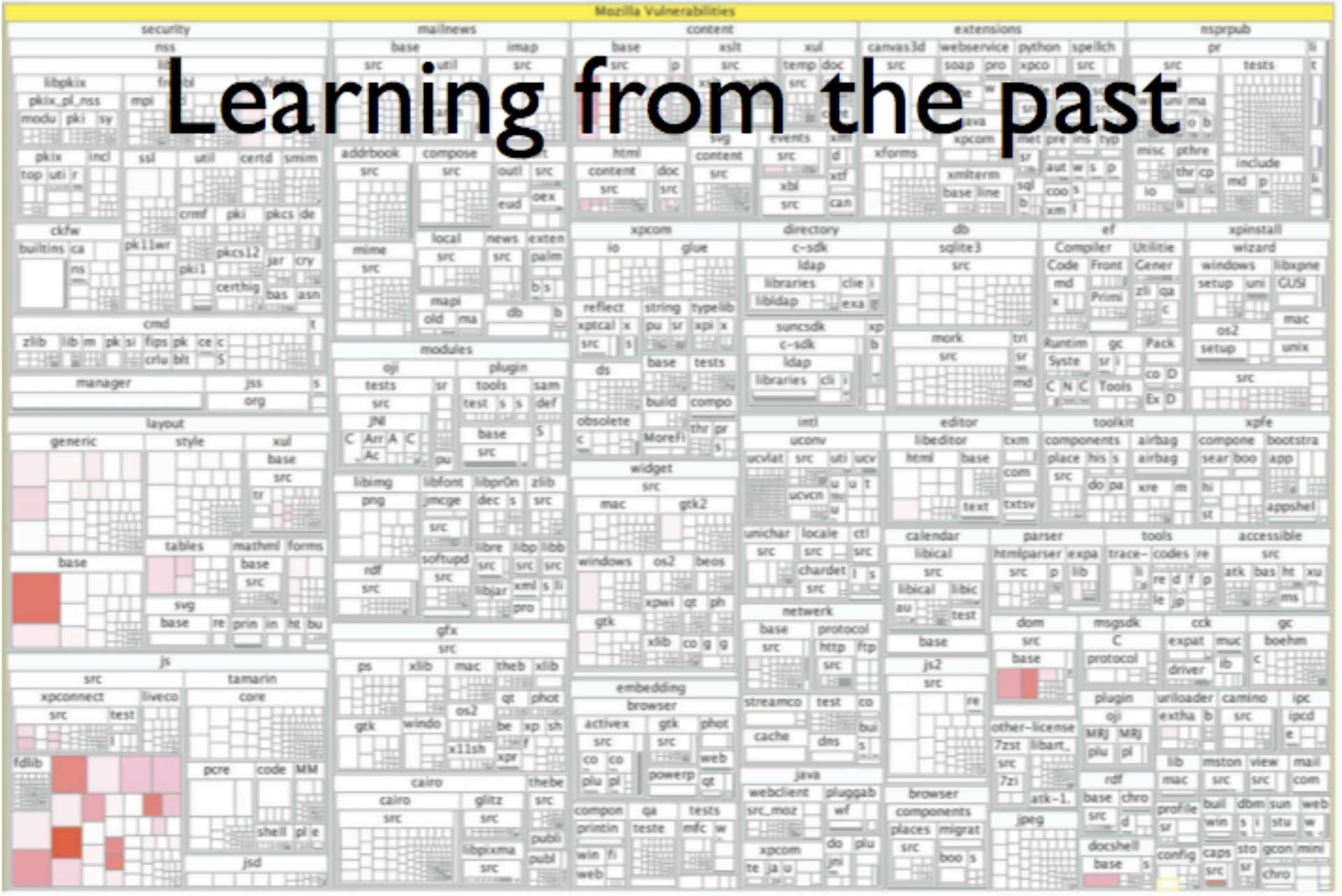
Abstraction is key

Many other approaches

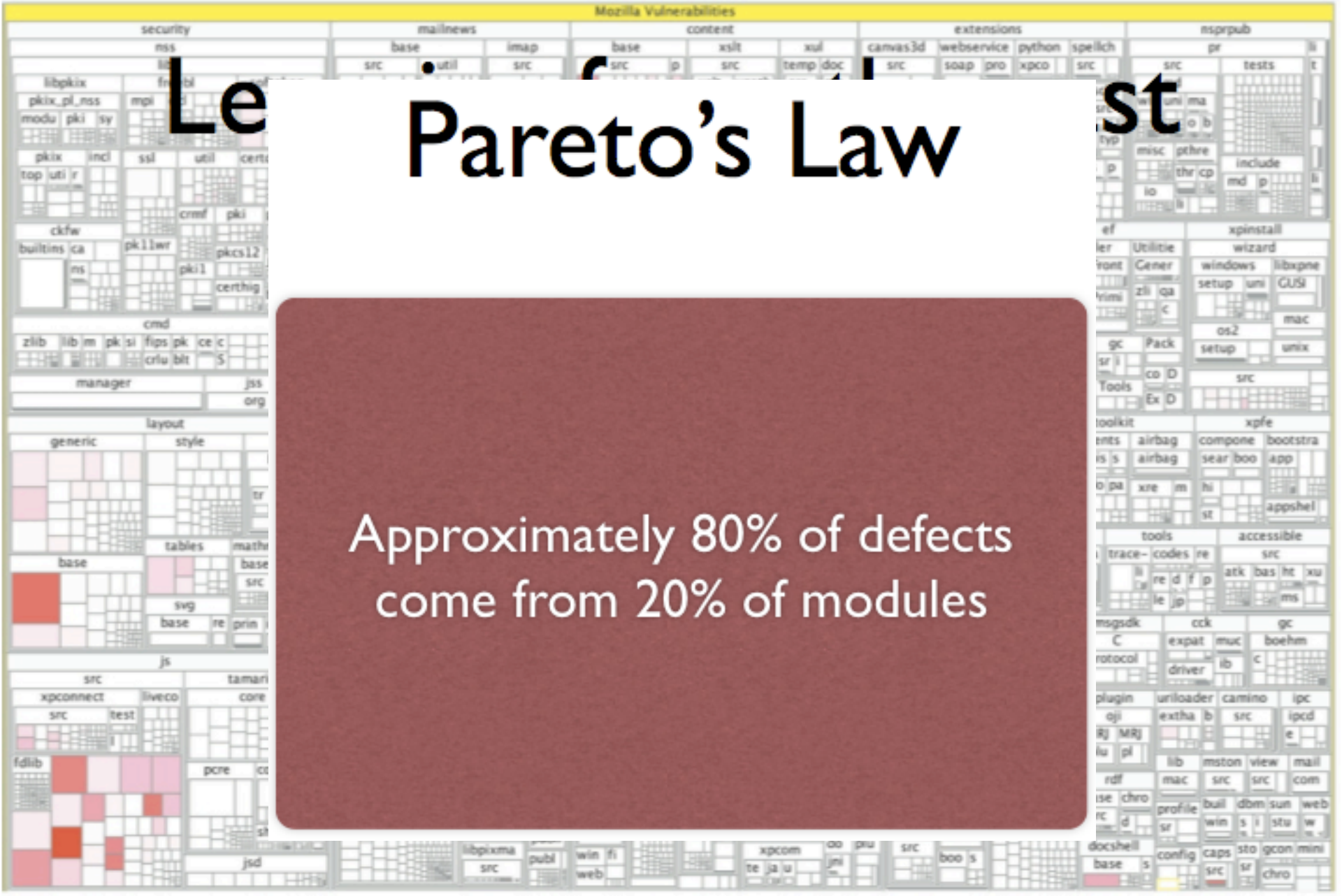
- decision tables
- flow graphs
- historical moolels

...

Historical models



Historical models



COVERAGE CRITERIA

Defined in terms of

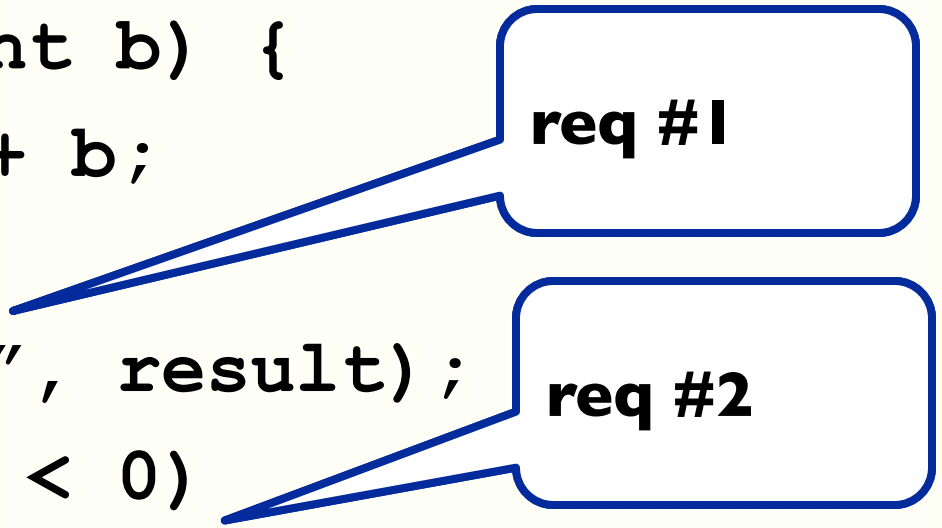
- test requirements

Result in

- test specifications
- test cases

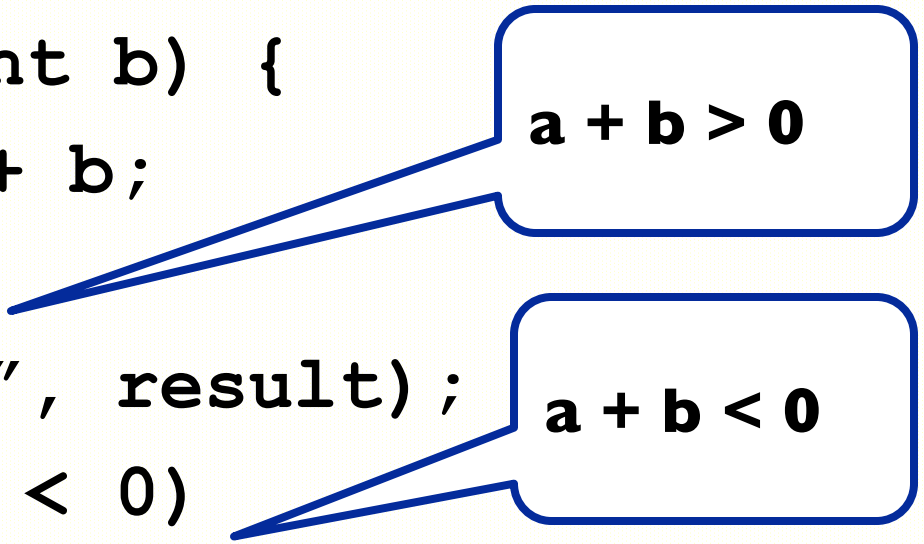
printSum: test requirements

```
printSum(int a, int b) {
   int result = a + b;
   if (result > 0)
     printcol("red", result);
   else if (result < 0)
     printcol("blue", result);
}
```



printSum: test specifications

```
printSum(int a, int b) {
   int result = a + b;
   if (result > 0)
     printcol("red", result);
   else if (result < 0)
     printcol("blue", result);
}
```



printSum: test cases

```
printSum(int a, int b) {
   int result = a + b;
   if (result > 0)
      printcol("red", result);
   else if (result < 0)
      printcol("blue", result);
}
```

Test Spec #1
a + b > 0

Test Spec #2
a + b < 0

#1 ((a=[3], b=[9]),(output color=[red], output value=[12]))

#2 ((a=[0], b=[-1]),(output color=[blue], output value=[-1]))

STATEMENT COVERAGE

Test requirements

Coverage measure

STATEMENT COVERAGE

Test requirements

statements in the program

Coverage measure

STATEMENT COVERAGE

Test requirements

statements in the program

Coverage measure

$$\frac{\text{number of executed statements}}{\text{total number of statements}}$$

printSum: statement coverage

```
printSum(int a, int b) {
   int result = a + b;
   if (result > 0)
     printcol("red", result);
   else if (result < 0)
     printcol("blue", result);
}
```

printSum: statement coverage

a == 3
b == 9

```
printSum(int a, int b) {
   int result = a + b;
   if (result > 0)
     printcol("red", result);
   else if (result < 0)
     printcol("blue", result);
}
```

Coverage: 0%

printSum: statement coverage

a == 3
b == 9

```
printSum(int a, int b) {
   int result = a + b;
   if (result > 0)
     printcol("red", result);
   else if (result < 0)
     printcol("blue", result);
}
```

Coverage: 71%

printSum: statement coverage

a == 3 b == 9	a == 0 b == -1

```
printSum(int a, int b) {
   int result = a + b;
   if (result > 0)
     printcol("red", result);
   else if (result < 0)
     printcol("blue", result);
}
```

Coverage: 71%

printSum: statement coverage

a == 3 b == 9	a == 0 b == -1

```
printSum(int a, int b) {
   int result = a + b;
   if (result > 0)
      printcol("red", result);
   else if (result < 0)
      printcol("blue", result);
}
```

Coverage: 100%

STATEMENT COVERAGE IN PRACTICE

Most used in industry

"Typical coverage" target is 80-90%.

Why don't we aim at 100%.

[ ]

printSum: statement coverage

a == 3 b == 9	a == 0 b == -1

```
printSum(int a, int b) {
   int result = a + b;
   if (result > 0)
     printcol("red", result);
   else if (result < 0)
     printcol("blue", result);
}
```

Coverage: 100%

printSum: statement coverage

a == 3 b == 9	a == 0 b == -1

```
printSum(int a, int b) {
   int result = a + b;
   if (result > 0)
     printcol("red", result);
   else if (result < 0)
     printcol("blue", result);
   [else do nothing]
}
```

Coverage: 100%

printSum: statement coverage

a == 3 b == 9	a == 0 b == -1

```
printSum(int a, int b) {
   int result = a + b;
   if (result > 0)
      printcol("red", result);
   else if (result < 0)
      printcol("blue", result);
   [else do nothing]
}
```

Coverage: 100%

BRANCH COVERAGE

Test requirements

Coverage measure

BRANCH COVERAGE

Test requirements

branches in the program

Coverage measure

BRANCH COVERAGE

Test requirements

branches in the program

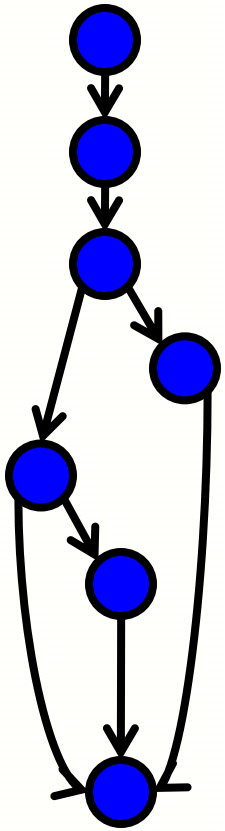
Coverage measure

$$\frac{\text{number of executed branches}}{\text{total number of branches}}$$

printSum: branch coverage

a == 3 b == 9	a == 0 b == -1

```
printSum(int a, int b) {
  int result = a + b;
  if (result > 0)
    printcol("red", result);
  else if (result < 0)
    printcol("blue", result);
  [else do nothing]}
```

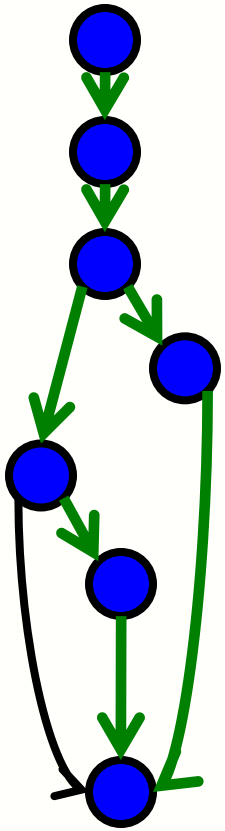


Coverage: ?

printSum: branch coverage

a == 3 b == 9	a == 0 b == -1

```
printSum(int a, int b) {
  int result = a + b;
  if (result > 0)
    printcol("red", result);
  else if (result < 0)
    printcol("blue", result);
  [else do nothing]}
```

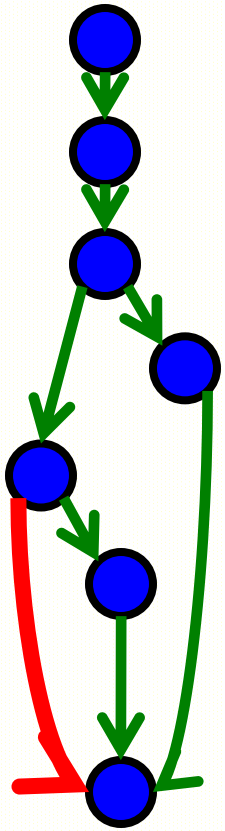


Coverage: 75%

printSum: branch coverage

a == 3 b == 9	a == 0 b == -1	a == -5 b == 5

```
printSum(int a, int b) {
  int result = a + b;
  if (result > 0)
    printcol("red", result);
  else if (result < 0)
    printcol("blue", result);
  [else do nothing]}
```

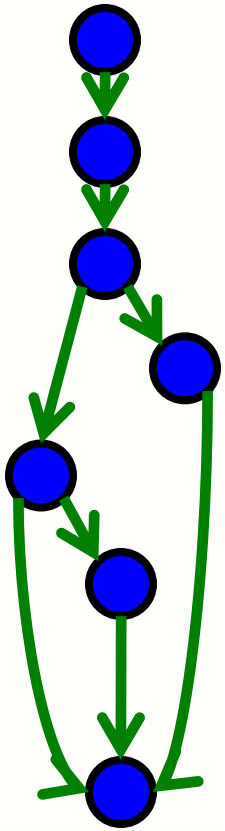


Coverage: 75%

printSum: branch coverage

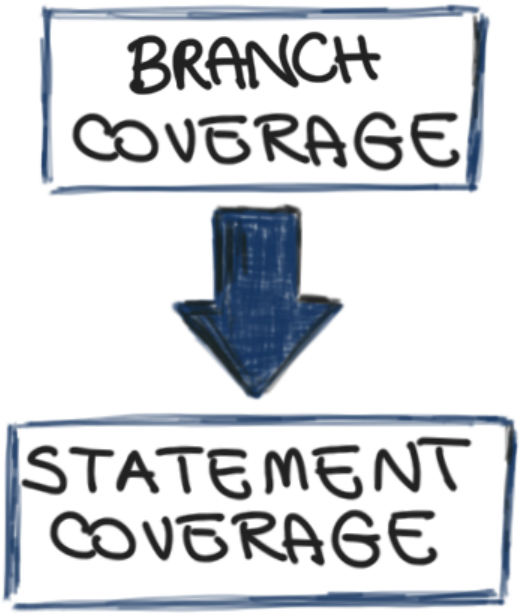
a == 3 b == 9	a == 0 b == -1	a == -5 b == 5

```
printSum(int a, int b) {
  int result = a + b;
  if (result > 0)
    printcol("red", result);
  else if (result < 0)
    printcol("blue", result);
  [else do nothing]}
```



Coverage: 100%

TEST CRITERIA SUBSUMPTION

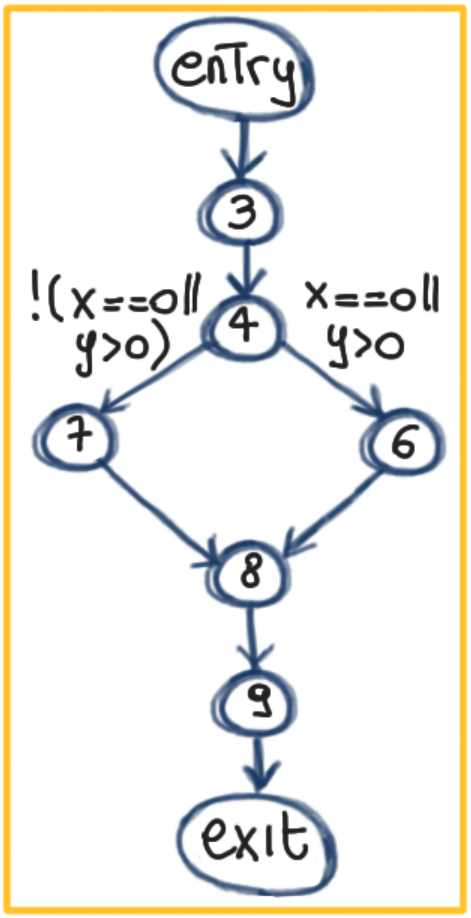


LET'S CONSIDER ANOTHER EXAMPLE

```
1. void main(){
2.    float x,y;
3.    read(x);
4.    read(y);
5.    if((x==0)||(y>0))
6.       y=y/x;
7.    else x=y+2;
8.    write(x);
9.    write(y);
10. }
```

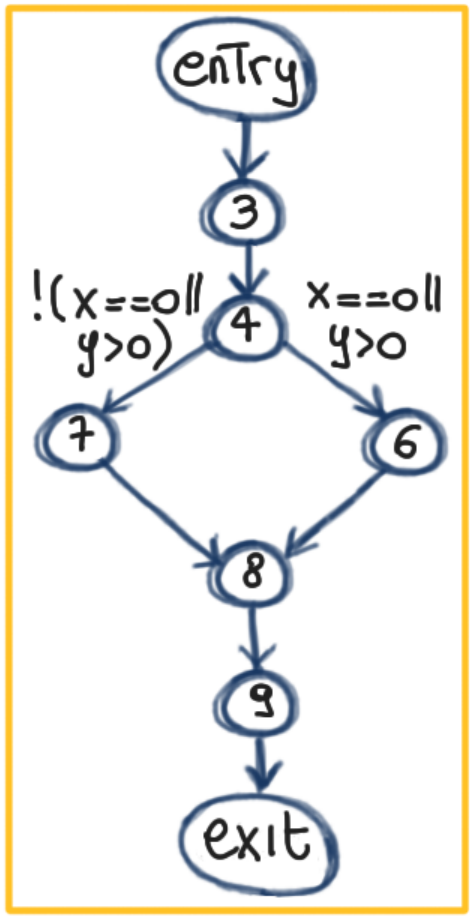
LET'S CONSIDER ANOTHER EXAMPLE

```
1. void main(){
2.    float x,y;
3.    read(x);
4.    read(y);
5.    if((x==0)||(y>0))
6.       y=y/x;
7.    else x=y+2;
8.    write(x);
9.    write(y);
10. }
```



LET'S CONSIDER ANOTHER EXAMPLE

```
1. void main(){
2.    float x,y;
3.    read(x);
4.    read(y);
5.    if((x==0)||(y>0))
6.       y=y/x;
7.    else x=y+2;
8.    write(x);
9.    write(y);
10. }
```

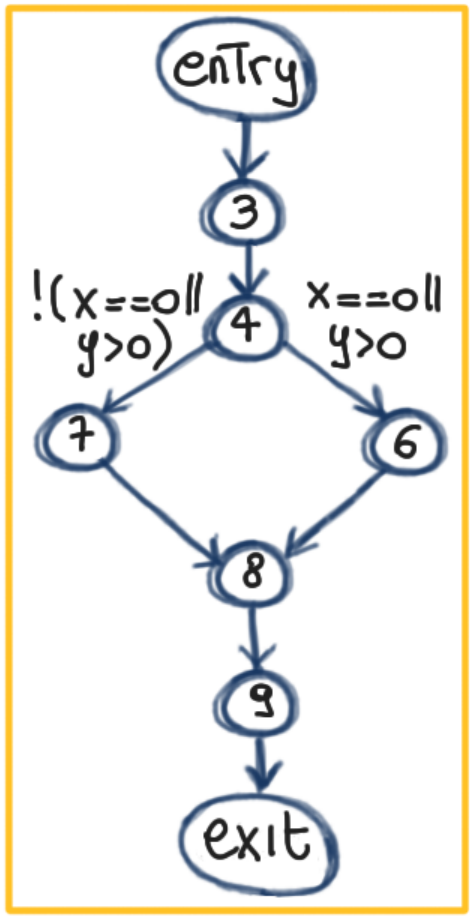


Tests: (x=5, y=6)
(x=5, y=-5)

Branch coverage:

LET'S CONSIDER ANOTHER EXAMPLE

```
1. void main(){
2.    float x,y;
3.    read(x);
4.    read(y);
5.    if((x==0)||(y>0))
6.       y=y/x;
7.    else x=y+2;
8.    write(x);
9.    write(y);
10. }
```

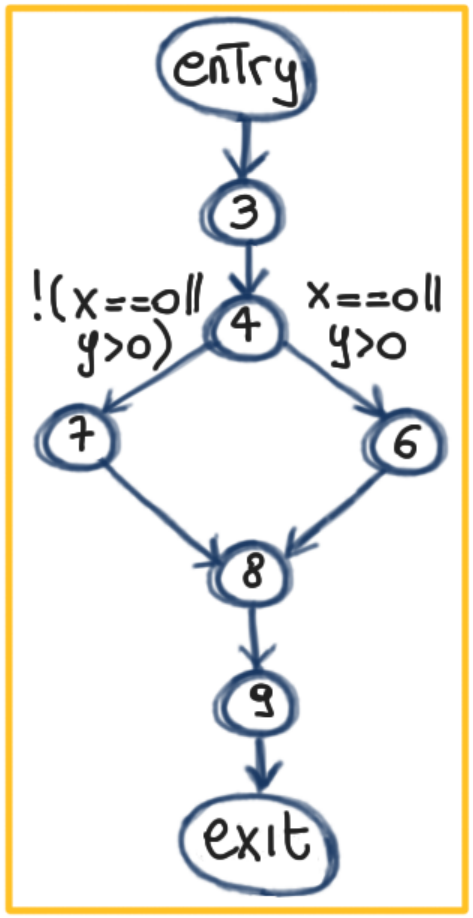


Tests: (x=5, y=6)
(x=5, y=-5)

Branch coverage: 100%

LET'S CONSIDER ANOTHER EXAMPLE

```
1. void main(){
2.   float x,y;
3.   read(x);
4.   read(y);
5.   if((x==0)||(y>0))
6.       y=y/x;
7.   else x=y+2;
8.   write(x);
9.   write(y);
10. }
```



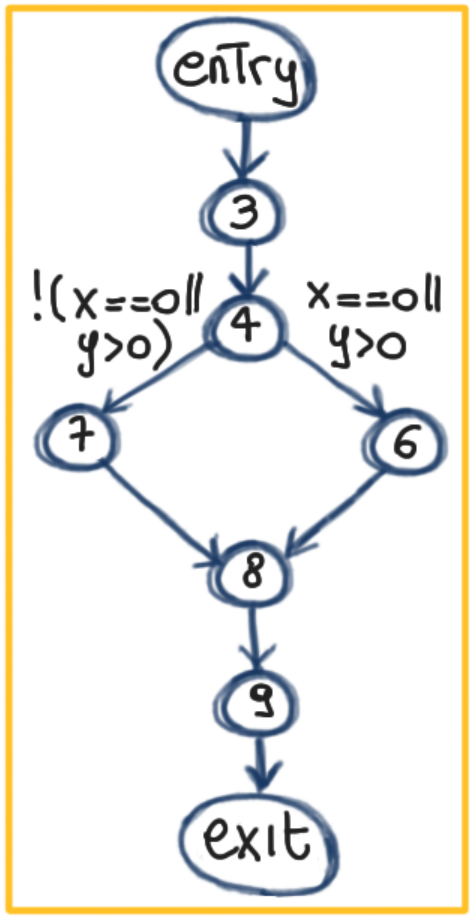
Tests: (x=5, y=6)
(x=5, y=-5)

Branch coverage: 100%

How can we be more thorough?

LET'S CONSIDER ANOTHER EXAMPLE

```
1. void main(){
2.    float x,y;
3.    read(x);
4.    read(y);
5.    if((x==0)||(y>0))
6.       y=y/x;
7.    else x=y+2;
8.    write(x);
9.    write(y);
10. }
```



Tests: (x=5, y=5)
(x=5, y=-5)

Branch coverage: 100%

How can we be more thorough?

We can make each condition T and F

CONDITION COVERAGE

Test requirements

Coverage measure

CONDITION COVERAGE

Test requirements

individual conditions in the program

Coverage measure

CONDITION COVERAGE

Test requirements

individual conditions in the program

Coverage measure

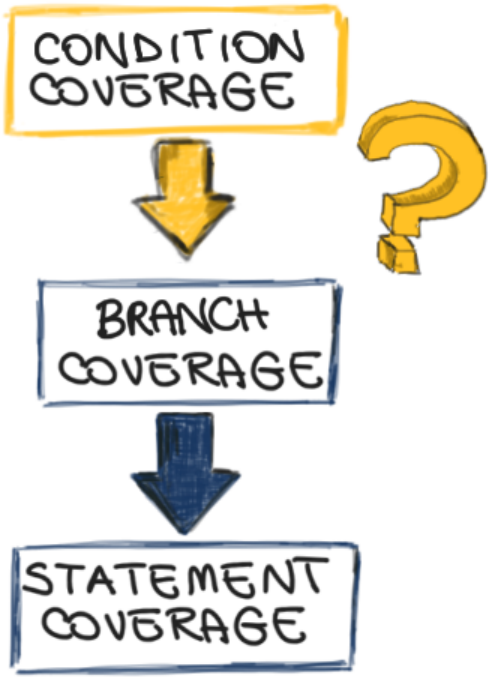
$$\frac{\text{number of conditions that are both T and F}}{\text{total number of conditions}}$$

SUBSUMPTION

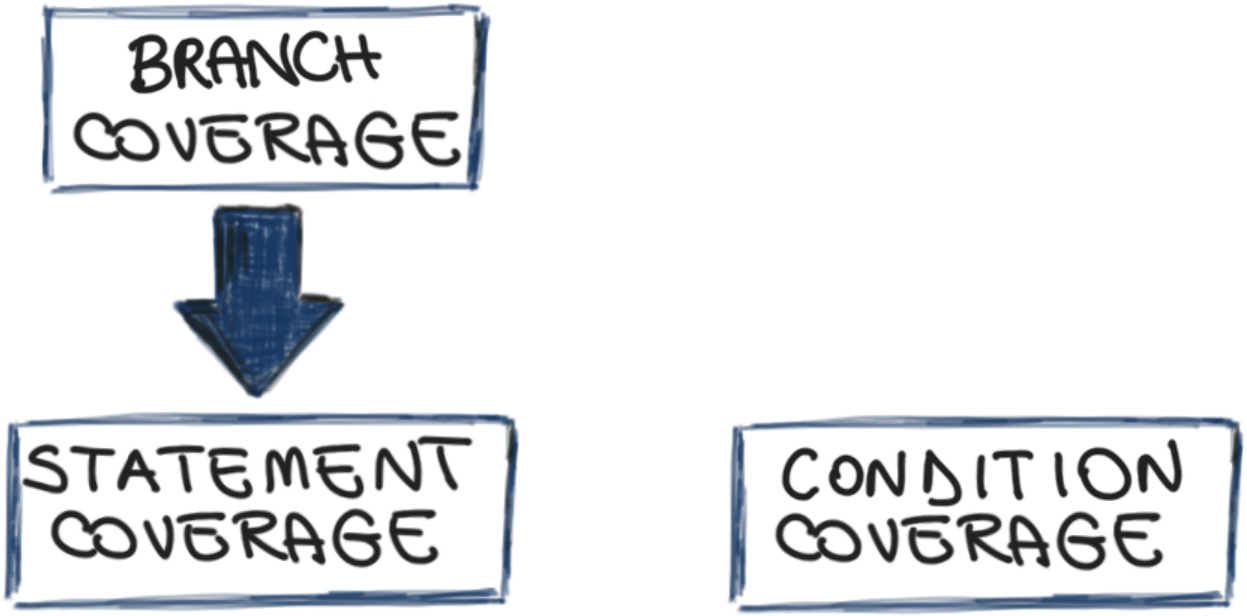
Does condition coverage imply branch coverage?

[ ] Yes

[ ] No

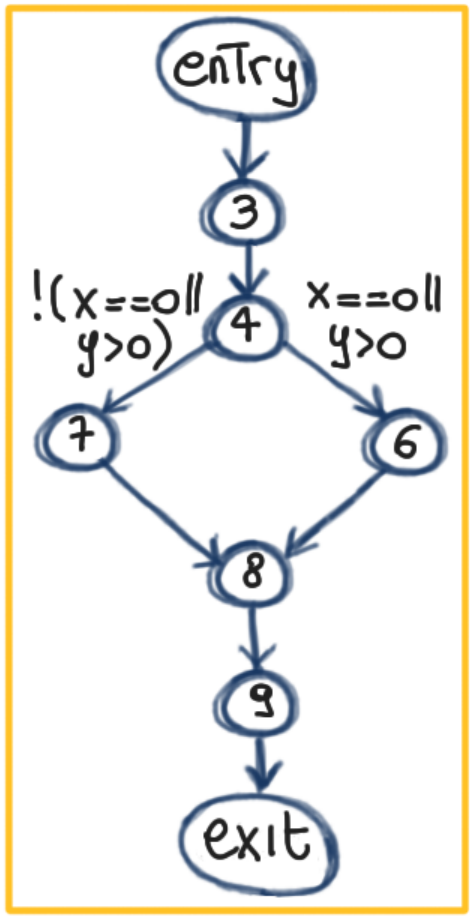


TEST CRITERIA SUBSUMPTION



LET'S CONSIDER OUR LAST EXAMPLE

```
1. void main(){
2.    float x,y;
3.    read(x);
4.    read(y);
5.    if((x==0)||(y>0))
6.       y=y/x;
7.    else x=y+2;
8.    write(x);
9.    write(y);
10. }
```



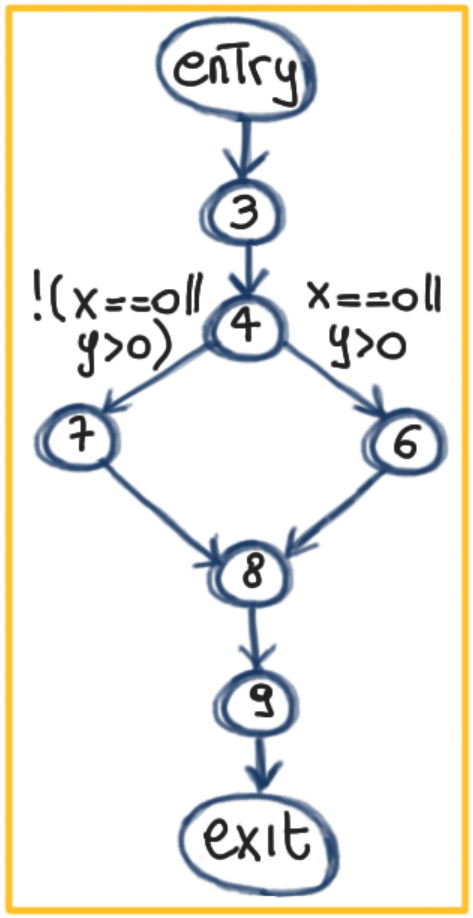
Tests: (x=0, y=-5)
(x=5, y=5)

Condition coverage: 100%

What about branch coverage?

LET'S CONSIDER OUR LAST EXAMPLE

```
1. void main(){
2.    float x,y;
3.    read(x);
4.    read(y);
5.    if((x==0)||(y>0))
6.       y=y/x;
7.    else x=y+2;
8.    write(x);
9.    write(y);
10. }
```



Tests: (x=0, y=-5)
(x=5, y=5)

Condition coverage: 100%

What about branch coverage? 50%.

BRANCH AND CONDITION COVERAGE (DECISION)

Test requirements	branches and individual conditions in the program
Coverage measure	computed considering both coverage measures

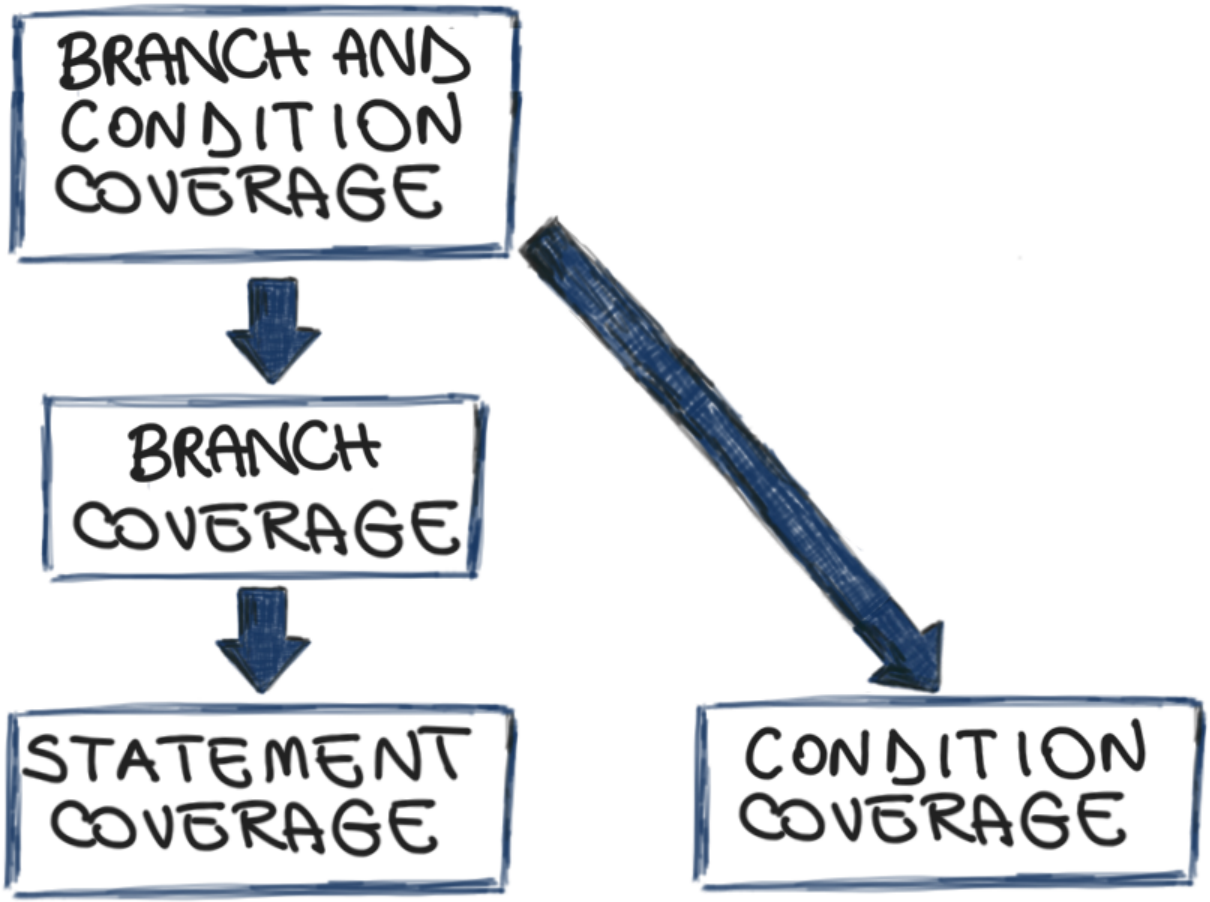
SUBSUMPTION

Does branch and condition coverage imply branch coverage?

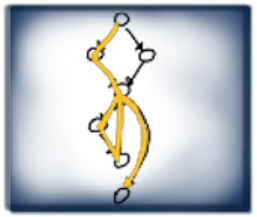
[ ] Yes

[ ] No

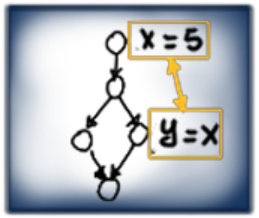
TEST CRITERIA SUBSUMPTION



OTHER CRITERIA



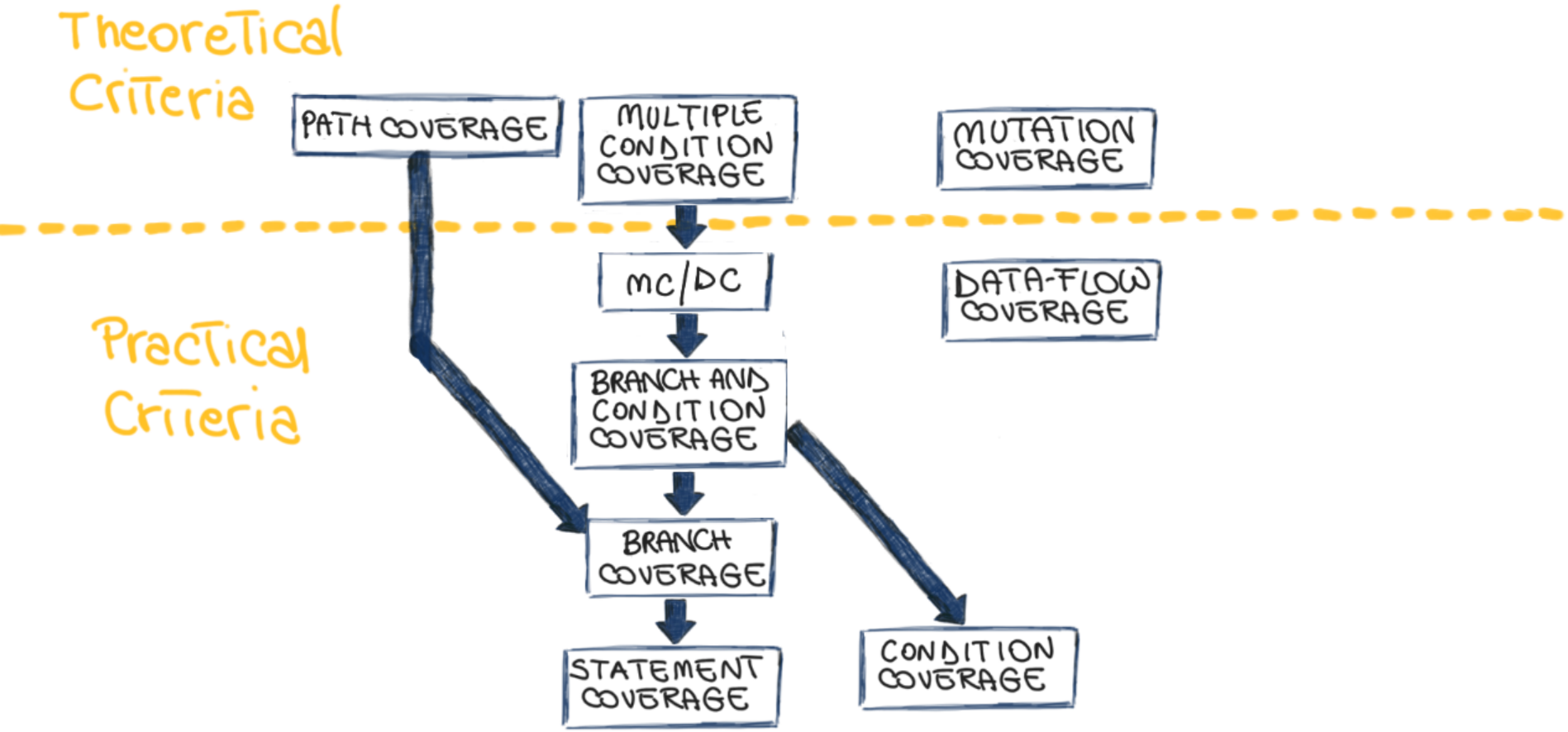
Path coverage



Data-flow coverage

Mutation coverage

TEST CRITERIA SUBSUMPTION



Write a faulty program P and two test suites T1 and T2 for P, such that:

1. T1 achieves 100% branch coverage but does not reveal the fault in P
2. T2 achieves 100% statement coverage, does not achieve 100% branch coverage and reveals the fault in P

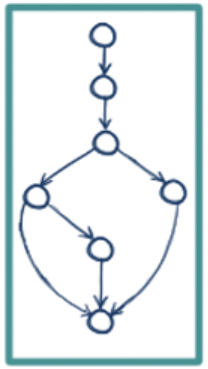
Write a faulty program P such that:

1. Any test suite that achieves 100% statement coverage reveals the fault in P
2. It is possible to write a test suite that achieves 100% branch coverage and does not reveal the fault in P

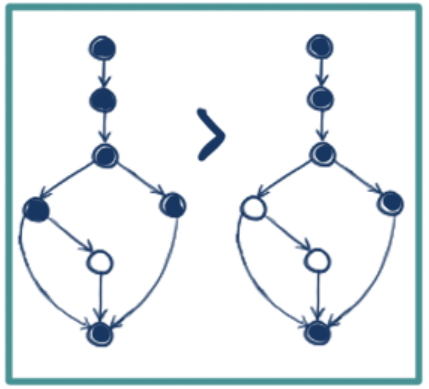
Write a faulty program P such that:

1. Any test suite that achieves 100% path coverage reveals the fault in P
2. It is possible to write a test suite that achieves 100% branch coverage and does not reveal the fault in P

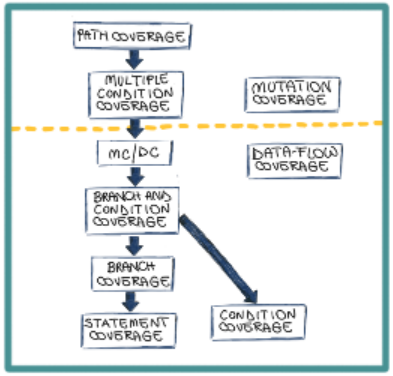
WHITE-BOX TESTING SUMMARY



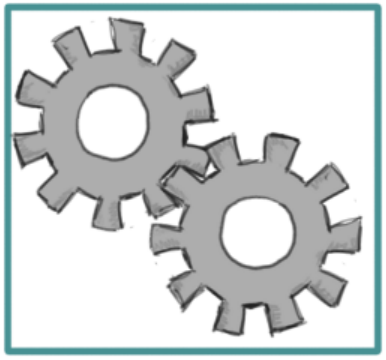
Works on a formal model



Comparable



Two broad classes:
- practical
- Theoretical



Fully automatable