Summary

Software Testing and Analysis

Process, Principles, and Techniques

JUNBEOM YOO

Dependable Software Laboratory
KONKUK University

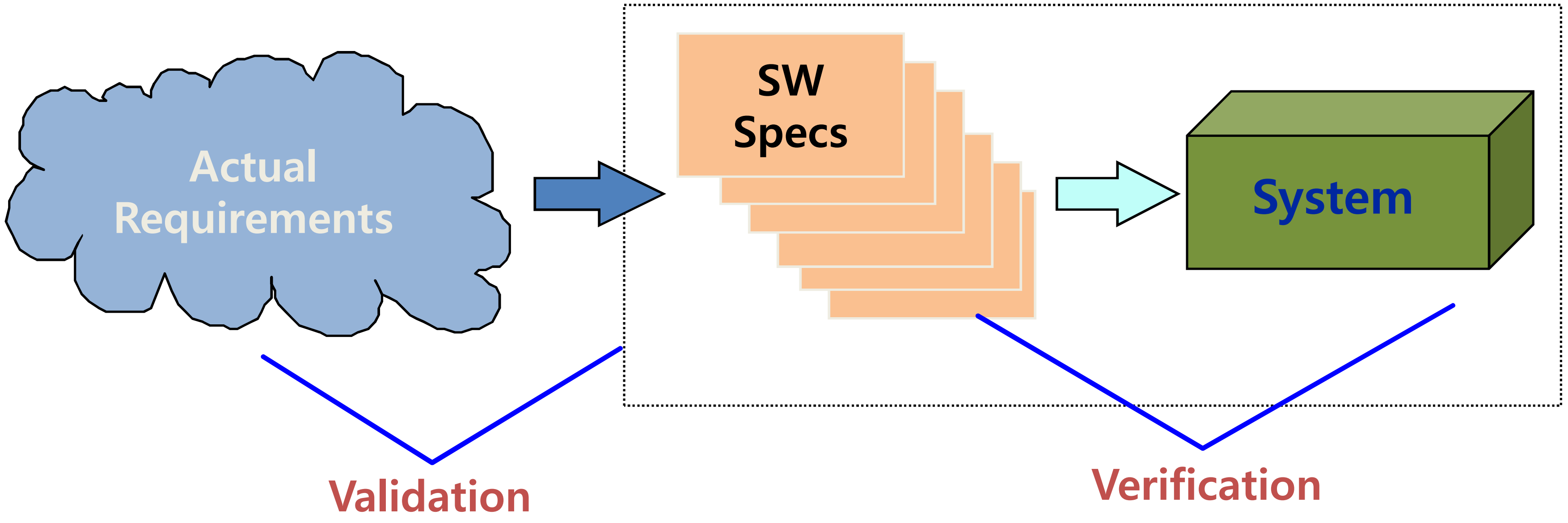
http://dslab.konkuk.ac.kr

Ver. 1.0 (2012.11)

Part I. Fundamentals of Test and Analysis

Verification and Validation

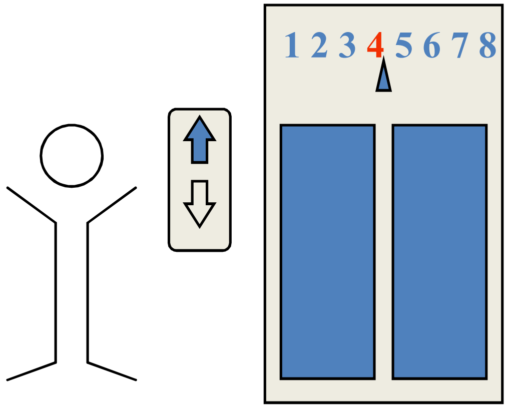
- Validation: "Does the software system meets the user's real needs?"
  - Are we building the right software?
- Verification: "Does the software system meets the requirements specifications?"
  - Are we building the software right?



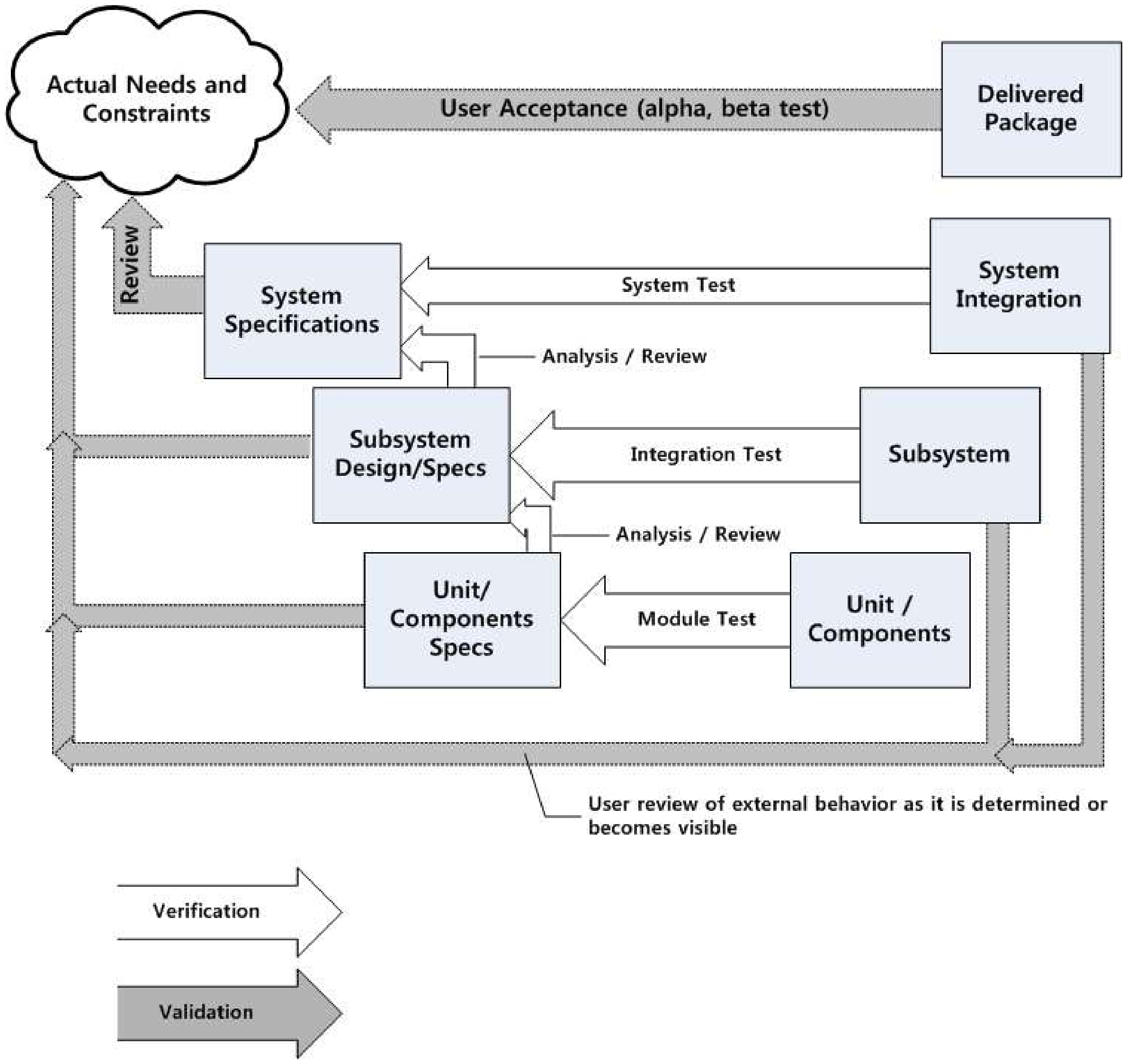
V&V Depends on the Specification

- Unverifiable (but validatable) specification: "If a user presses a request button at floor i, an available elevator must arrive at floor i soon."

- Verifiable specification: "If a user presses a request button at floor i, an available elevator must arrive at floor i within 30 seconds"

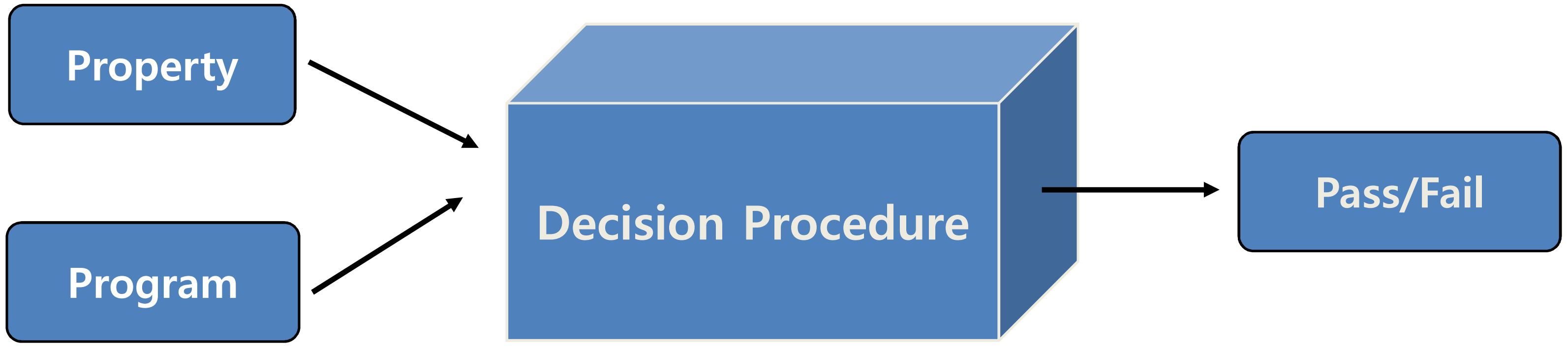


V-Model of V&V Activities

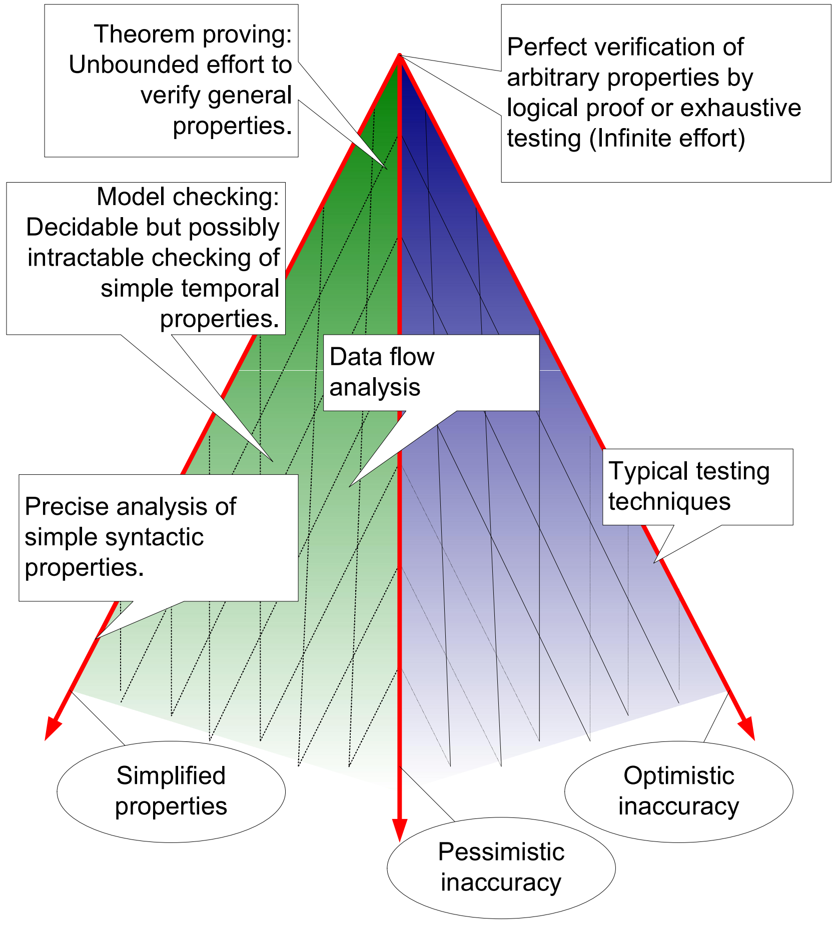


Undeciability of Correctness Properties

- Correctness properties are not decidable.
  - Halting problem can be embedded in almost every property of interest.



Verification Trade-off Dimensions


Theorem proving: Unbounded effort to verify general properties.
Perfect verification of arbitrary properties by logical proof or exhaustive testing (Infinite effort)
Model checking: Decidable but possibly intractable checking of simple temporal properties.
Data flow analysis
Typical testing techniques
Precise analysis of simple syntactic properties.
Simplified properties
Optimistic inaccuracy
Pessimistic inaccuracy

- Optimistic inaccuracy
  - We may accept some programs that do not possess the property.
  - It may not detect all violations.
  - Example: Testing
- Pessimistic inaccuracy
  - It is not guaranteed to accept a program even if the program does possess the property being analyzed, because of false alarms.
  - Example: Automated program analysis
- Simplified properties
  - It reduces the degree of freedom by simplifying the property to check.
  - Example: Model Checking

Software Quality and Process

- Qualities cannot be added after development
  - Quality results from a set of inter-dependent activities.
  - Analysis and testing are crucial but far from sufficient.

- Testing is not a phase, but a lifestyle
  - Testing and analysis activities occur from early in requirements engineering through delivery and subsequent evolution.
  - Quality depends on every part of the software process.

- An essential feature of software processes is that software test and analysis is thoroughly integrated and not an afterthought

Quality Process

- Quality process
  - A set of activities and responsibilities
    - Focused on ensuring adequate dependability
    - Concerned with project schedule or with product usability

- Quality process provides a framework for
  - Selecting and arranging A&T activities
  - Considering interactions and trade-offs with other important goals

Planning and Monitoring

- Quality process
  - A&T planning
  - Balances several activities across the whole development process
  - Selects and arranges them to be as cost-effective as possible
  - Improves early visibility

- A&T planning is integral to the quality process.
  - Quality goals can be achieved only through careful planning.

Quality Goals

- Goal must be further refined into a clear and reasonable set of objectives.

- Product quality: goals of software quality engineering
- Process quality: means to achieve the goals

- Product qualities
  - Internal qualities: invisible to clients
    - maintainability, flexibility, reparability, changeability
  - External qualities: directly visible to clients
    - Usefulness:
      - usability, performance, security, portability, interoperability
    - Dependability:
      - correctness, reliability, safety, robustness

Dependability Properties

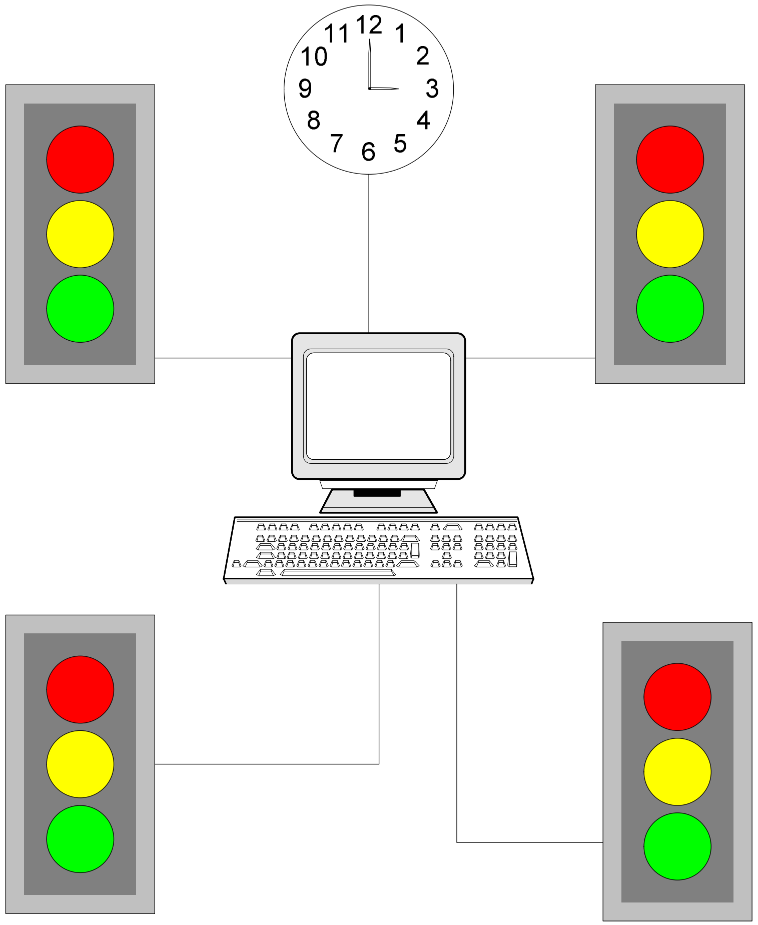
- Correctness
  - A program is correct if it is consistent with its specification.
  - Seldom practical for non-trivial systems
- Reliability
  - Likelihood of correct function for some "unit" of behavior
  - Statistical approximation to correctness (100% reliable = correct)

for Normal Operation

- Safety
  - Concerned with preventing certain undesirable behavior, called hazards
- Robustness
  - Providing acceptable (degraded) behavior under extreme conditions
  - Fail softly

for Abnormal Operation & Situation

An Example of Dependability Property



- **Correctness, Reliability:**
  - Let traffic pass according to correct pattern and central scheduling
- **Robustness, Safety:**
  - Provide degraded function when it fails
  - Never signal conflicting greens
    - Blinking red / blinking yellow is better than no lights.
    - No lights is better than conflicting greens.

Dependable Software Laboratory

Part II. Basic Techniques

Model

- A model is
  - A representation that is simpler than the artifact it represents,
  - But preserves some important attributes of the actual artifact

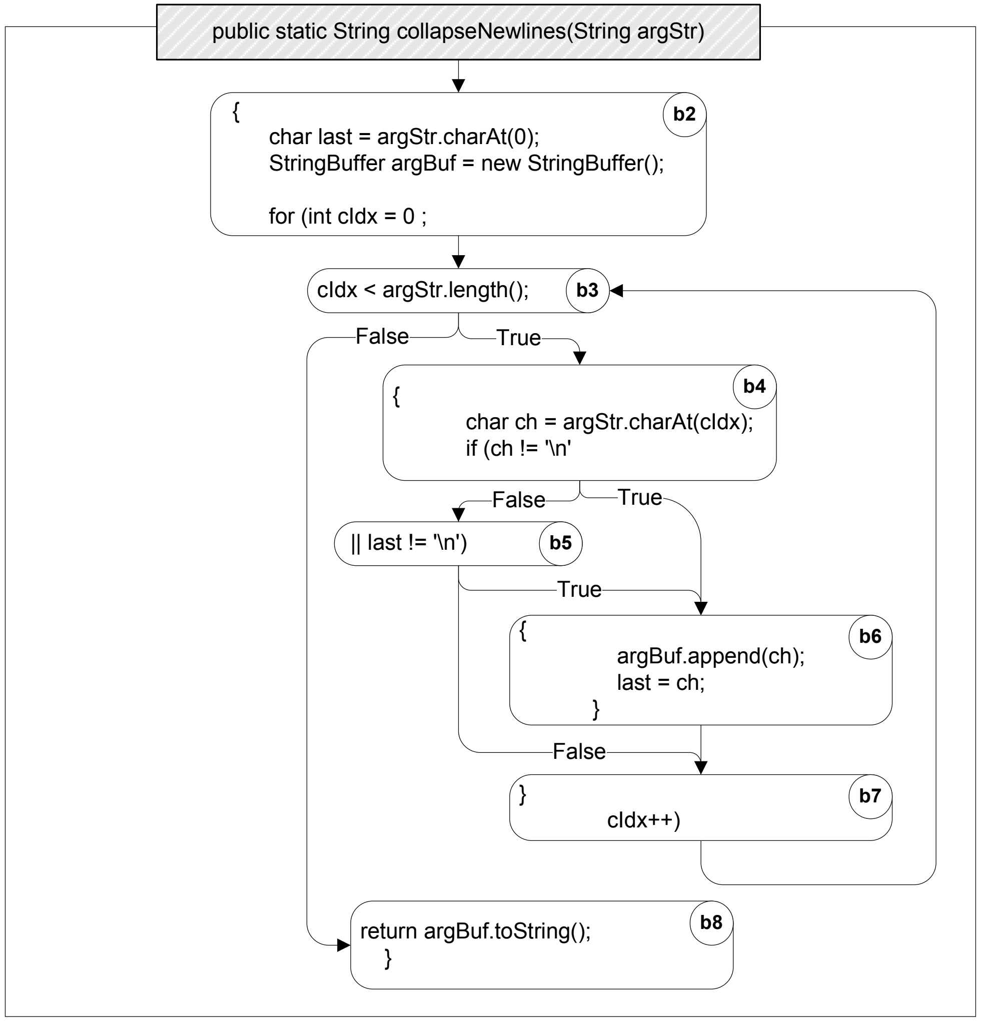
- Our concern is with models of program execution.

Intraprocedural Control Flow Graph

- Called “Control Flow Graph” or “CGF”
  - A directed graph (N, E)
- Nodes
  - Regions of source code (basic blocks)
  - Basic block = maximal program region with a single entry and single exit point
  - Statements are often grouped in single regions to get a compact model.
  - Sometime single statements are broken into more than one node to model control flow within the statement.
- Directed edges
  - Possibility that program execution proceeds from the end of one region directly to the beginning of another

An Example of CFG

```
public static String collapseNewlines(String argStr)
  {
      char last = argStr.charAt(0);
      StringBuffer argBuf = new StringBuffer();

      for (int cIdx = 0 ; cIdx < argStr.length(); cIdx++)
      {
          char ch = argStr.charAt(cIdx);
          if (ch != '\n' || last != '\n')
          {
              argBuf.append(ch);
              last = ch;
          }
      }

      return argBuf.toString();
  }
```



public static String collapseNewlines(String argStr)
{ char last = argStr.charAt(0); StringBuffer argBuf = new StringBuffer(); for (int cIdx = 0 ;
b2
cIdx < argStr.length();
b3
False
True
{ char ch = argStr.charAt(cIdx); if (ch != '\n'
b4
False
True
|| last != '\n')
b5
True
{ argBuf.append(ch); last = ch; }
b6
False
} cIdx++)
b7
return argBuf.toString(); }
b8

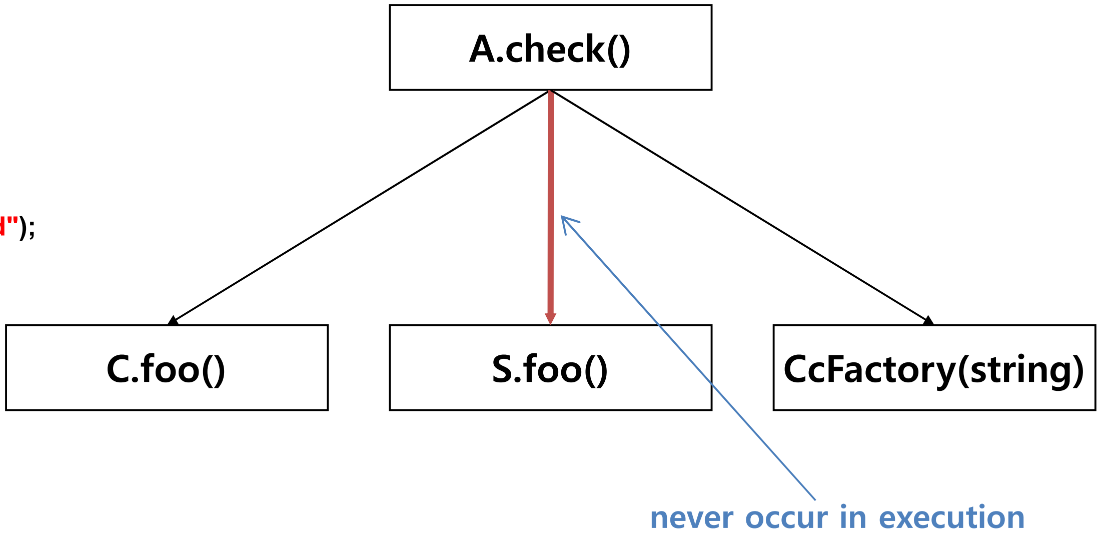
Call Graphs

- "Interprocedural Control Flow Graph"
  - A directed graph (N, E)
- Nodes
  - Represent procedures, methods, functions, etc.
- Edges
  - Represent 'call' relation
- Call graph presents many more design issues and trade-off than CFG.
  - Overestimation of call relation
  - Context sensitive/insensitive

Overestimation in a Call Graph

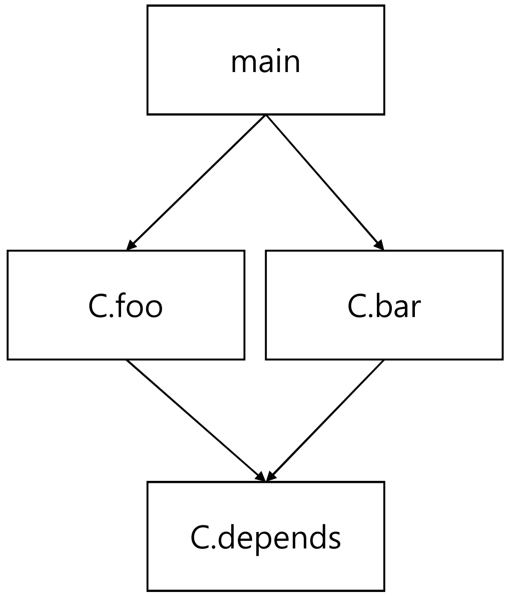
- The static call graph includes calls through dynamic bindings that never occur in execution.

```
public class C {
   public static C cFactory(String kind) {
        if (kind == "C") return new C();
        if (kind == "S") return new S();
        return null;
   }
   void foo() {
        System.out.println("You called the parent's method");
   }
   public static void main(String args[]) {
        (new A()).check();
   }
}
class S extends C {
   void foo() {
        System.out.println("You called the child's method");
   }
}
class A {
   void check() {
        C myC = C.cFactory("S");
        myC.foo();
   }
}
```

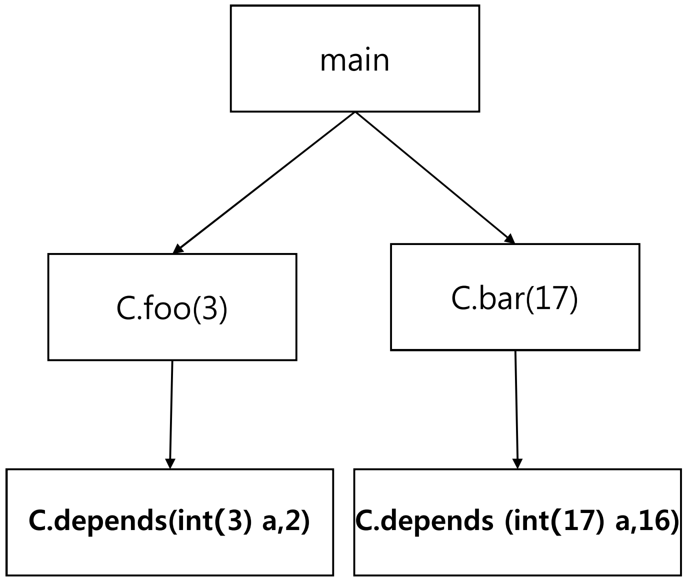


Context Sensitive/Insensitive Call Graphs

```
public class Context {
    public static void main(String args[]) {
        Context c = new Context();
        c.foo(3);
        c.bar(17);
    }

    void foo(int n) {
        int[]  myArray = new int[ n ];
        depends( myArray, 2) ;
    }

    void bar(int n) {
        int[]  myArray = new int[ n ];
        depends( myArray, 16) ;
    }

    void depends( int[] a, int n ) {
        a[n] = 42;
    }
}
```



< Context Insensitive >



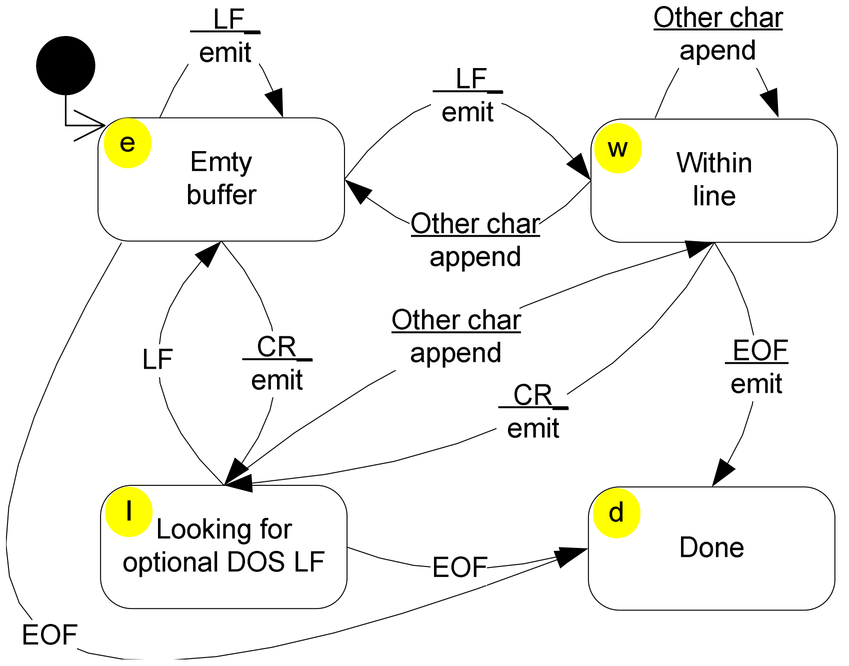
< Context Sensitive >

Finite State Machines

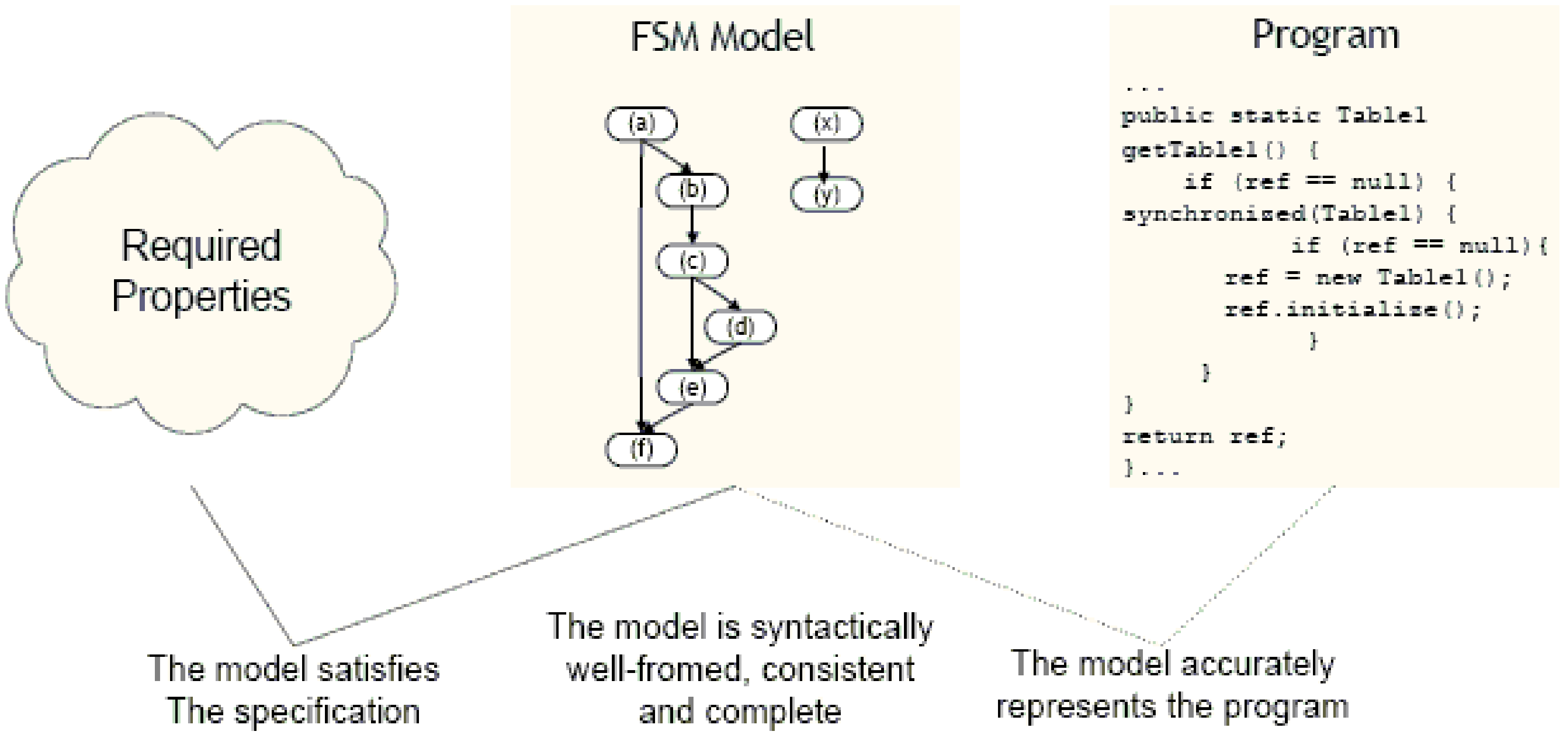
- CFGs can be extracted from programs.
- FSMs are constructed prior to source code, and serve as specifications.
  - A directed graph (N, E)
  - CFG and FSM are duals.

- Nodes
  - A finite set of states
- Edges
  - A set of transitions among states

	LF	CR	EOF	other char
e	e / emit	l / emit	d / -	w / append
w	e / emit	l / emit	d / emit	w / append
l	e / -		d / -	w / append



Correctness Relations for FSM Models

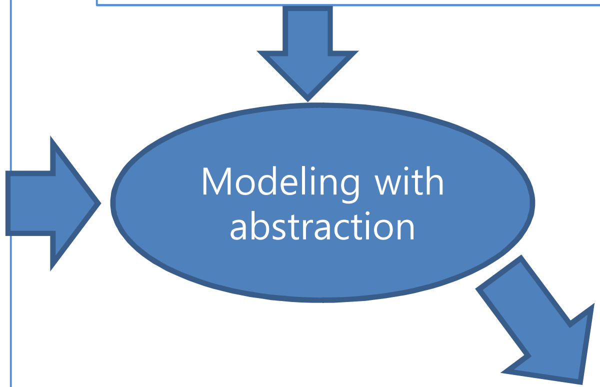

Required Properties
FSM Model
(a)
(b)
(c)
(d)
(e)
(f)
(x)
(y)
Program
The model satisfies The specification
The model is syntactically well-fromed, consistent and complete
The model accurately represents the program

```
...
public static Table1
getTable1() {
     if (ref == null) {
synchronized(Table1) {
              if (ref == null){
           ref = new Table1();
           ref.initialize();
                 }
          }
}
return ref;
}...
```

Abstract Function for Modeling FSMs

```
1   /** Convert each line from standard input */
2   void transduce() {
3
4     #define BUFLEN 1000
5     char buf[BUFLEN];   /* Accumulate line into this buffer   */
6     int   pos = 0;          /* Index for next character in buffer */
7
8     char inChar; /* Next character from input */
9
10    int atCR = 0; /* 0="within line", 1="optional DOS LF" */
11
12    while ((inChar = getchar()) != EOF ) {
13      switch (inChar) {
14      case LF:
15        if (atCR) {    /* Optional DOS LF */
16          atCR = 0;
17        } else {        /* Encountered CR within line */
18          emit(buf, pos);
19          pos = 0;
20        }
21        break;
22      case CR:
23        emit(buf, pos);
24        pos = 0;
25        atCR = 1;
26        break;
27      default:
28        if (pos >= BUFLEN-2) fail("Buffer overflow");
29        buf[pos++] = inChar;
30      } /* switch */
31    }
32    if (pos > 0) {
33      emit(buf, pos);
34    }
35  }
```

Abstract state	*Concrete state*		
	Lines	atCR	pos
e (Empty buffer)	3 – 13	0	0
w (Within line)	13	0	> 0
l (Looking for LF)	13	1	0
d (Done)	36	–	–



	LF	CR	EOF	other
e	e / emit	l / emit	d / –	w / append
w	e / emit	l / emit	d / emit	w / append
l	e / –	l / emit	d / –	w / append

Why Data Flow Models Need?

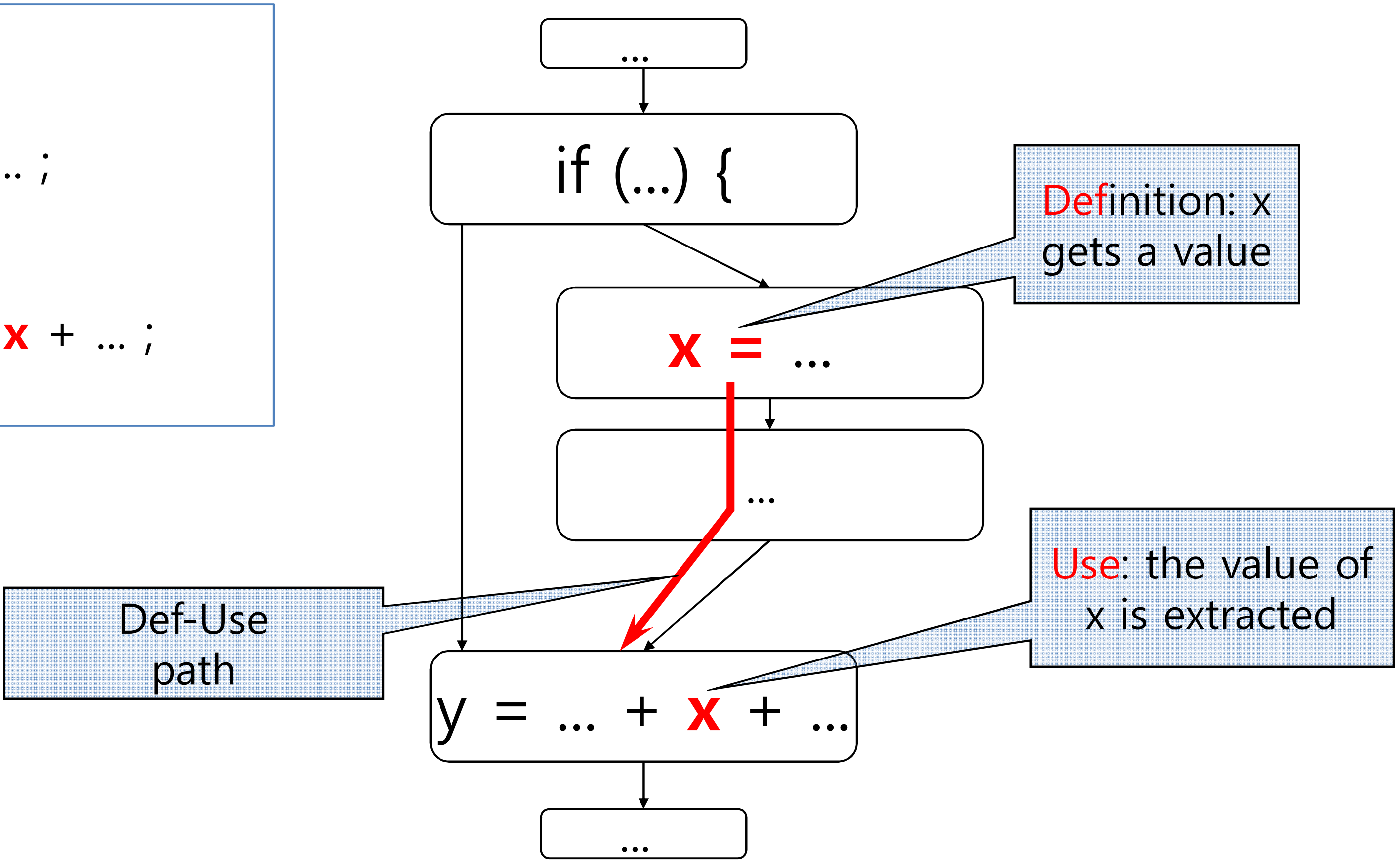
- Models from Chapter 5 emphasized control flow only.
  - Control flow graph, call graph, finite state machine
- We also need to reason about data dependence.
  - To reason about transmission of information through program variables
  - "Where does this value of x come from?"
  - "What would be affected by changing this? "
  - …
- Many program analyses and test design techniques use data flow information and dependences
  - Often in combination with control flow

Definition-Use Pairs

- A def-use (du) pair associates a point in a program where a value is produced with a point where it is used
- Definition: where a variable gets a value
  - Variable declaration
  - Variable initialization
  - Assignment
  - Values received by a parameter
- Use: extraction of a value from a variable
  - Expressions
  - Conditional statements
  - Parameter passing
  - Returns

Def-Use Pairs

```
…
if (…) {
      x = … ;
…
}
y = … + x + … ;
…
```

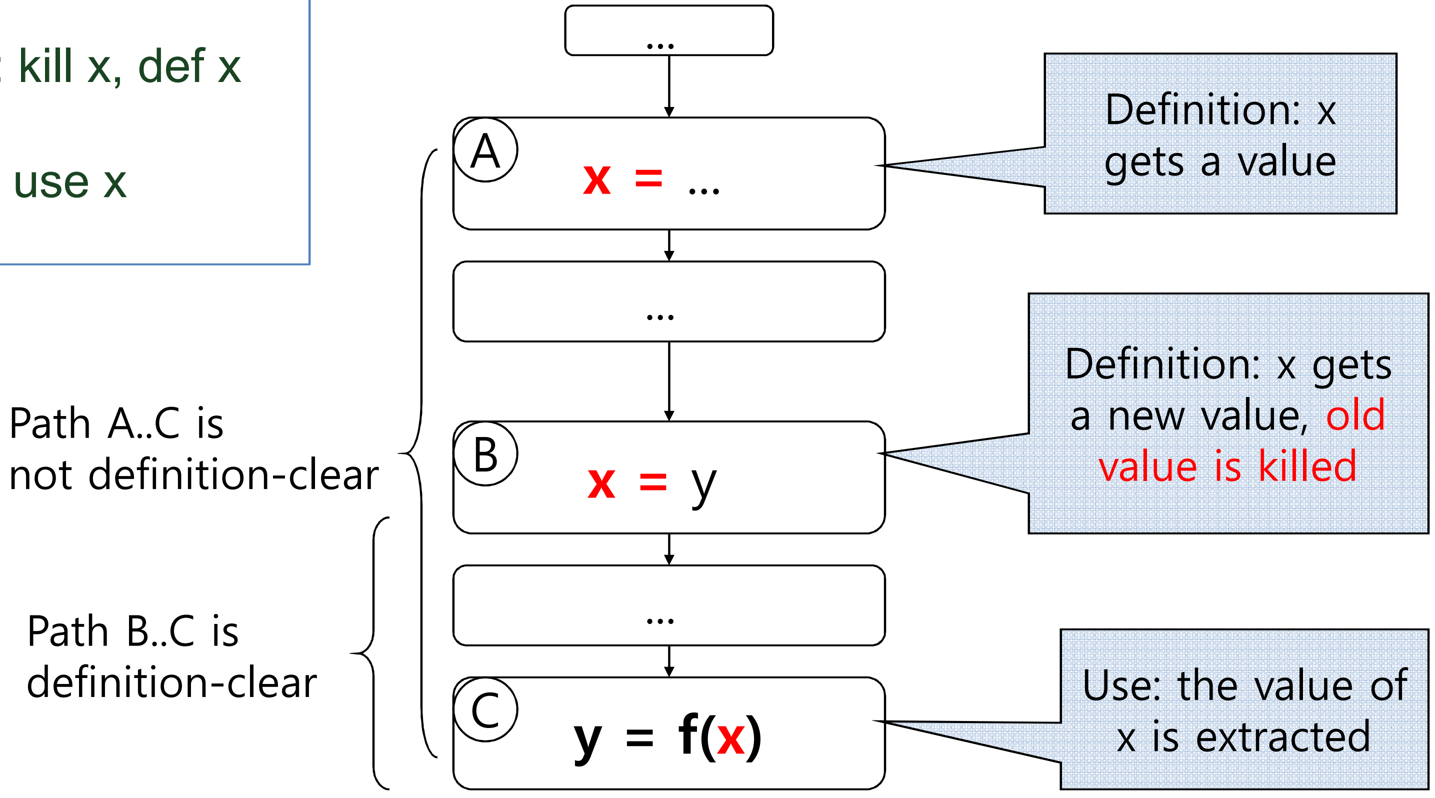


Definition-Clear & Killing

- A definition-clear path is a path along the CFG from a definition to a use of the same variable without another definition of the variable between.
- If, instead, another definition is present on the path, then the latter definition kills the former
- A def-use pair is formed <u>if and only if</u> there is a definition-clear path between the definition and the use

Definition-Clear & Killing

```
x = ...        // A: def x
q = ...
x = y;         //  B: kill x, def x
z = ...
y = f(x);      // C: use x
```

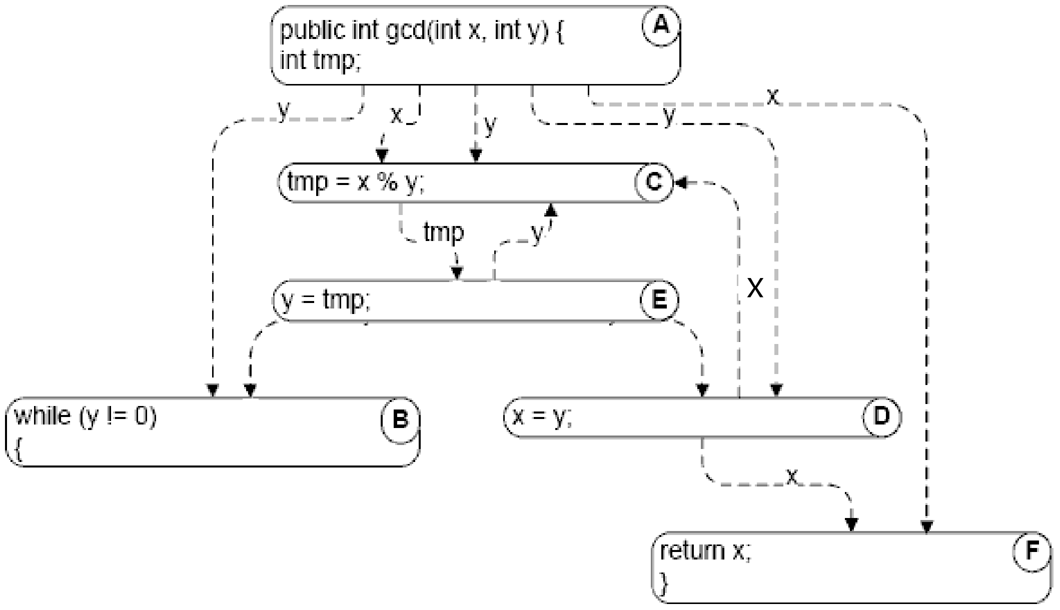


...
A
x = ...
...
B
x = y
...
C
y = f(x)
Definition: x gets a value
Definition: x gets a new value, old value is killed
Use: the value of x is extracted
Path A..C is not definition-clear
Path B..C is definition-clear

(Direct) Data Dependence Graph

- Direct data dependence graph
  - A direct graph (N, E)
    - Nodes: as in the control flow graph (CFG)
    - Edges: def-use (du) pairs, labelled with the variable name

```
/**  Euclid's algorithm */

public int gcd(int x, int y) {
        int tmp;              // A: def x, y, tmp
        while (y != 0) {      // B: use y
            tmp = x % y;      // C: def tmp; use x, y
            x = y;            // D: def x; use y
            y = tmp;          // E: def y; use tmp
        }
    return x;              // F: use x
}
```

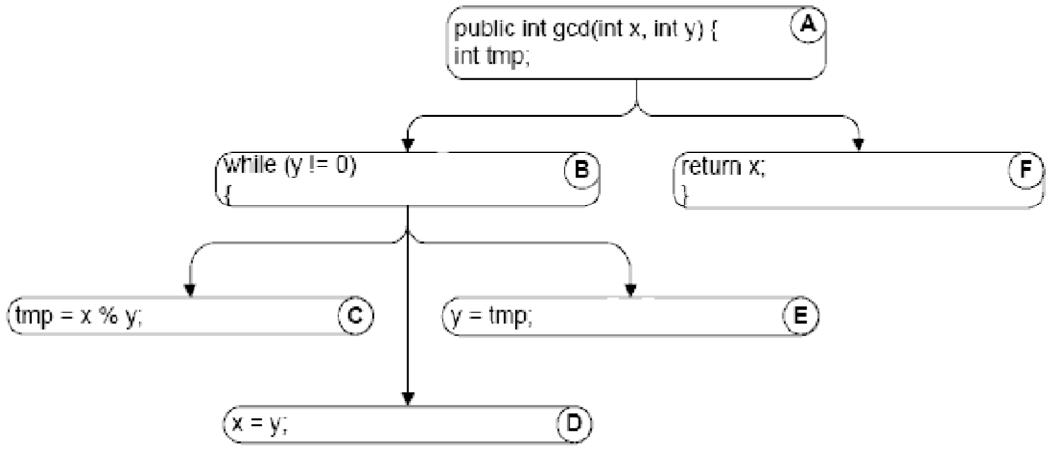


Control Dependence

- Data dependence
  - "Where did these values come from?"

- Control dependence
  - "Which statement controls whether this statement executes?"
  - A directed graph
    - Nodes: as in the CFG
    - Edges: unlabelled, from entry/branching points to controlled blocks

```
/**  Euclid's algorithm */

public int gcd(int x, int y) {
        int tmp;              // A: def x, y, tmp
        while (y != 0) {      // B: use y
            tmp = x % y;      // C: def tmp; use x, y
            x = y;            // D: def x; use y
            y = tmp;          // E: def y; use tmp
        }
    return x;             // F: use x
}
```



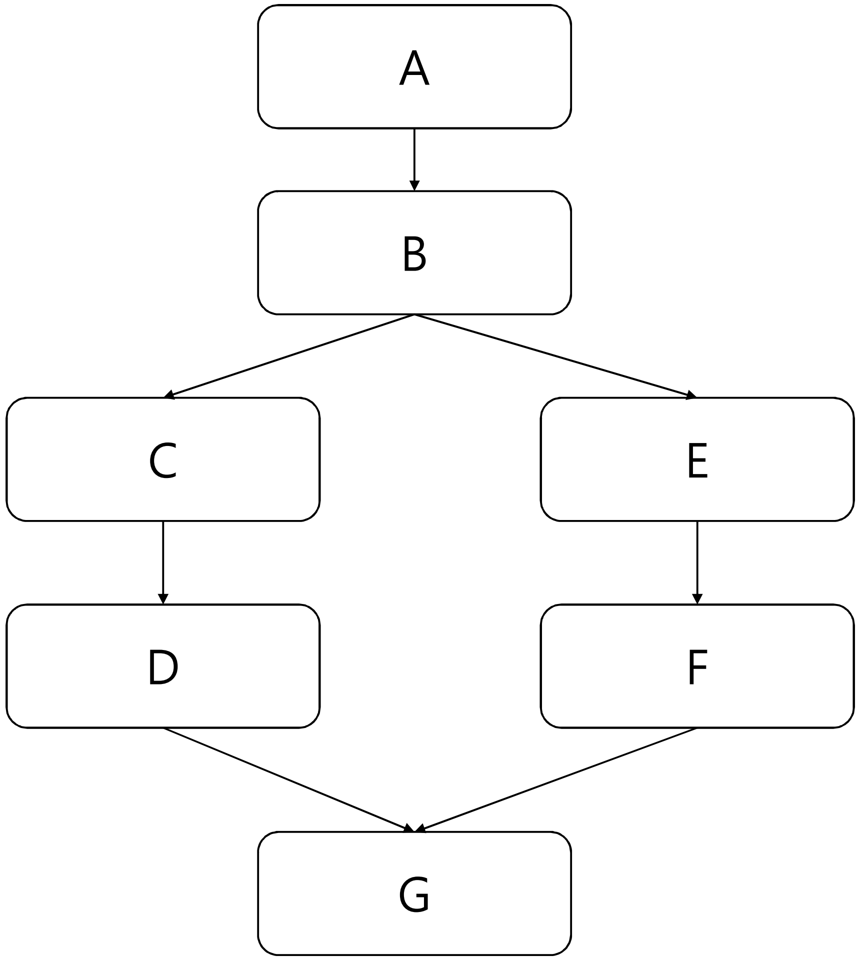
Dominator

- Pre-dominators in a rooted, directed graph can be used to make this intuitive notion of “controlling decision” precise.

- Node M dominates node N, if every path from the root to N passes through M.
  - A node will typically have many dominators, but except for the root, there is a unique immediate dominator of node N which is closest to N on any path from the root, and which is in turn dominated by all the other dominators of N.
  - Because each node (except the root) has a unique immediate dominator, the immediate dominator relation forms a tree.

- Post-dominators are calculated in the reverse of the control flow graph, using a special “exit” node as the root.

An Example of Dominators



- A pre-dominates all nodes.
- G post-dominates all nodes.

- F and G post-dominate E.
- G is the immediate post-dominator of B.

- C does not post-dominate B.

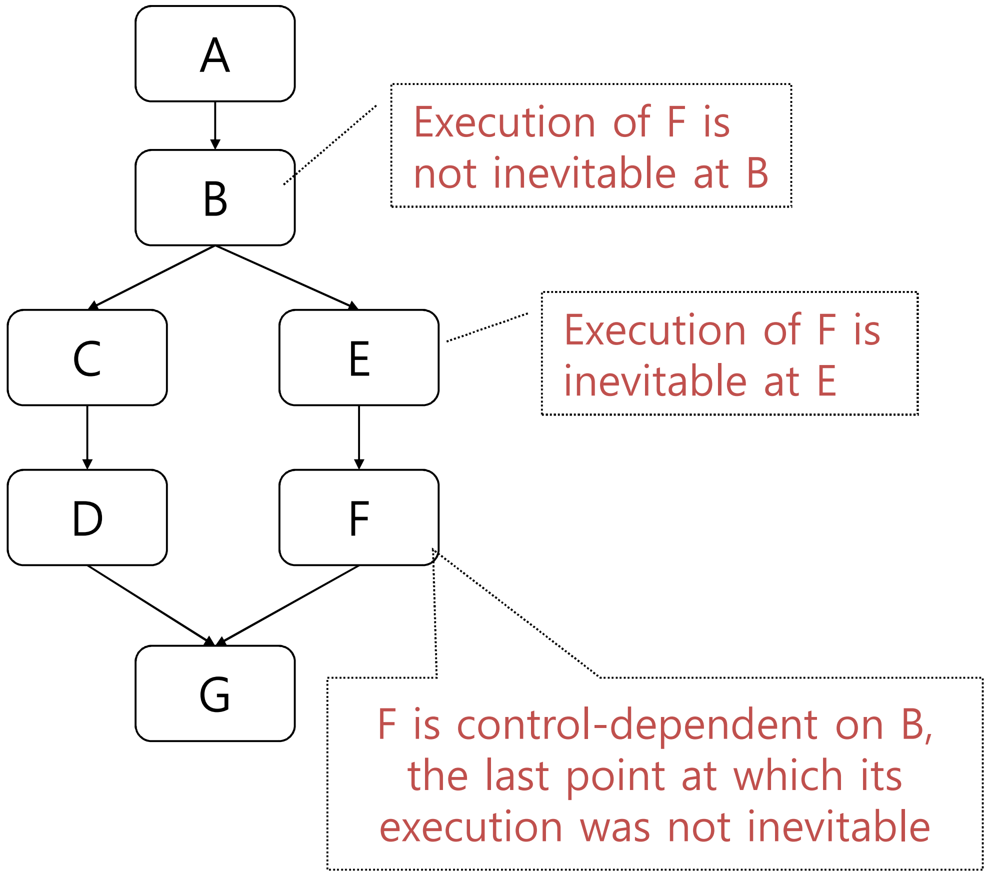
- B is the immediate pre-dominator of G.
- F does not pre-dominate G.

More Precise Definition of Control Dependence

- We can use <u>post-dominators</u> to give a more precise definition of control dependence
  - Consider again a node N that is reached on some but not all execution paths.
  - There must be some node C with the following property:
    - C has at least two successors in the control flow graph (i.e., it represents a control flow decision).
    - C is not post-dominated by N.
    - There is a successor of C in the control flow graph that is post-dominated by N.
  - When these conditions are true, we say node N is control-dependent on node C.

- Intuitively, C is the last decision that controls whether N executes.

An Example of Control Dependence


A
B
C
E
D
F
G
Execution of F is not inevitable at B
Execution of F is inevitable at E
F is control-dependent on B, the last point at which its execution was not inevitable

Symbolic Execution

- Builds predicates that characterize
  - Conditions for executing paths
  - Effects of the execution on program state

- Bridges program behavior to logic

- Finds important applications in
  - Program analysis
  - Test data generation
  - Formal verification (proofs) of program correctness
    - Rigorous proofs of properties of critical subsystems
      - Example: safety kernel of a medical device
    - Formal verification of critical properties particularly resistant to dynamic testing
      - Example: security properties
    - Formal verification of algorithm descriptions and logical designs
      - less complex than implementations

Symbolic State and Interpretation

- Tracing execution with symbolic values and expressions is the basis of symbolic execution.
  - Values are expressions over symbols.
  - Executing statements computes new expressions with the symbols.

Execution with concrete values

(before)

low	12
high	15
mid	-

mid = (high + low) / 2

(after)

low	12
high	15
mid	13

Execution with symbolic values

(before)

low	L
high	H
mid	-

mid = (high + low) / 2

(after)

Low	L
high	H
mid	(L+H) / 2

Tracing Execution with Symbolic Executions

$\wedge\ \forall k, 0 \le k < size : dictKeys[k] = key \rightarrow L \le k \le H$

$\wedge\ H \ge M \ge L$

```
char *binarySearch( char *key, char *dictKeys[ ],
        char *dictValues[ ],  int dictSize) {

  int low = 0;
  int high = dictSize - 1;
  int mid;
  int comparison;

  while (high >= low) {
    mid = (high + low) / 2;
    comparison = strcmp( dictKeys[mid], key );
    if (comparison < 0) {
      low = mid + 1;
    } else if ( comparison > 0 ) {
      high = mid - 1;
    } else {
      return dictValues[mid];
    }
  }
  return 0;

}
```

Execution with symbolic values

```
                (before)
                low = 0
∧               high = (H-1)/2 -1
∧               mid = (H-1)/2

while (high >= low) {

                (after)
                low = 0
∧               high = (H-1)/2 -1
∧               mid = (H-1)/2
∧               (H-1)/2 - 1 >= 0
...
∧               not((H-1)/2 - 1 >= 0)
```

when true

when false

Summary Information

- Symbolic representation of paths may become extremely complex.

- We can simplify the representation by replacing a complex condition *P* with a weaker condition *W* such that

$$P => W$$

  - *W* describes the path with less precision
  - *W* is a *<u>summary</u>* of *P*

An Example of Summary Information

- If we are reasoning about the correctness of the binary search algorithm,
  - In “ *mid = (high+low)/2* ”

Complete condition:

$$\begin{aligned} & low = L \\ \wedge\ & high = H \\ \wedge\ & mid = M \\ \wedge\ & M = (L+H) / 2 \end{aligned}$$

Weaker condition:

$$\begin{aligned} & low = L \\ \wedge\ & high = H \\ \wedge\ & mid = M \\ \wedge\ & L <= M <= H \end{aligned}$$

- The weaker condition contains less information, but still enough to reason about correctness.

Compositional Reasoning

- Follow the hierarchical structure of a program
  - at a small scale (within a single procedure)
  - at larger scales (across multiple procedures)
- Hoare triple: **[*pre*] block [*post*]**
- If the program is in a state satisfying the precondition *pre* at entry to the block, then after execution of the block, it will be in a state satisfying the postcondition *post*

Reasoning about Hoare Triples: Inference

While loops:

I : invariant
C : loop condition
S : body of the loop

premise

$$\frac{[\, I \wedge C \,]\ S\ [\, I \,]}{[\, I \,]\ \mathrm{while}(C)\ \{\ S\ \}\ [I \wedge \neg C]}$$

conclusion

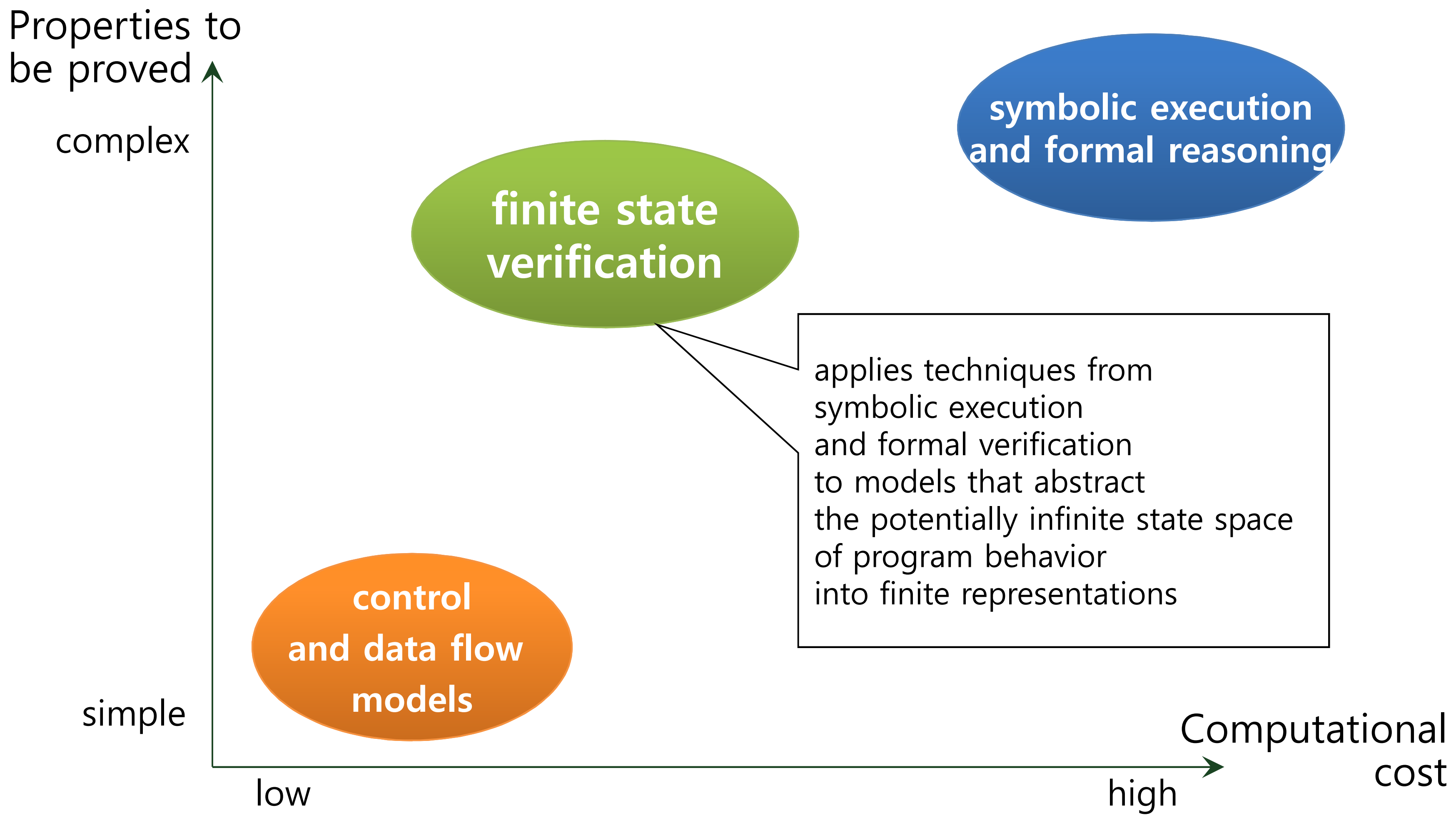
Inference rule says:
if we can verify the premise (top),
then we can infer the conclusion (bottom)

Other Inference Rule

if statement:

$$\frac{[P \wedge C]\ \text{thenpart}\ [Q] \qquad [P \wedge \neg C]\ \text{elsepart}\ [Q]}{[P]\ \text{if}\ (C)\ \{\text{thenpart}\}\ \text{else}\ \{\text{elsepart}\}\ [Q]}$$

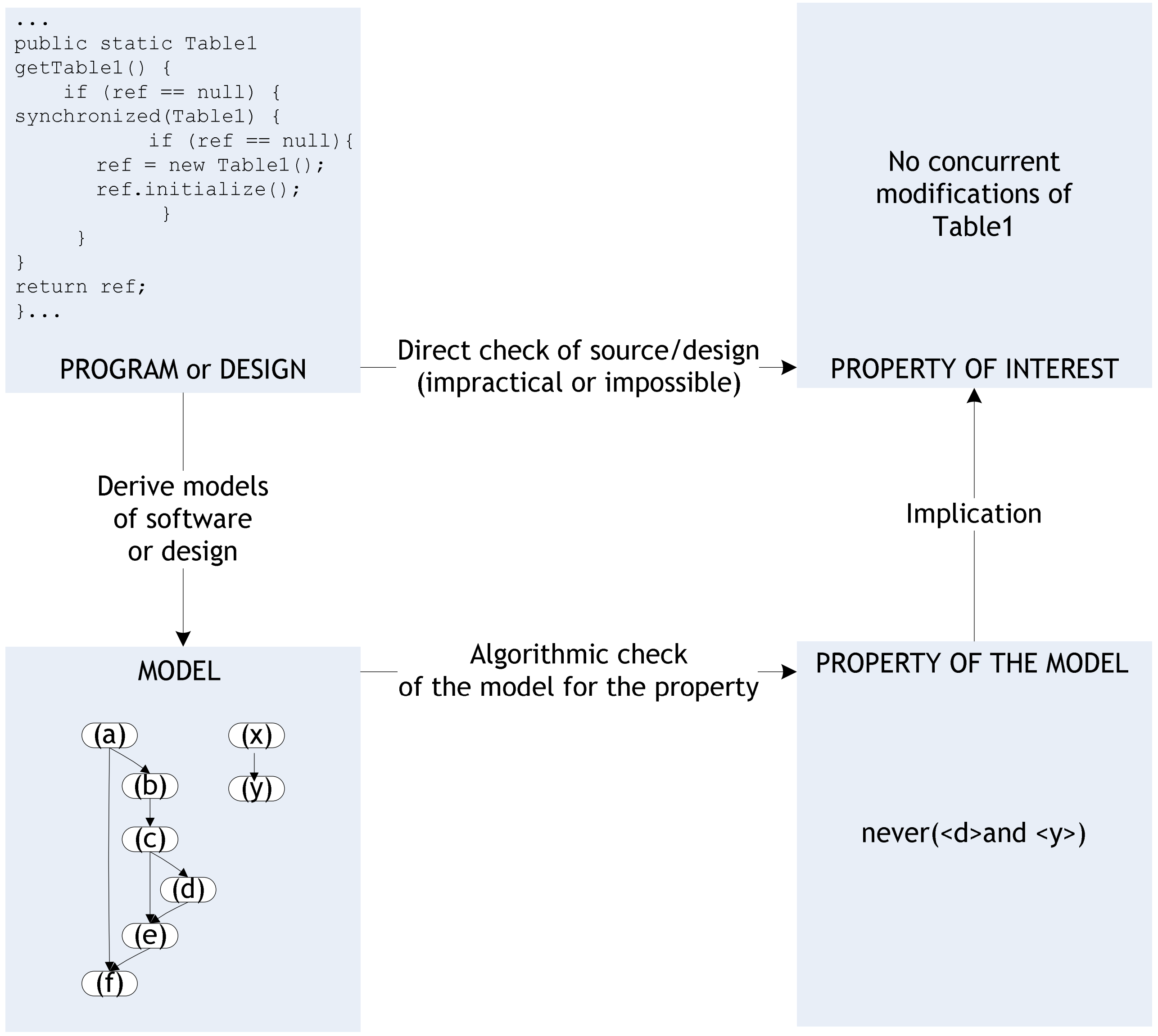
Resources and Results



Properties to be proved
complex
simple
low
high
Computational cost
symbolic execution and formal reasoning
finite state verification
control and data flow models

applies techniques from
symbolic execution
and formal verification
to models that abstract
the potentially infinite state space
of program behavior
into finite representations

Finite State Verification Framework



```
...
public static Table1
getTable1() {
    if (ref == null) {
synchronized(Table1) {
           if (ref == null){
       ref = new Table1();
       ref.initialize();
            }
      }
}
return ref;
}...
```

PROGRAM or DESIGN
Direct check of source/design (impractical or impossible)
No concurrent modifications of Table1
PROPERTY OF INTEREST
Derive models of software or design
Implication
MODEL
Algorithmic check of the model for the property
PROPERTY OF THE MODEL
(a)
(x)
(b)
(y)
(c)
(d)
(e)
(f)
never(<d>and <y>)

The State Space Explosion Problem

- Dining philosophers - looking for deadlock with SPIN

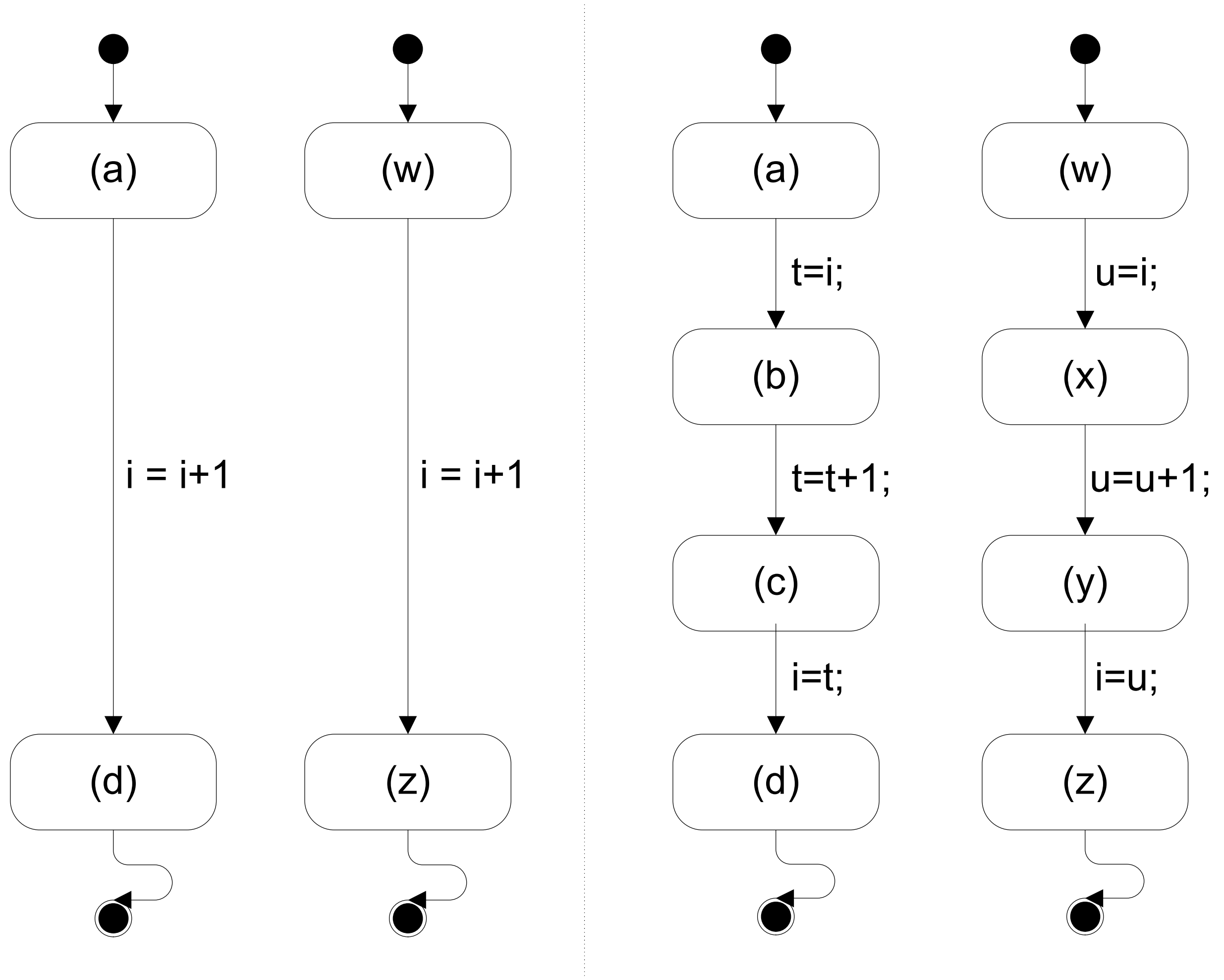
5 phils+forks	145 states
	deadlock found
10 phils+forks	18,313 states
	error trace too long to be useful
15 phils+forks	148,897 states
	error trace too long to be useful

- Team Practice and Homework.

The Model Correspondence Problem

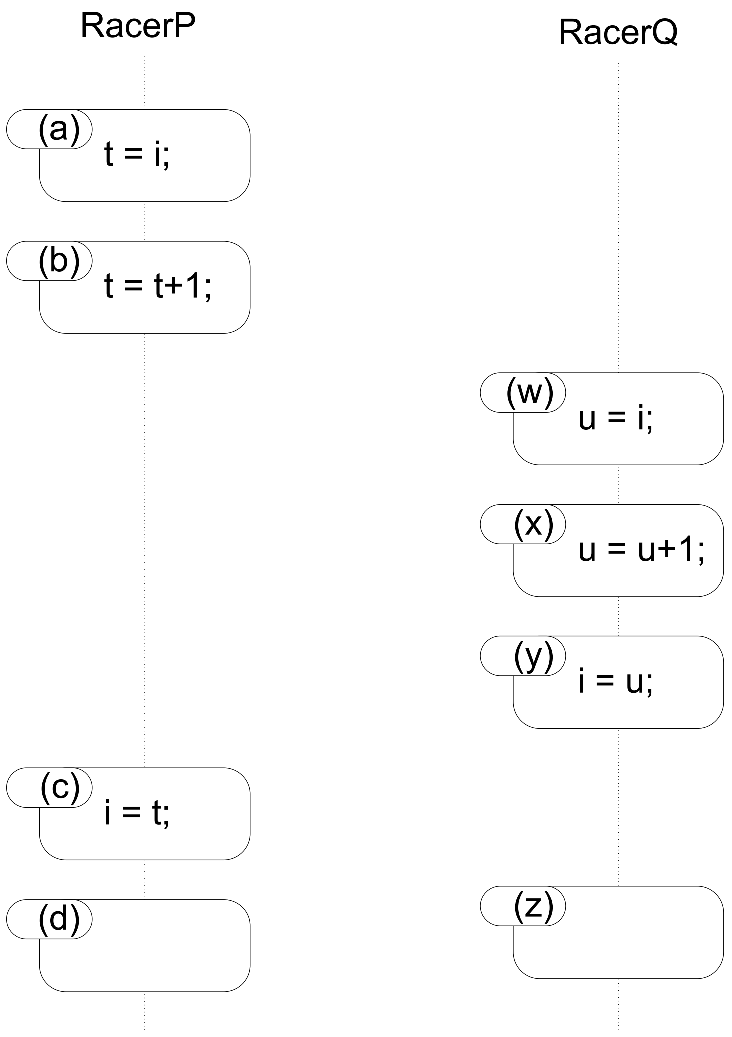
- Verifying correspondence between model and program
  - Extract the model from the source code with verified procedures
    - Blindly mirroring all details → state space explosion
    - Omitting crucial detail → “false alarm” reports
  - Produce the source code automatically from the model
    - Most applicable within well-understood domains
  - Conformance testing
    - Combination of FSV and testing is a good tradeoff

Granularity of Modeling



Analysis of Different Models

- We can find the race only with fine-grain models.



Intentional Models

- Enumerating all reachable states is a limiting factor of finite state verification.
- We can reduce the space by using intentional (symbolic) representations.
  - describe sets of reachable states without enumerating each one individually
- Example (set of Integers)
  - Enumeration {2, 4, 6, 8, 10, 12, 14, 16, 18}
  - Intentional representation: $\{x \in N \mid x \bmod 2 = 0 \text{ and } 0 < x < 20\}$
    $\leftarrow$ "characteristic function"
- Intentional models do not necessarily grow with the size of the set they represent

OBDD: A Useful Intentional Model

- OBDD (Ordered Binary Decision Diagram)
  - A compact representation of Boolean functions

- Characteristic function for transition relations
  - Transitions = pairs of states
  - Function from pairs of states to Booleans is true, if there is a transition between the pair.
  - Built iteratively by breadth-first expansion of the state space:
    - Create a representation of the whole set of states reachable in k+1 steps from the set of states reachable in k steps
    - OBDD stabilizes when all the transitions that can occur in the next step are already represented in the OBDD.

From OBDD to Symbolic Checking

- Intentional representation itself is not enough.
- We must have an algorithm for determining whether it satisfies the property we are checking.

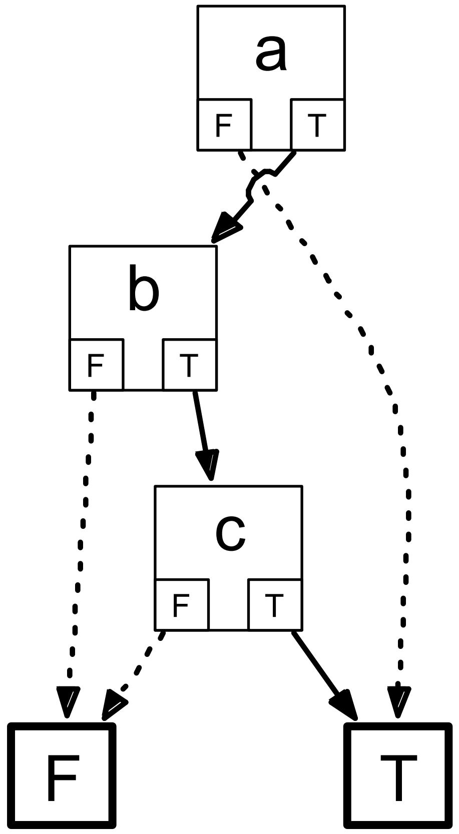
- Example: A set of communicating state machines using OBDD
  - To represent the transition relation of a set of communicating state machines
  - To model a class of temporal logic specification formulas

- Combine OBDD representations of model and specification to produce a representation of just the set of transitions leading to a violation of the specification
  - If the set is empty, the property has been verified.

Representing Transition Relations as Boolean Functions

- a ⇒ b and c
  not(a) or (b and c)

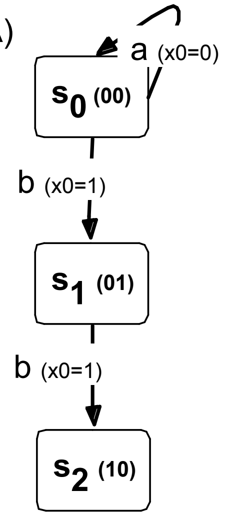
- BDD is a decision tree that has been transformed into an acyclic graph by merging nodes leading to identical sub-trees.



Representing Transition Relations as Boolean Functions : Steps

A. Assign a label to each state
B. Encode transitions
C. The transition tuples correspond to paths leading to true, and all other paths lead to false.

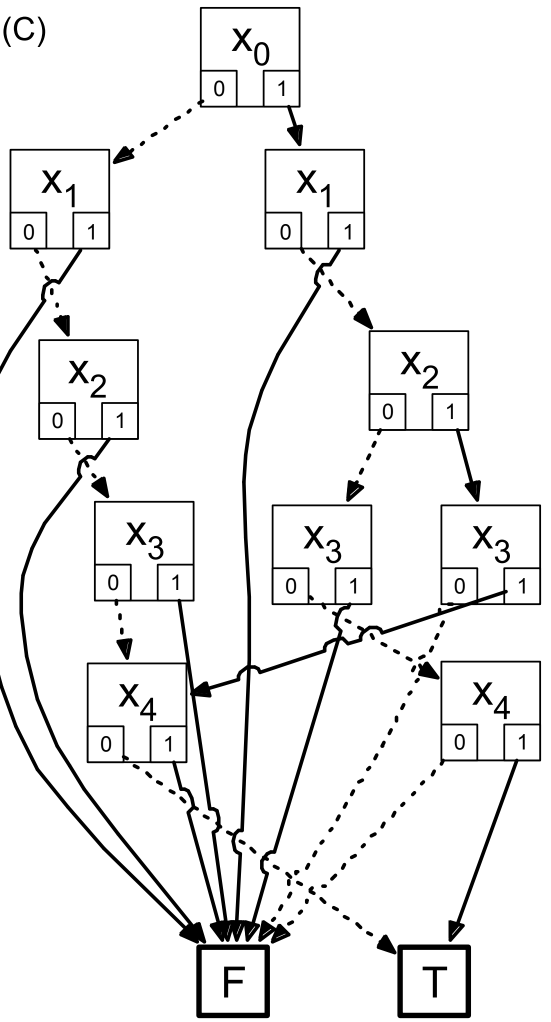
(A)



(B)

x_0	x_1 x_2	x_3 x_4
0	0 0	0 0
1	0 0	0 1
1	0 1	1 0
sym	from state	to state

(C)



Intentional vs. Explicit Representations

- Worst case:
  - Given a large set S of states,
  - a representation capable of distinguishing each subset of S cannot be more compact on average than the representation that simply lists elements of the chosen subset.

- Intentional representations work well when they exploit structure and regularity of the state space.

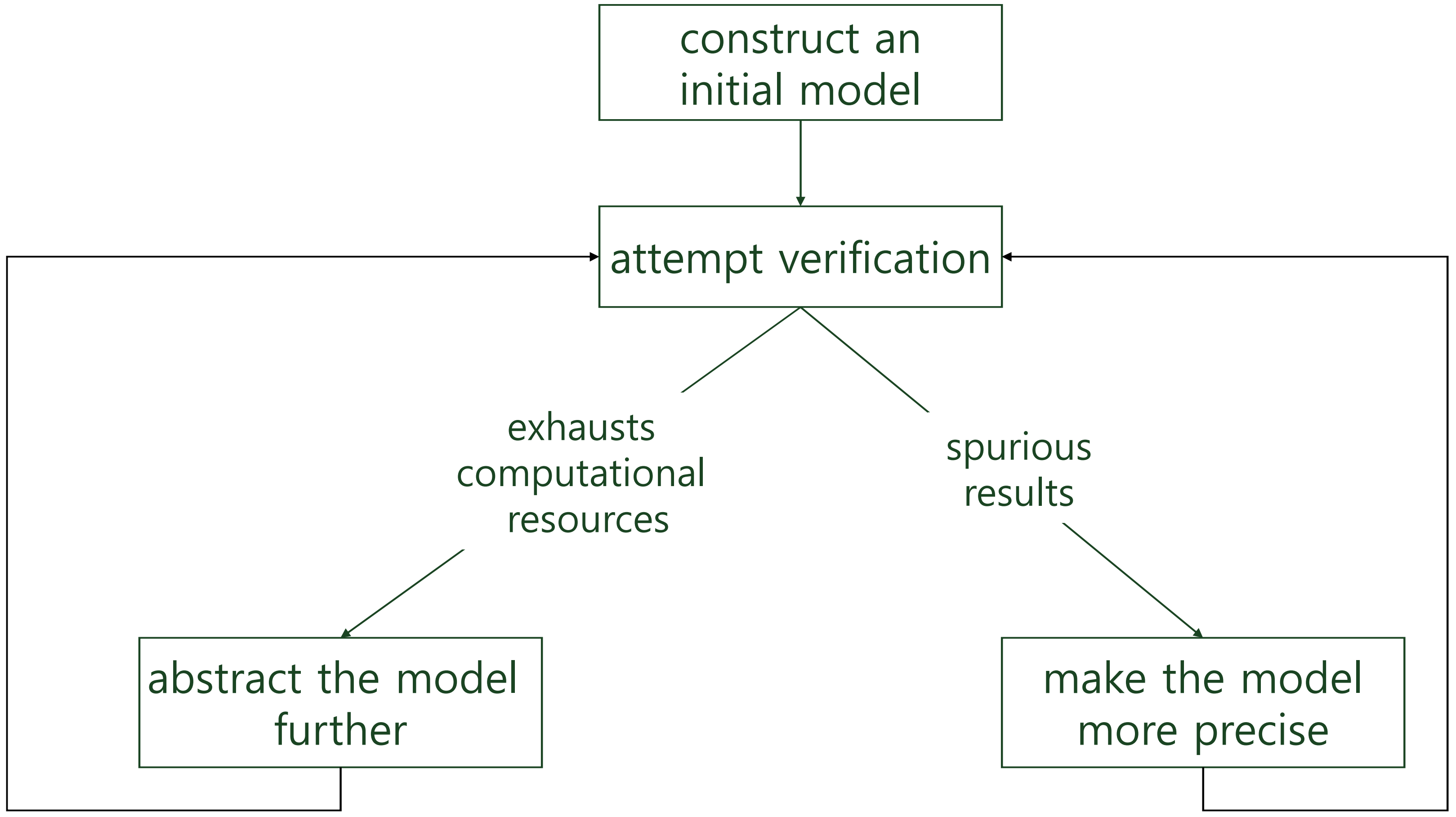
Model Refinement

- Construction of finite state models
  - Should balance precision and efficiency
- Often the first model is unsatisfactory
  - Report potential failures that are obviously impossible
  - Exhaust resources before producing any result

- Minor differences in the model can have large effects on tractability of the verification procedure.

- Finite state verification as iterative process is required.

Iteration Process


construct an initial model
attempt verification
exhausts computational resources
spurious results
abstract the model further
make the model more precise

Refinement 1: Adding Details to the Model

$M_1 \models P$	Initial (coarse grain) model (The counter example that violates P is possible in M_1, but does not correspond to an execution of the real program.)
$M_2 \models P$	Refined (more detailed) model (the counterexample above is not possible in M_2 , but a new counterexamples violates M_2 , and does not correspond to an execution of the real program too.)
....	
$M_k \models P$	Refined (final) model (the counter example that violates P in M_k corresponds to an execution in the real program.)

Refinement 2: Add Premises to the Property

Initial (coarse grain) model

$M \models P$

Add a constraint C_1 that eliminates the bogus behavior

$M \models C_1 \Rightarrow P$

$M \models (C_1 \text{ and } C_2) \Rightarrow P$

....

Until the verification succeeds or produces a valid counter example

Part III. Problems and Methods

Terminology in Testing

Terms	Descriptions
Test case	a set of inputs, execution conditions, and a pass/fail criterion
Test case specification (Test specification)	a requirement to be satisfied by one or more test cases
Test obligation	a partial test case specification, requiring some property deemed important to thorough testing
Test suite	a set of test cases
Test (**Test execution)**	the activity of executing test cases and evaluating their results
Adequacy criterion	a predicate that is true (satisfied) or false of a ⟨program, test suite⟩ pair

Source of Test Specification

Testing	Other names	Source of test specification / Example
Functional Testing	Black box testing Specification-based testing	Software specification
		If specification requires robust recovery from power failure, test obligations should include simulated power failure.
Structural Testing	White box testing	Source code
		Traverse each program loop one or more times
Model-based Testing		Models of system • Models used in specification or design • Models derived from source code
		Exercise all transitions in communication protocol model
Fault-based Testing		Hypothesized faults, common bugs
		Check for buffer overflow handling (common vulnerability) by testing on very large inputs

Adequacy Criteria

- Adequacy criterion = Set of test obligations

- A test suite satisfies an adequacy criterion, iff
  - All the tests succeed (pass), and
  - Every test obligation in the criterion is satisfied by at least one of the test cases in the test suite.

  - Example:
    - "The statement coverage adequacy criterion is satisfied by test suite S for program P, if each executable statement in P is executed by at least one test case in S, and the outcome of each test execution was pass."

Coverage

- Measuring **coverage** (% of satisfied test obligations) can be a useful indicator of
  - Progress toward a thorough test suite (thoroughness of test suite)
  - Trouble spots requiring more attention in testing

- But, coverage is only a proxy for thoroughness or adequacy.
  - It's easy to improve coverage without improving a test suite (much easier than designing good test cases)
  - The only measure that really matters is (cost-) effectiveness.

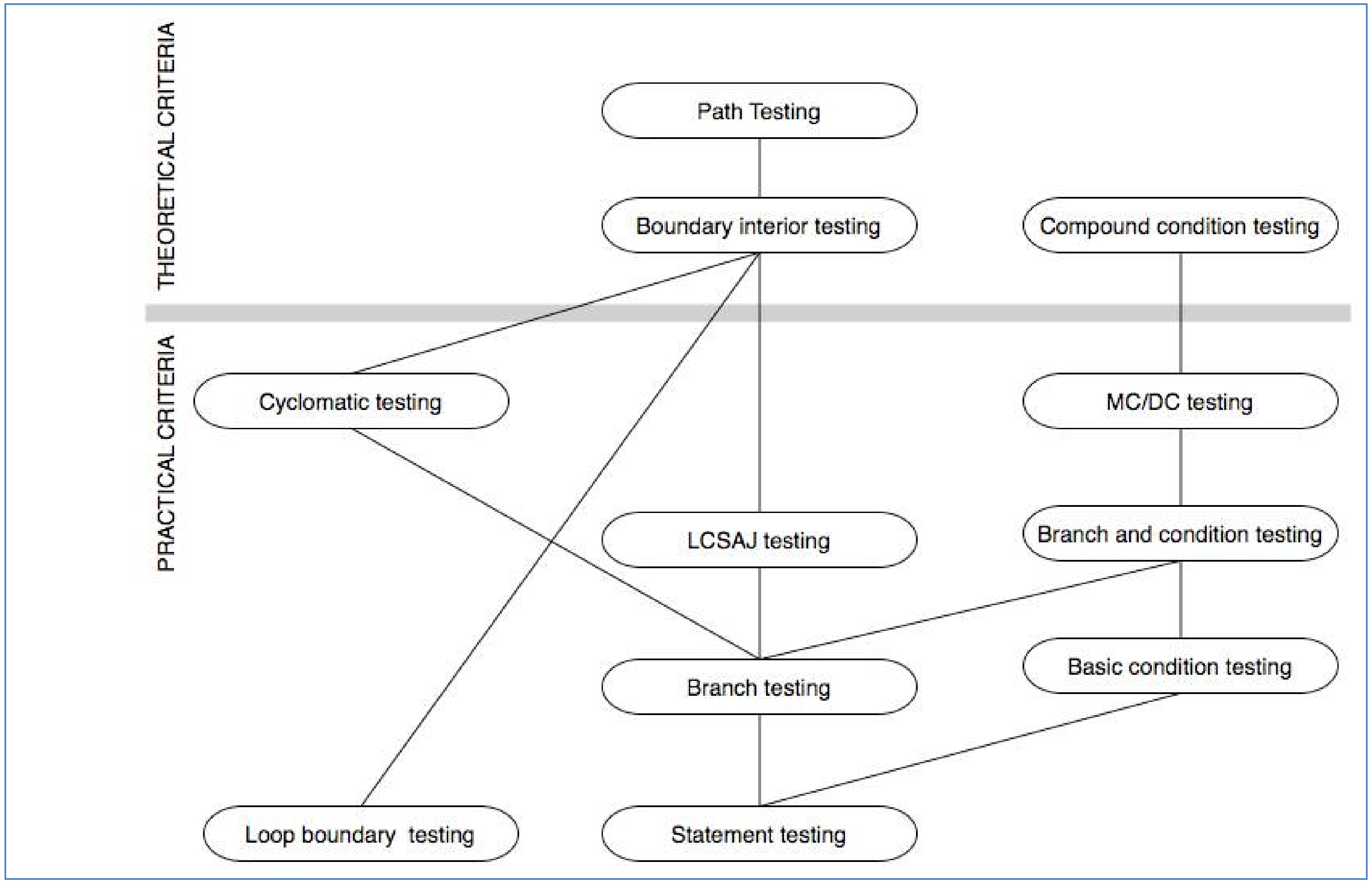
Comparing Criteria

- Can we distinguish stronger from weaker adequacy criteria?

- Analytical approach
  - Describe conditions under which one adequacy criterion is provably stronger than another
  - Just a piece of the overall "effectiveness" question
  - Stronger = gives stronger guarantees
    → **Subsumes relation**

Subsumes Relation

- Test adequacy criterion *A* **subsumes** test adequacy criterion *B* iff, for every program *P*, every test suite satisfying *A* with respect to *P* also satisfies *B* with respect to *P*.
  - E.g. Exercising all program branches (branch coverage) subsumes exercising all program statements.
- A common analytical comparison of closely related criteria
  - Useful for working from easier to harder levels of coverage, but not a direct indication of quality


THEORETICAL CRITERIA
PRACTICAL CRITERIA
Path Testing
Boundary interior testing
Compound condition testing
Cyclomatic testing
MC/DC testing
LCSAJ testing
Branch and condition testing
Basic condition testing
Branch testing
Loop boundary testing
Statement testing

Functional Testing

- **Functional testing**
  - Deriving test cases from program specifications
  - 'Functional' refers to the source of information used in test case design, not to what is tested.

- Also known as:
  - Specification-based testing (from specifications)
  - Black-box testing (no view of source code)

- Functional specification = description of intended program behavior
  - Formal or informal

Systematic testing vs. Random testing

- **Random (uniform) testing**
  - Pick possible inputs uniformly
  - Avoids designer's bias
  - But, treats all inputs as equally valuable

- **Systematic (non-uniform) testing**
  - Try to select inputs that are especially valuable
  - Usually by choosing representatives of classes that are apt to fail often or not at all

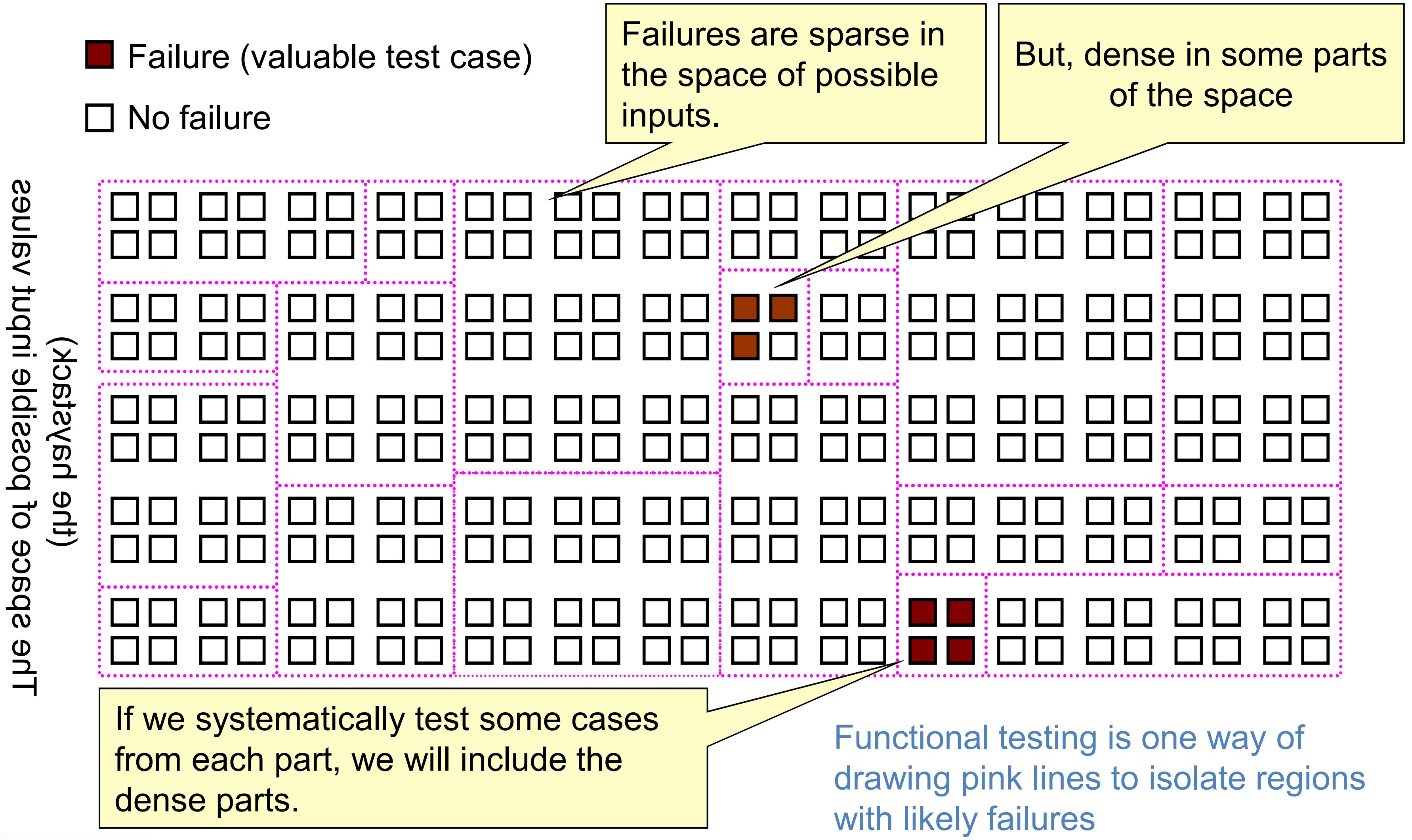
- Functional testing is a systematic (partition-based) testing strategy.

Purpose of Testing

- Our goal is to find needles and remove them from hay.
  → Look systematically (non-uniformly) for needles !!!
  - We need to use everything we know about needles.
    - E.g. Are they heavier than hay? Do they sift to the bottom?

- To estimate the proportion of needles to hay
  → Sample randomly !!!
  - Reliability estimation requires unbiased samples for valid statistics.
  - But that's not our goal.

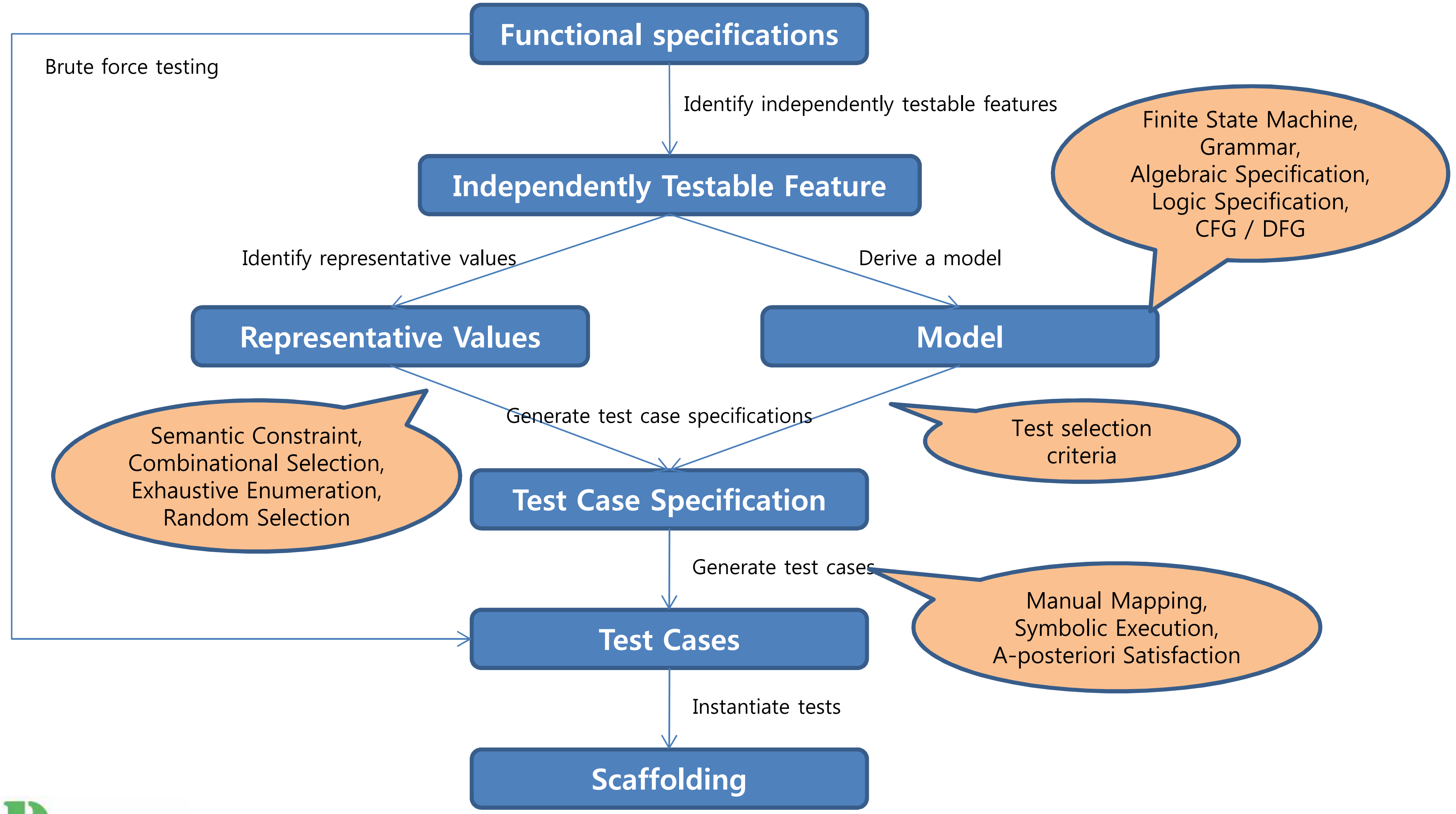
Systematic Partition Testing



■ Failure (valuable test case)
□ No failure
Failures are sparse in the space of possible inputs.
But, dense in some parts of the space
The space of possible input values (the haystack)
If we systematically test some cases from each part, we will include the dense parts.

Functional testing is one way of drawing pink lines to isolate regions with likely failures

Main Steps of Functional Program Testing


Functional specifications
Brute force testing
Identify independently testable features
Finite State Machine, Grammar, Algebraic Specification, Logic Specification, CFG / DFG
Independently Testable Feature
Identify representative values
Derive a model
Representative Values
Model
Generate test case specifications
Semantic Constraint, Combinational Selection, Exhaustive Enumeration, Random Selection
Test selection criteria
Test Case Specification
Generate test cases
Manual Mapping, Symbolic Execution, A-posteriori Satisfaction
Test Cases
Instantiate tests
Scaffolding

Key Ideas in Combinatorial Approaches

1. Category-partition testing
   - Separate (manual) identification of values that characterize the input space from (automatic) generation of combinations for test cases

2. Pairwise testing
   - Systematically test interactions among attributes of the program input space with a relatively small number of test cases

3. Catalog-based testing
   - Aggregate and synthesize the experience of test designers in a particular organization or application domain, to aid in identifying attribute values

1. Category-Partition Testing

1. Decompose the specification into independently testable features
   - for each feature, identify parameters and environment elements
   - for each parameter and environment element, identify elementary characteristics (→ categories)

2. Identify representative values
   - for each characteristic(category), identify classes of values
     - normal values
     - boundary values
     - special values
     - error values

3. Generate test case specifications

Identify Independently Testable Units

Model	Model number
	Number of required slots for selected model (#SMRS)
	Number of optional slots for selected model (#SMOS)

Components	Correspondence of selection with model slots
	Number of required components with selection ≠ empty
	Required component selection
	Number of optional components with selection ≠ empty
	Optional component selection

Product Database	Number of models in database (#DBM)
	Number of components in database (#DBC)

Step 2: Identify Representative Values

- Identify representative classes of values for each of the categories
- Representative values may be identified by applying
  - Boundary value testing
    - Select extreme values within a class
    - Select values outside but as close as possible to the class
    - Select interior (non-extreme) values of the class
  - Erroneous condition testing
    - Select values outside the normal domain of the program

Representative Values: Model

- Model number
  - Malformed
  - Not in database
  - Valid
- Number of required slots for selected model (#SMRS)
  - 0
  - 1
  - Many
- Number of optional slots for selected model (#SMOS)
  - 0
  - 1
  - Many

Step 3: Generate Test Case Specifications

- A combination of values for each category corresponds to a test case specification.
  - In the example, we have 314,928 test cases.
  - Most of which are impossible.
  - Example: zero slots and at least one incompatible slot

- Need to introduce constraints in order to
  - Rule out impossible combinations, and
  - Reduce the size of the test suite, if too large

  - Example:
    - Error constraints
    - Property constraints
    - Single constraints

Error Constraints

- [error] indicates a value class that corresponds to an erroneous values.
  - Need to be tried only once
- Error value class
  - No need to test all possible combinations of errors, and one test is enough.

Model number	
Malformed	[error]
Not in database	[error]
Valid	
Correspondence of selection with model slots	
Omitted slots	[error]
Extra slots	[error]
Mismatched slots	[error]
Complete correspondence	
Number of required comp. with non empty selection	
0	[error]
< number of required slots	[error]
Required comp. selection	
≥ 1 not in database	[error]
Number of models in database (#DBM)	
0	[error]
Number of components in database (#DBC)	
0	[error]

Error constraints reduce test suite from 314,928 to 2,711 test cases

Property Constraints

Number of required slots for selected model (#SMRS)

1	[property RSNE]
Many	[property RSNE] [property RSMANY]

Number of optional slots for selected model (#SMOS)

1	[property OSNE]
Many	[property OSNE] [property OSMANY]

Number of required comp. with non empty selection

0	[if RSNE] [error]
< number required slots	[if RSNE] [error]
= number required slots	[if RSMANY]

Number of optional comp. with non empty selection

< number required slots	[if OSNE]
= number required slots	[if OSMANY]

from 2,711 to 908 test cases

Single Constraints

Number of required slots for selected model (#SMRS)

0	[single]
1	[property RSNE] [single]

Number of optional slots for selected model (#SMOS)

0	[single]
1	[single] [property OSNE]

Required component selection

Some default	[single]

Optional component selection

Some default	[single]

Number of models in database (#DBM)

1	[single]

Number of components in database (#DBC)

1	[single]

from 908 to 69 test cases

Check Configuration – Summary of Categories

Parameter Model

- Model number
  - Malformed [error]
  - Not in database [error]
  - Valid
- Number of required slots for selected model (#SMRS)
  - 0 [single]
  - 1 [property RSNE] [single]
  - Many [property RSNE] [property RSMANY]
- Number of optional slots for selected model (#SMOS)
  - 0 [single]
  - 1 [property OSNE] [single]
  - Many [property OSNE] [property OSMANY]

Environment Product data base

- Number of models in database (#DBM)
  - 0 [error]
  - 1 [single]
  - Many
- Number of components in database (#DBC)
  - 0 [error]
  - 1 [single]
  - Many

Parameter Component

- Correspondence of selection with model slots
  - Omitted slots [error]
  - Extra slots [error]
  - Mismatched slots [error]
  - Complete correspondence
- # of required components (selection ≠ empty)
  - 0 [if RSNE] [error]
  - < number required slots [if RSNE] [error]
  - = number required slots [if RSMANY]
- Required component selection
  - Some defaults [single]
  - All valid
  - ≥ 1 incompatible with slots
  - ≥ 1 incompatible with another selection
  - ≥ 1 incompatible with model
  - ≥ 1 not in database [error]
- # of optional components (selection ≠ empty)
  - 0
  - < #SMOS [if OSNE]
  - = #SMOS [if OSMANY]
- Optional component selection
  - Some defaults [single]
  - All valid
  - ≥ 1 incompatible with slots
  - ≥ 1 incompatible with another selection
  - ≥ 1 incompatible with model
  - ≥ 1 not in database [error]

2. Pairwise Combination Testing

- Category partition works well when intuitive constraints reduce the number of combinations to a small amount of test cases.
  - Without many constraints, the number of combinations may be unmanageable.

- Pairwise combination
  - Instead of exhaustive combinations
  - Generate combinations that efficiently cover all pairs (triples,...) of classes
  - Rationale:
    - Most failures are triggered by single values or combinations of a few values.
    - Covering pairs (triples,...) reduces the number of test cases, but reveals most faults.

An Example: Display Control

- No constraints reduce the total number of combinations 432 (3x4x3x4x3) test cases, if we consider all combinations.

Display Mode	Language	Fonts	Color	Screen size
full-graphics	English	Minimal	Monochrome	Hand-held
text-only	French	Standard	Color-map	Laptop
limited-bandwidth	Spanish	Document-loaded	16-bit	Full-size
	Portuguese		True-color	

Pairwise Combination: 17 Test Cases

Language	Color	Display Mode	Fonts	Screen Size
English	Monochrome	Full-graphics	Minimal	Hand-held
English	Color-map	Text-only	Standard	Full-size
English	16-bit	Limited-bandwidth	-	Full-size
English	True-color	Text-only	Document-loaded	Laptop
French	Monochrome	Limited-bandwidth	Standard	Laptop
French	Color-map	Full-graphics	Document-loaded	Full-size
French	16-bit	Text-only	Minimal	-
French	True-color	-	-	Hand-held
Spanish	Monochrome	-	Document-loaded	Full-size
Spanish	Color-map	Limited-bandwidth	Minimal	Hand-held
Spanish	16-bit	Full-graphics	Standard	Laptop
Spanish	True-color	Text-only	-	Hand-held
Portuguese	-	-	Monochrome	Text-only
Portuguese	Color-map	-	Minimal	Laptop
Portuguese	16-bit	Limited-bandwidth	Document-loaded	Hand-held
Portuguese	True-color	Full-graphics	Minimal	Full-size
Portuguese	True-color	Limited-bandwidth	Standard	Hand-held

Adding Constraints

- Simple constraints
  - Example: "Color monochrome not compatible with screen laptop and full size" can be handled by considering the case in separate tables.

Display Mode	Language	Fonts	Color	Screen size
full-graphics	English	Minimal	Monochrome	Hand-held
text-only	French	Standard	Color-map	
limited-bandwidth	Spanish	Document-loaded	16-bit	
	Portuguese		True-color	

Display Mode	Language	Fonts	Color	Screen size
full-graphics	English	Minimal		
text-only	French	Standard	Color-map	Laptop
limited-bandwidth	Spanish	Document-loaded	16-bit	Full-size
	Portuguese		True-color	

Structural Testing

- Judging test suite thoroughness based on the structure of the program itself
  - Also known as
    - White-box testing
    - Glass-box testing
    - Code-based testing
  - Distinguish from functional (requirements-based, “black-box”) testing

- Structural testing is still testing product functionality against its specification.
  - Only the measure of thoroughness has changed.

Rationale of Structural Testing

- One way of answering the question "What is missing in our test suite?"
  - If a part of a program is not executed by any test case in the suite, faults in that part cannot be exposed.
  - But what's the 'part'?
    - Typically, a control flow element or combination
    - Statements (or CFG nodes), Branches (or CFG edges)
    - Fragments and combinations: Conditions, paths
- Structural testing complements functional testing.
  - Another way to recognize cases that are treated differently
- Recalling fundamental rationale
  - Prefer test cases that are treated differently over cases treated the same

No Guarantee

- Executing all control flow elements does not guarantee finding all faults.
  - Execution of a faulty statement may not always result in a failure.
    - The state may not be corrupted when the statement is executed with some data values.
    - Corrupt state may not propagate through execution to eventually lead to failure.

- What is the value of structural coverage?
  - <u>Increases confidence in thoroughness of testing</u>

Structural Testing Complements Functional Testing

- Control flow-based testing includes cases that may not be identified from specifications alone.
  - Typical case: Implementation of a single item of the specification by multiple parts of the program
  - E.g. Hash table collision (invisible in interface specification)

- Test suites that satisfy control flow adequacy criteria could fail in revealing faults that can be caught with functional criteria.
  - Typical case: Missing path faults

Structural Testing, in Practice

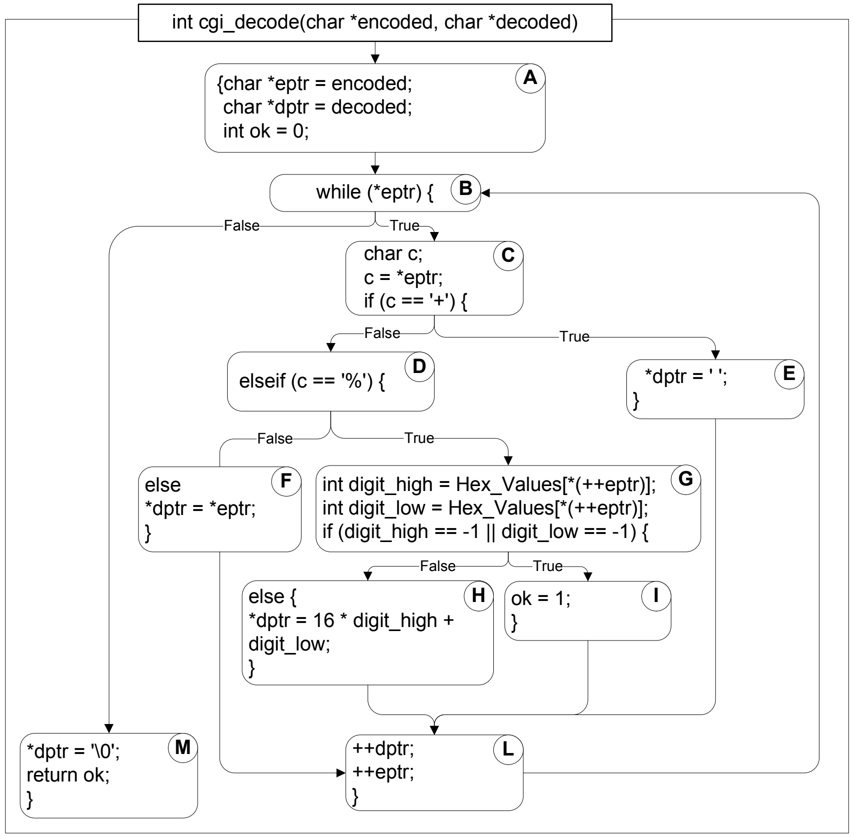
- Create functional test suite first, then measure structural coverage to identify and see what is missing.

- Interpret unexecuted elements
  - May be due to natural differences between specification and implementation
  - May reveal flaws of the software or its development process
    - Inadequacy of specifications that do not include cases present in the implementation
    - Coding practice that radically diverges from the specification
    - Inadequate functional test suites

- Attractive because structural testing is automated
  - Coverage measurements are convenient progress indicators.
  - Sometimes used as a criterion of completion of testing
    - Use with caution: does not ensure effective test suites

An Example Program: ‘cgi_decode’ and CFG

```
#include “hex_values.h”

int cgi_decode(char* encoded, char* *decoded) {
   char *eptr = encoded;
   char *dptr = decoded;
   int ok = 0;

   while (*eptr) {
      char c;
      c = *eptr;

      if (c == ‘+’) {
         *dptr = ‘ ‘;
      } else if (c = ‘%’) {
        int digit_high = Hex_Values[*(++eptr)];
        int digit_low = Hex_Values[*(++eptr)];

        if (digit_high == -1 || digit_low == -1) {
            ok = 1;
         } else {
            *dptr = 16 * digit_high + digit_low;
         }
      } else {
         *dptr = *eptr;
      }
      ++dptr;
      ++eptr;
   }

   *dptr = ‘₩0’;
   return ok;
}
```



Structural Testing Techniques

1. Statement Testing

2. Branch Testing

3. Condition Testing
   – Basic
   – Compounded
   – MC/DC

4. Path Testing
   – Bounded interior
   – Loop boundary
   – LCSAJ
   – Cyclomatic

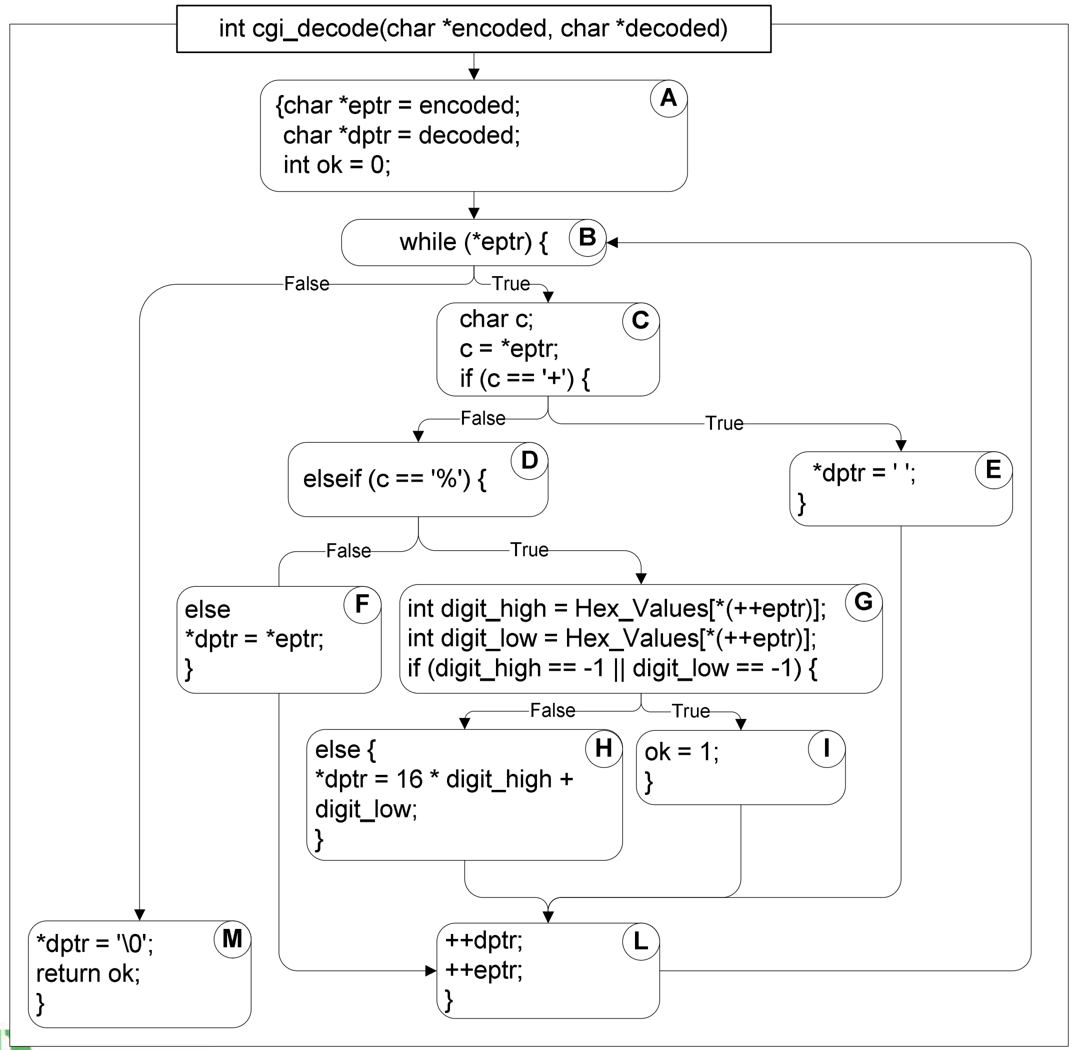
1. Statement Testing

- Adequacy criterion:
  - Each statement (or node in the CFG) must be executed at least once.
- Coverage:

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

- Rationale:
  - A fault in a statement can only be revealed by executing the faulty statement.
- Nodes in a CFG often represent basic blocks of multiple statements.
  - Some standards refer to 'basic block coverage' or 'node coverage'.
  - Difference in granularity, but not in concept

An Example: for Function “cgi_decode”



< Test cases >

T_0 =
{“”, “test”, “test+case%1Dadequacy”}
17/18 = 94% Statement coverage

T_1 =
{“adequate+test%0Dexecution%7U”}
18/18 = 100% Statement coverage

T_2 = {“%3D”, “%A”, “a+b”, “test”}
18/18 = 100% Statement coverage

T_3 = {“ ”, “+%0D+%4J”}
…

T4 = {“first+test%9Ktest%K9”}
…

Coverage is not a Matter of Size

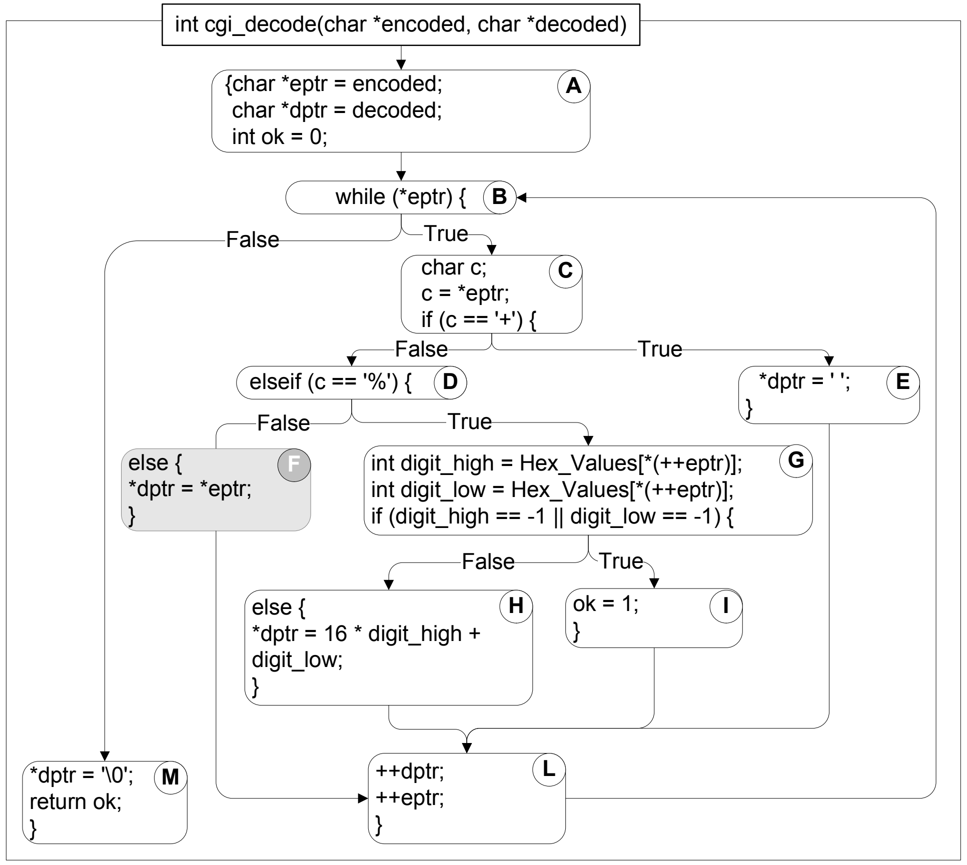
- Coverage does not depend on the number of test cases.
  - T_0 , T_1 : $T_1 >_{coverage} T_0$ $T_1 <_{cardinality} T_0$
  - T_1 , T_2 : $T_2 =_{coverage} T_1$ $T_2 >_{cardinality} T_1$
- Minimizing test suite size is not the goal.
  - Small test cases make failure diagnosis easier.
  - But, a failing test case in T_2 gives more information for fault localization than a failing test case in T_1

Complete Statement Coverage

- Complete statement coverage may not imply executing all branches in a program.

- Example:
  - Suppose block F were missing
  - But, statement adequacy would not require false branch from D to L

- T3 = {“ ”, “+%0D+%4J”}
  - 100% statement coverage
  - No false branch from D



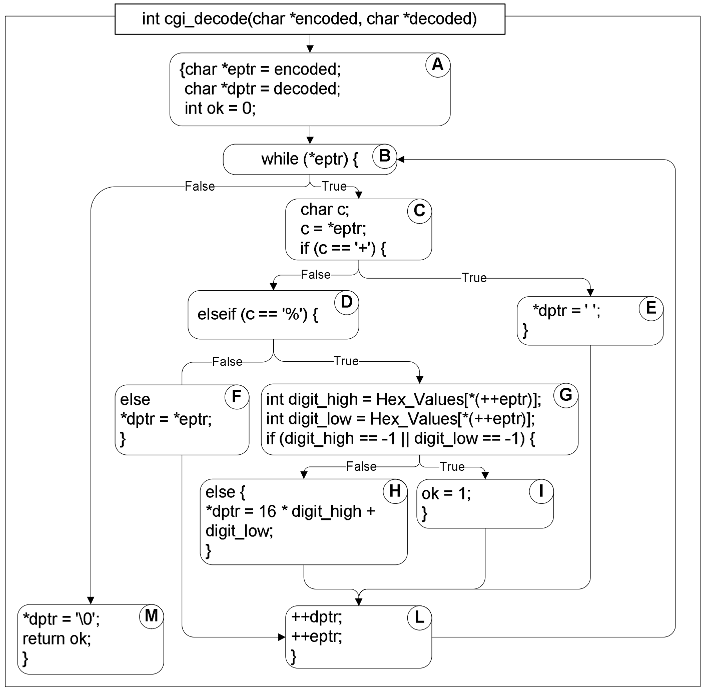
2. Branch Testing

- Adequacy criterion:
  - Each branch (edge in the CFG) must be executed at least once.

- Coverage:

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

- Example:
  - T_3 = {“”, “+%0D+%4J”}
    - 100% Stmt Cov.
    - 88% Branch Cov. (7/8 branches)
  - T_2 = {“%3D”, “%A”, “a+b”, “test”}
    - 100% Stmt Cov.
    - 100% Branch Cov. (8/8 branches)



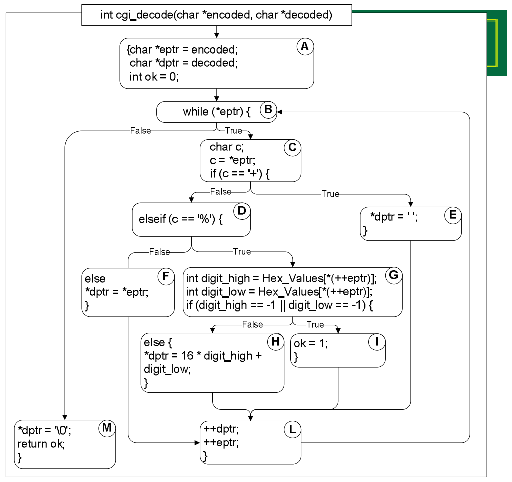
```
int cgi_decode(char *encoded, char *decoded)

A: {char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;

B: while (*eptr) {

C: char c;
   c = *eptr;
   if (c == '+') {

D: elseif (c == '%') {

E: *dptr = ' ';
   }

F: else
   *dptr = *eptr;
   }

G: int digit_high = Hex_Values[*(++eptr)];
   int digit_low = Hex_Values[*(++eptr)];
   if (digit_high == -1 || digit_low == -1) {

H: else {
   *dptr = 16 * digit_high +
   digit_low;
   }

I: ok = 1;
   }

L: ++dptr;
   ++eptr;
   }

M: *dptr = '\0';
   return ok;
   }
```

Statements vs. Branches

- Traversing all edges causes all nodes to be visited.
  - Therefore, test suites that satisfy the branch adequacy also satisfy the statement adequacy criterion for the same program.
  - Branch adequacy subsumes statement adequacy.

- The converse is not true (see T_3)
  - A statement-adequate test suite may not be branch-adequate.

All Branches Coverage



int cgi_decode(char *encoded, char *decoded)
{char *eptr = encoded; char *dptr = decoded; int ok = 0;
A
while (*eptr) {
B
False
True
char c; c = *eptr; if (c == '+') {
C
False
True
elseif (c == '%') {
D
*dptr = ' '; }
E
False
True
else *dptr = *eptr; }
F
int digit_high = Hex_Values[*(++eptr)]; int digit_low = Hex_Values[*(++eptr)]; if (digit_high == -1 || digit_low == -1) {
G
False
True
else { *dptr = 16 * digit_high + digit_low; }
H
ok = 1; }
I
*dptr = '\0'; return ok; }
M
++dptr; ++eptr; }
L

- "All branches coverage" can still miss conditions.

- Example:
  - Supposed that we missed the negation operator of "digit_high == -1"

    digit_high == 1 || digit_low == -1

- Branch adequacy criterion can be satisfied by varying only 'digit_low'.
  - The faulty sub-expression might never determine the result.
  - We might never really test the faulty condition, even though we tested both outcomes of the branch.

DEPENDABLE SOFTWARE LABORATORY

3. Condition Testing

- Branch coverage exposes faults in how a computation has been decomposed into cases.
  - Intuitively attractive: checking the programmer's case analysis
  - But, only roughly: grouping cases with the same outcome

- Condition coverage considers case analysis in more detail.
  - Consider 'individual conditions' in a compound Boolean expression
    - E.g. both parts of '"igit_high == 1 || digit_low == -1"

- Adequacy criterion:
  - Each basic condition must be executed at least once.

- Basic condition testing coverage:

$$\frac{\text{number of truth values taken by all basic conditions}}{2 * \text{number of basic conditions}}$$

Basic Conditions vs. Branches

- Basic condition adequacy criterion can be satisfied without satisfying branch coverage.

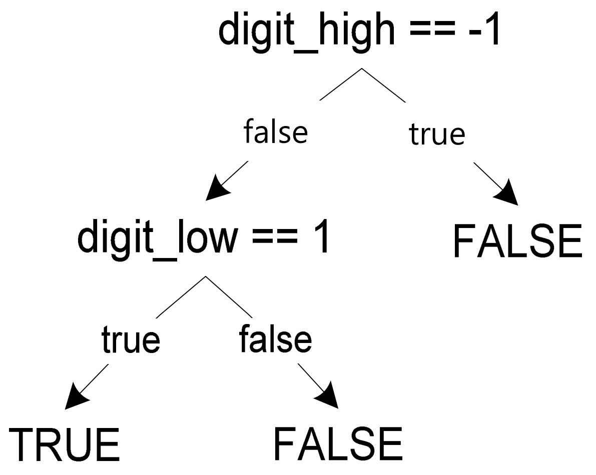
- T4 = {“first+test%9Ktest%K9”}
  - Satisfies basic condition adequacy
  - But, does not satisfy branch condition adequacy

- Branch and basic condition are not comparable.
  - Neither implies the other.

Covering Branches and Conditions

- Branch and condition adequacy:
  - Cover all conditions and all decisions

- Compound condition adequacy:
  - Cover all possible evaluations of compound conditions.
  - Cover all branches of a decision tree.



Compounded Conditions

- Compound conditions often have exponential complexity.
- Example: (((a || b) && c) || d) && e

Test Case	a	b	c	d	e
(1)	T	—	T	—	T
(2)	F	T	T	—	T
(3)	T	—	F	T	T
(4)	F	T	F	T	T
(5)	F	F	—	T	T
(6)	T	—	T	—	F
(7)	F	T	T	—	F
(8)	T	—	F	T	F
(9)	F	T	F	T	F
(10)	F	F	—	T	F
(11)	T	—	F	F	—
(12)	F	T	F	F	—
(13)	F	F	—	F	—

Modified Condition/Decision (MC/DC)

- Motivation
  - Effectively test important combinations of conditions, without exponential blowup in test suite size
  - "Important" combinations means:
    - Each basic condition shown to independently affect the outcome of each decision

- Requires
  - For each basic condition C, two test cases,
  - Values of all 'evaluated' conditions except C are the same.
  - Compound condition as a whole evaluates to 'true' for one and 'false' for the other.

Complexity of MC/DC

- MC/DC has a linear complexity.

- Example: (((a || b) && c) || d) && e

Test Case	a	b	c	d	e	outcome
(1)	true	--	true	--	true	true
(2)	false	true	true	--	true	true
(3)	true	--	false	true	true	true
(6)	true	--	true	--	false	false
(11)	true	--	false	false	--	false
(13)	false	false	--	false	--	false

- Underlined values independently affect the output of the decision.
  - Required by the RTCA/DO-178B standard

Comments on MC/DC

- MC/DC is
  - Basic condition coverage (C)
  - Branch coverage (DC)
  - Plus one additional condition (M)
    - Every condition must independently affect the decision's output.

- It is subsumed by compound conditions and subsumes all other criteria discussed so far.
  - Stronger than statement and branch coverage

- A good balance of thoroughness and test size
  - Widely used

4. Path Testing

- There are many more paths than branches.
  - Decision and condition adequacy criteria consider individual decisions only.

- Path testing focuses combinations of decisions along paths.

- Adequacy criterion:
  - Each path must be executed at least once.

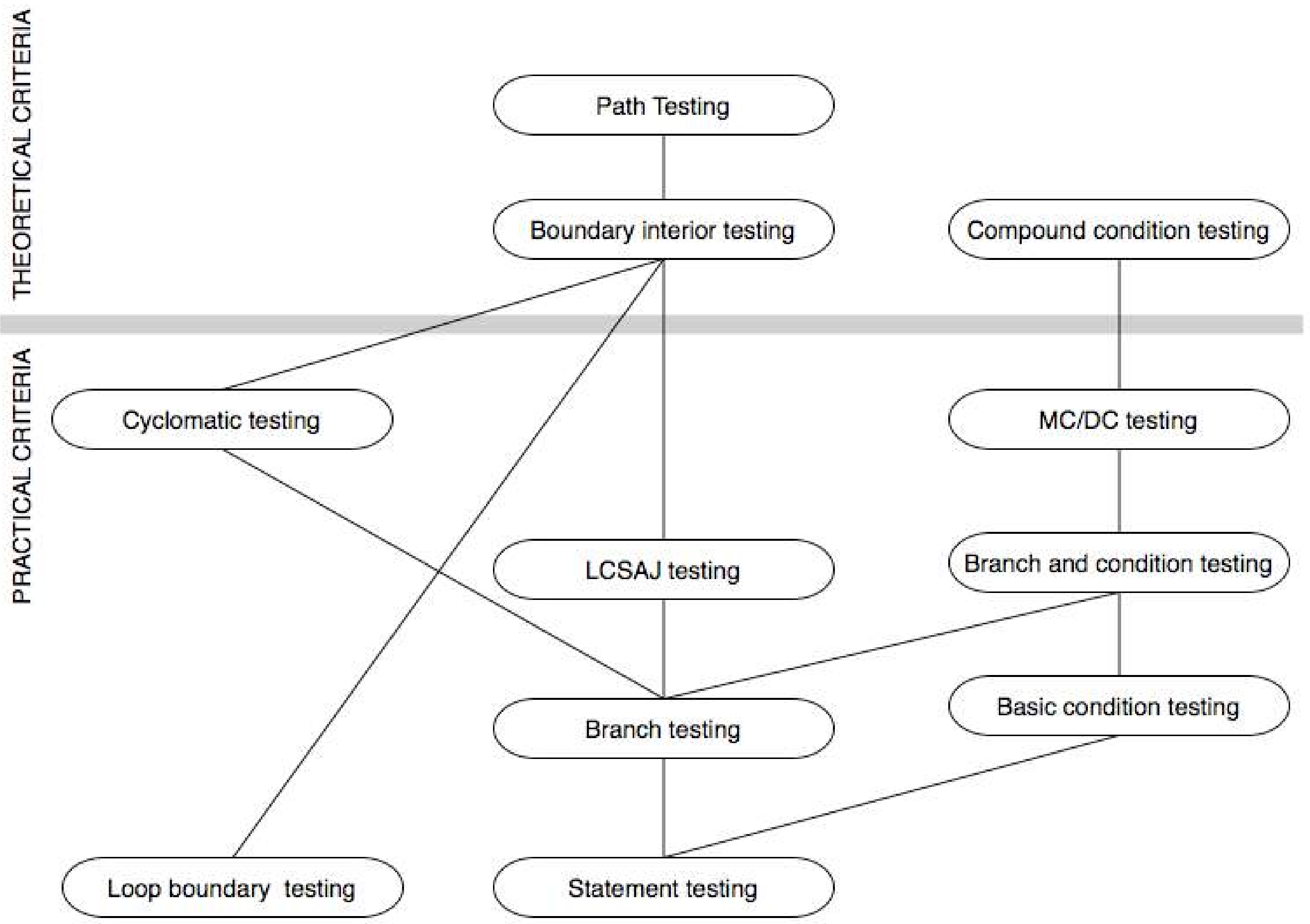
- Coverage:

$$\frac{\text{number of executed paths}}{\text{number of paths}}$$

Path Coverage Criteria in Practice

- The number of paths in a program with loops is unbounded.
  - Usually impossible to satisfy
- For a feasible criterion,
  - Should partition infinite set of paths into a finite number of classes
- Useful criteria can be obtained by limiting
  - Number of traversals of loops
  - Length of the paths to be traversed
  - Dependencies among selected paths

Comparing Structural Testing Criteria



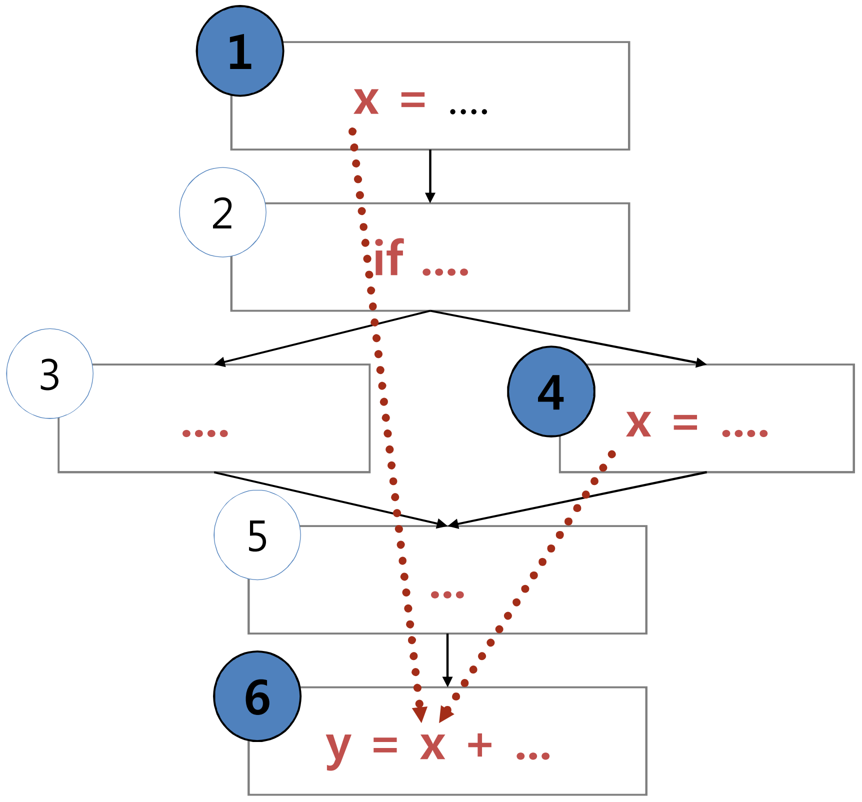
Subsumption Relation among Structural Test Adequacy Criteria

Motivation

- Middle ground in structural testing
  - Node and edge coverage don’t test interactions.
  - Path-based criteria require impractical number of test cases.
    - Only a few paths uncover additional faults, anyway.
  - Need to distinguish “important” paths

- Intuition: Statements interact through data flow.
  - Value computed in one statement, is used in another.
  - Bad value computation can be revealed only when it is used.

Def-Use Pairs



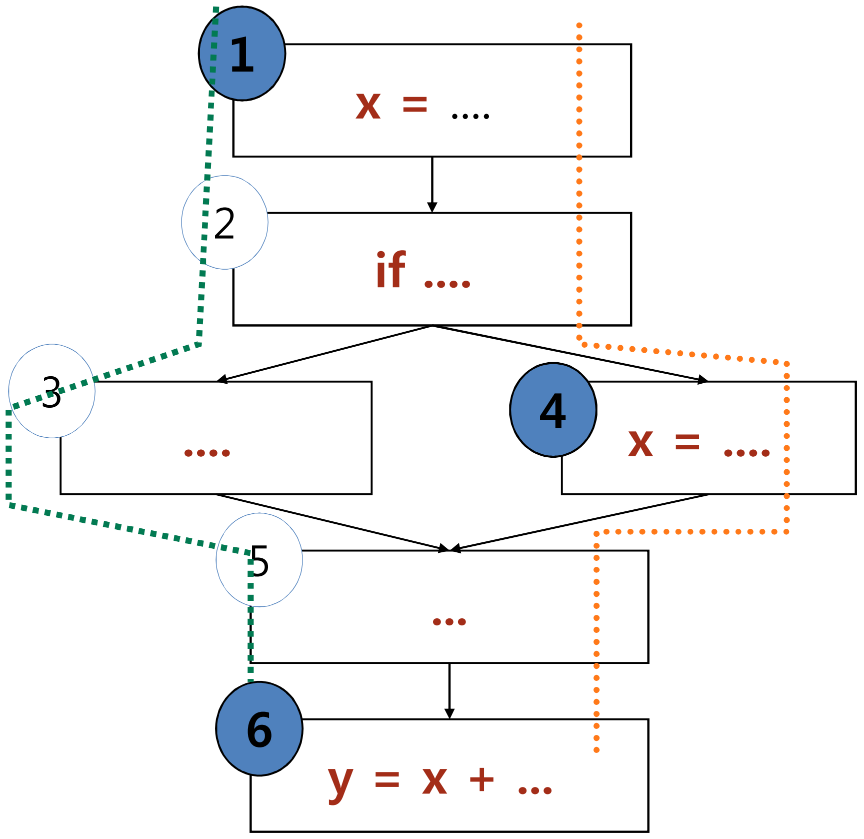
- Value of x at 6 could be computed at 1 or at 4.
- Bad computation at 1 or 4 could be revealed only if they are used at 6.
- (1, 6) and (4, 6) are def-use (DU) pairs.
  - defs at 1, 4
  - use at 6

Terminology

- DU pair
  - A pair of definition and use for some variable, such that at least one DU path exists from the definition to the use.
  - "x = …" is a definition of x
  - "= … x …" is a use of x

- DU path
  - A definition-clear path on the CFG starting from a definition to a use of a same variable
  - Definition clear: Value is not replaced on path.
  - Note: Loops could create infinite DU paths between a def and a use.

Definition-Clear Path



- 1,2,3,5,6 is a definition-clear path from 1 to 6.
  - x is not re-assigned between 1 and 6.
- 1,2,4,5,6 is not a definition-clear path from 1 to 6.
  - the value of x is "killed" (reassigned) at node 4.
- (1, 6) is a DU pair because 1,2,3,5,6 is a definition-clear path.

Adequacy Criteria

- All DU pairs
  - Each DU pair is exercised by at least one test case.
- All DU paths
  - Each simple (non looping) DU path is exercised by at least one test case.
- All definitions
  - For each definition, there is at least one test case which exercises a DU pair containing it.
  - Because, every computed value is used somewhere.
- Corresponding coverage fractions can be defined similarly.

Difficult Cases

- x[i] = … ; … ; y = x[j]
  - DU pair (only) if i==j
- p = &x ; … ; *p = 99 ; … ; q = x
  - *p is an alias of x
- m.putFoo(…); … ; y=n.getFoo(…);
  - Are m and n the same object?
  - Do m and n share a "foo" field?
- Problem of aliases:
  - Which references are (always or sometimes) the same?

Data Flow Coverage in Practice

- The path-oriented nature of data flow analysis makes the infeasibility problem especially relevant.
  - Combinations of elements matter.
  - Impossible to (infallibly) distinguish feasible from infeasible paths.
  - More paths = More work to check manually

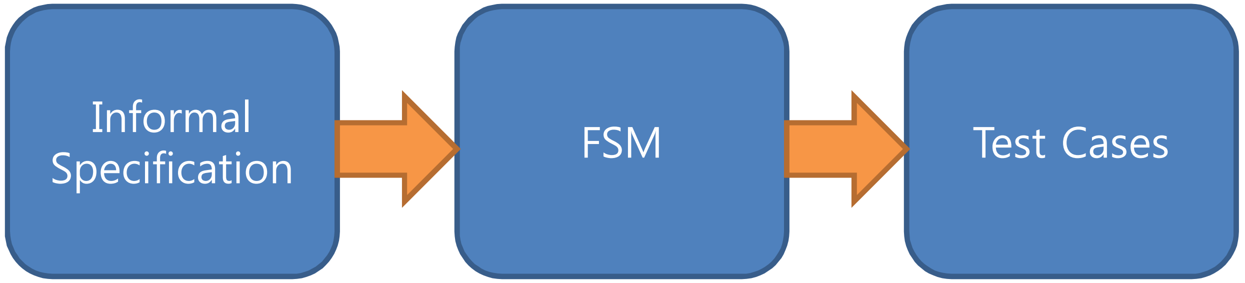
- In practice, reasonable coverage is (often, not always) achievable.
  - Number of paths is exponential in worst case, but often linear.
  - All DU paths is more often impractical.

Overview

- Models used in specification or design have structure.
  - Useful information for selecting representative classes of behavior
  - Behaviors that are treated differently with respect to the model should be tried by a thorough test suite.
  - In combinatorial testing, it is difficult to capture that structure clearly and correctly in constraints.

- We can devise test cases to check actual behavior against behavior specified by the model.
  - "Coverage" similar to structural testing, but applied to specification and design models

Deriving Test Cases from Finite State Machines



Informal Specification: Feature “Maintenance” of the Chipmunk Web Site

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.

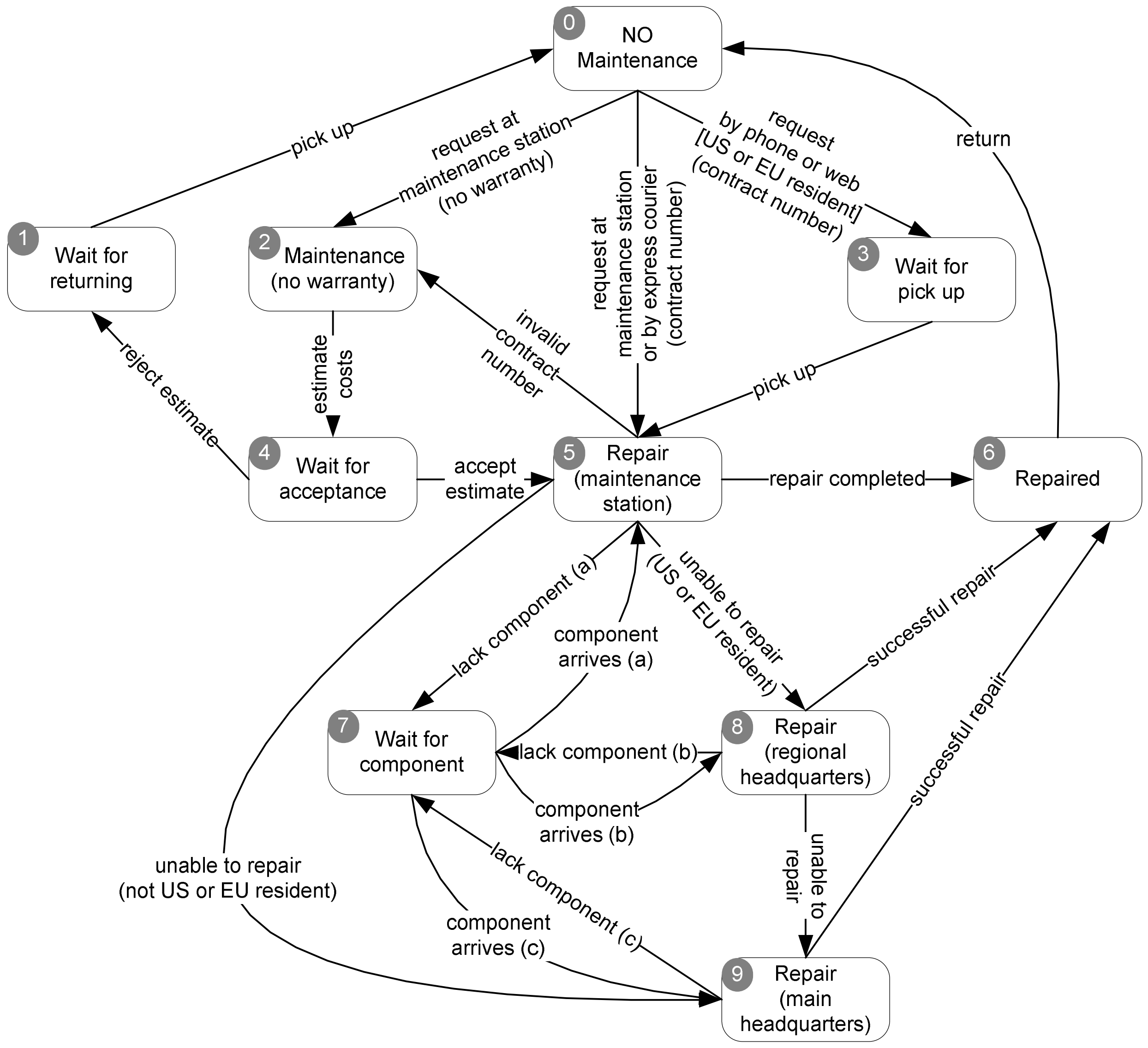
Small problems can be repaired directly at the maintenance station. If the maintenance station cannot solve the problem, the product is sent to the maintenance regional headquarters (if in US or EU) or to the maintenance main headquarters (otherwise).

If the maintenance regional headquarters cannot solve the problem, the product 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.

Corresponding Finite State Machine



Test Cases Generated from the FSM

- FSM can be used both to
  1. Guide test selection (checking each state transition)
  2. Constructing an oracle that judge whether each observed behavior is correct

TC1	0	2	4	1	0						
TC2	0	5	2	4	5	6	0				
TC3	0	3	5	9	6	0					
TC4	0	3	5	7	5	8	7	8	9	6	0

- Questions:
  - Is this a thorough test suite?
  - How can we judge?

    → Coverage criteria require.

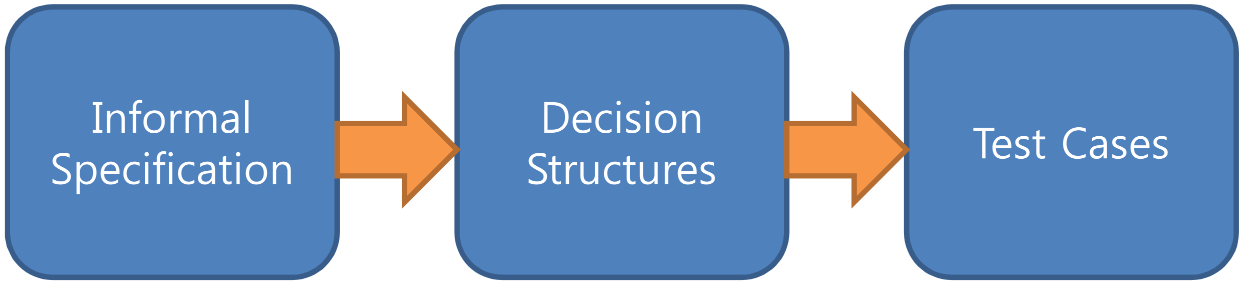
Transition Coverage Criteria

- State coverage
  - Every state in the model should be visited by at least one test case.

- Transition coverage
  - Every transition between states should be traversed by at least one test case.
  - Most commonly used criterion
  - A transition can be thought of as a (precondition, postcondition) pair

Deriving Test Cases from Decision Structures

- Some specifications are structured as decision tables, decision trees, or flow charts.
- We can exercise these as if they were program source code.



Corresponding Decision Table

	Education		Individual					
EduAc	T	T	F	F	F	F	F	F
BusAc	-	-	F	F	F	F	F	F
CP > CT1	-	-	F	F	T	T	-	-
YP > YT1	-	-	-	-	-	-	-	-
CP > CT2	-	-	-	-	F	F	T	T
YP > YT2	-	-	-	-	-	-	-	-
SP < Sc	F	T	F	T	-	-	-	-
SP < T1	-	-	-	-	F	T	-	-
SP < T2	-	-	-	-	-	-	F	T
Out	Edu	SP	ND	SP	T1	SP	T2	SP

…

Constraints

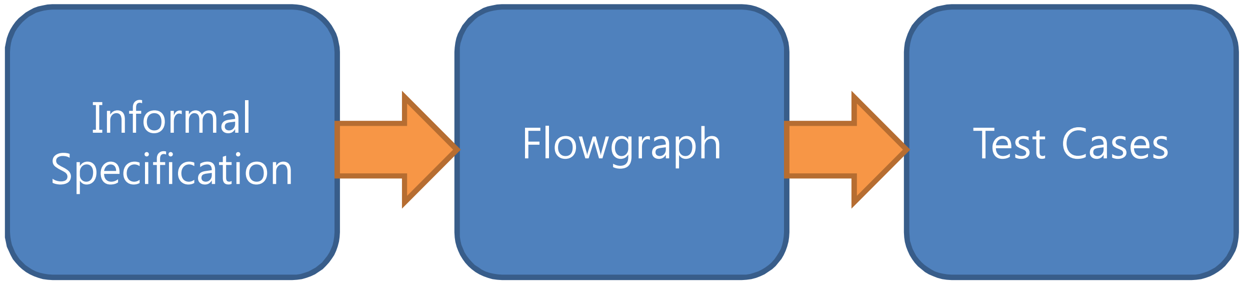
at-most-one (EduAc, BusAc)
YP > YT2 → YP > YT1
CP > CT2 → CP > CT1
SP > T2 → SP > T1

at-most-one (YP < YT1, YP > YT2)
at-most-one (CP < CT1, CP > CT2)
at-most-one (SP < T1, SP > T2

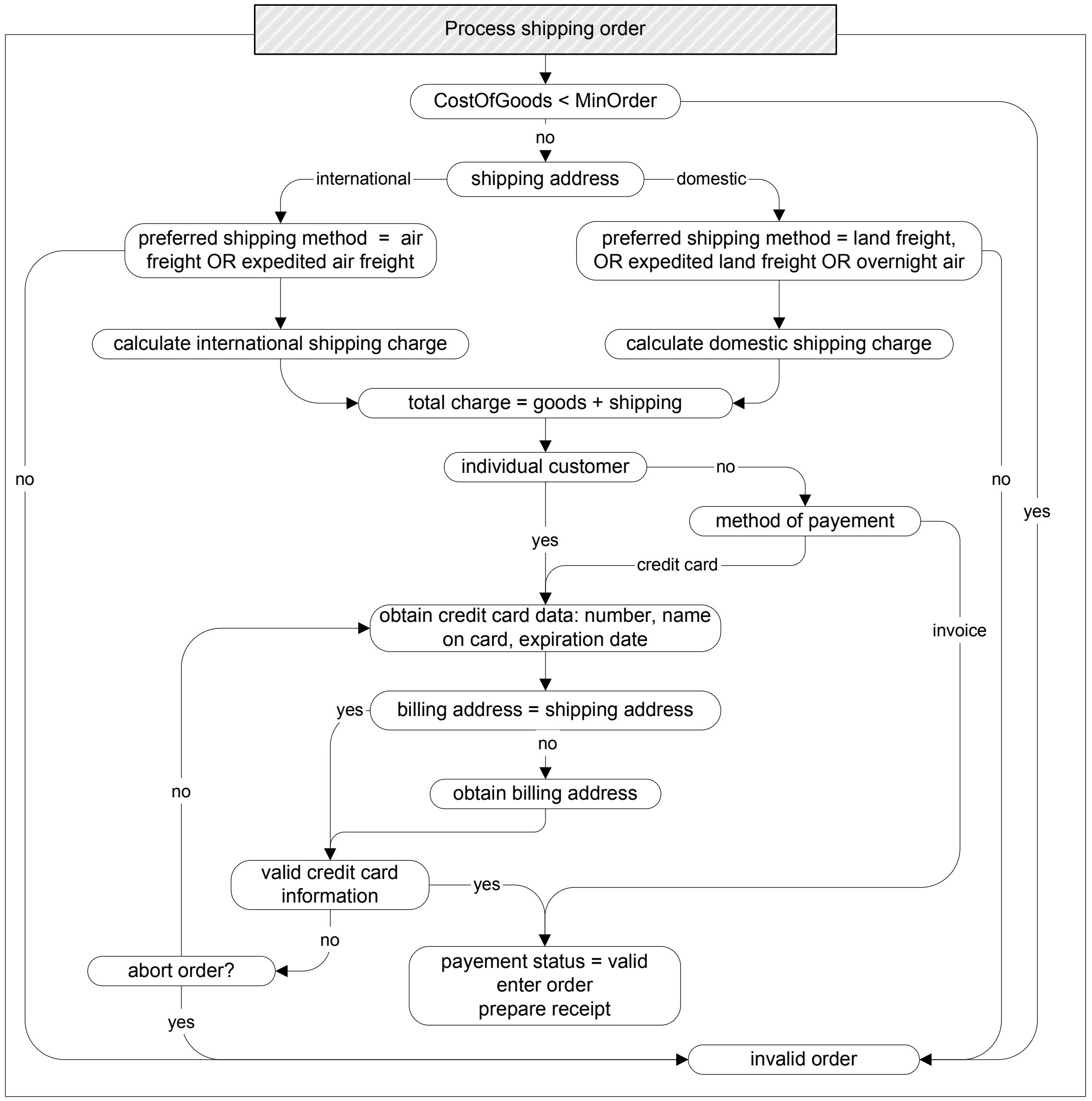
Deriving Test Cases from Control and Data Flow Graph

- If the specification or model has both decisions and sequential logic, we can cover it like program source code.

- Flowgraph based testing



Corresponding Control Flow Graph


Process shipping order
CostOfGoods < MinOrder
no
yes
shipping address
international
domestic
preferred shipping method = air freight OR expedited air freight
preferred shipping method = land freight, OR expedited land freight OR overnight air
no
no
calculate international shipping charge
calculate domestic shipping charge
total charge = goods + shipping
individual customer
no
method of payement
yes
credit card
invoice
obtain credit card data: number, name on card, expiration date
billing address = shipping address
yes
no
obtain billing address
no
valid credit card information
yes
no
abort order?
yes
payement status = valid
enter order
prepare receipt
invalid order

Test Cases Generated from the CFG

- Node adequacy criteria

Case	Too Small	Ship Where	Ship Method	Cust Type	Pay Method	Same Address	CC valid
TC-1	No	Int	Air	Bus	CC	No	Yes
TC-2	No	Dom	Air	Ind	CC	-	No (abort)

- Branch adequacy criteria

Case	Too Small	Ship Where	Ship Method	Cust Type	Pay Method	Same Address	CC valid
TC-1	No	Int	Air	Bus	CC	No	Yes
TC-2	No	Dom	Land	-	-	-	-
TC-3	Yes	-	-	-	-	-	-
TC-4	No	Dom	Air	-	-	-	-
TC-5	No	Int	Land	-	-	-	-
TC-6	No	-	-	Edu	Inv	-	-
TC-7	No	-	-	-	CC	Yes	-
TC-8	No	-	-	-	CC	-	No (abort)
TC-9	No	-	-	-	CC	-	No (no abort)

Estimating Test Suite Quality

- Supposed that I have a program with bugs.
- Add 100 new bugs
  - Assume they are exactly like real bugs in every way
  - I make 100 copies of my program, each with one of my 100 new bugs.
- Run my test suite on the programs with seeded bugs
  - And the tests revealed 20 of the bugs.
  - The other 80 program copies do not fail.
- What can I infer about my test suite's quality?

Basic Assumptions

- We want to judge effectiveness of a test suite in finding real faults,
  - by measuring how well it finds seeded fake faults.

- Valid to the extent that the seeded bugs are representative of real bugs
  - Not necessarily identical
  - But, the differences should not affect the selection

Mutation Testing

- A mutant is a copy of a program with a mutation.
- A mutation is a syntactic change (a seeded bug).
  - Example: change (i < 0) to (i <= 0)
- Run test suite on all the mutant programs
- A mutant is killed, if it fails on at least one test case. (The bug is found.)
- <u>If many mutants are killed, infer that the test suite is also effective at finding real bugs.</u>

Assumptions on Mutation Testing

- Competent programmer hypothesis
  - Programs are nearly correct.
    - Real faults are small variations from the correct program.
    - Therefore, mutants are reasonable models of real buggy programs.

- Coupling effect hypothesis
  - Tests that find simple faults also find more complex faults.
  - Even if mutants are not perfect representatives of real faults, a test suite that kills mutants is good at finding real faults too.

Mutant Operators

- Syntactic changes from legal program to illegal program
  - Specific to each programming language

- Examples:
  - crp: constant for constant replacement
    - E.g. from (x < 5) to (x < 12)
    - Select from constants found somewhere in program text
  - ror: relational operator replacement
    - E.g. from (x <= 5) to (x < 5)
  - vie: variable initialization elimination
    - E.g. change int x =5; to int x;

Fault-based Adequacy Criteria

- Mutation analysis consists of the following steps:
  1. Select mutation operators
  2. Generate mutants
  3. Distinguish mutants

- Live mutants
  - Mutants not killed by a test suite

- Given a set of mutants SM and a test suite T, <u>the fraction of nonequivalence mutants killed by T</u> measures the adequacy of T with respect to SM.

Automating Test Execution

- Designing test cases and test suites is creative.
  - Demanding intellectual activity
  - Requiring human judgment
- Executing test cases should be automatic.
  - Design once, execute many times
- Test automation separates the creative human process from the mechanical process of test execution.

From Test Case Specifications to Test Cases

- Test design often yields test case specifications, rather than concrete data.
  - E.g. "a large positive number", not 420,023
  - E.g. "a sorted sequence, length > 2", not "Alpha, Beta, Chi, Omega"
- Other details for execution may be omitted.

- Test Generation creates concrete, executable test cases from test case specifications.

- A Tool chain for test case generation & execution
  - A combinatorial test case generation to create test data
    - Optional: Constraint-based data generator to "concretize" individual values, e.g., from "positive integer" to 42
  - 'DDSteps' to convert from spreadsheet data to 'JUnit' test cases
  - 'JUnit' to execute concrete test cases

Scaffolding

- Code produced to support development activities
  - Not part of the “product” as seen by the end user
  - May be temporary (like scaffolding in construction of buildings)

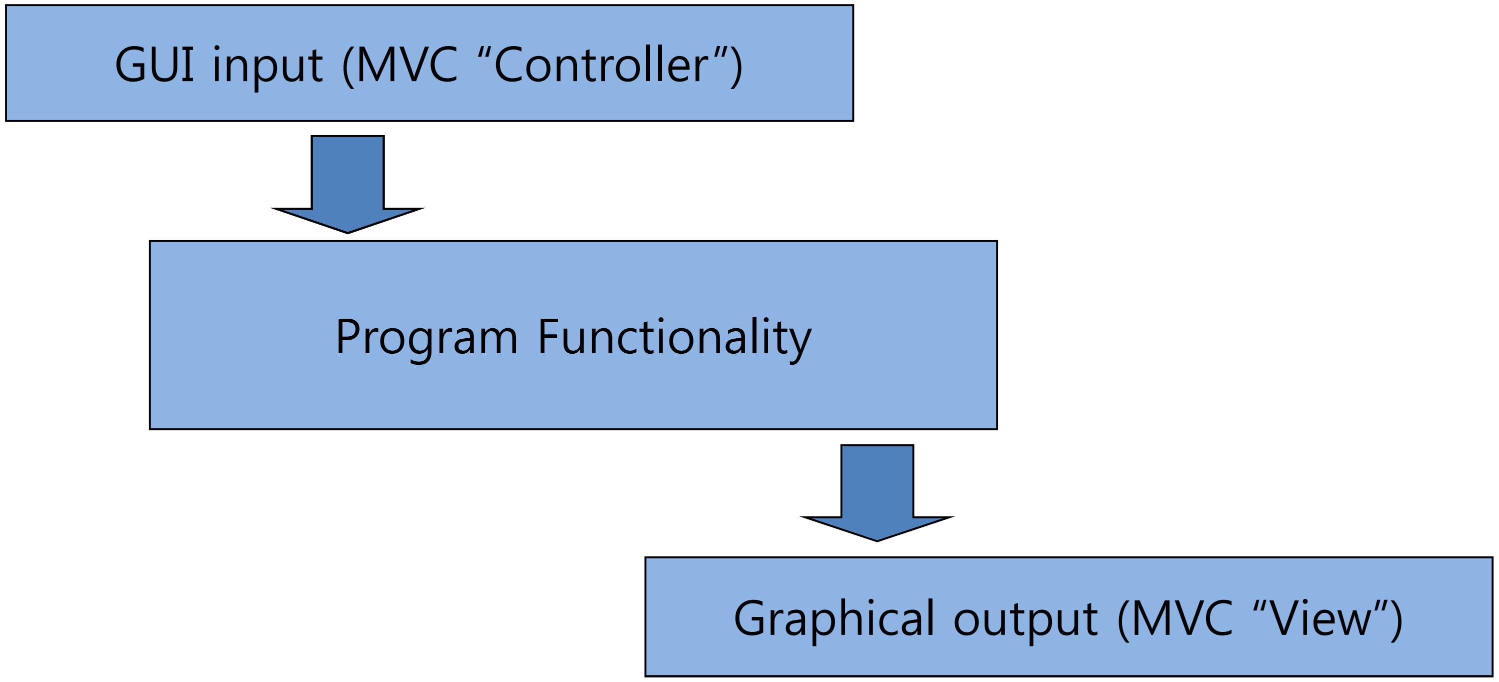
- Scaffolding includes
  - Test harnesses
  - Drivers
  - Stubs

Scaffolding

- Test driver
  - A "main" program for running a test
    - May be produced before a "real" main program
    - Provide more control than the "real" main program
  - To drive program under test through test cases
- Test stub
  - Substitute for called functions/methods/objects
- Test harness
  - Substitutes for other parts of the deployed environment
  - E.g. Software simulation of a hardware device

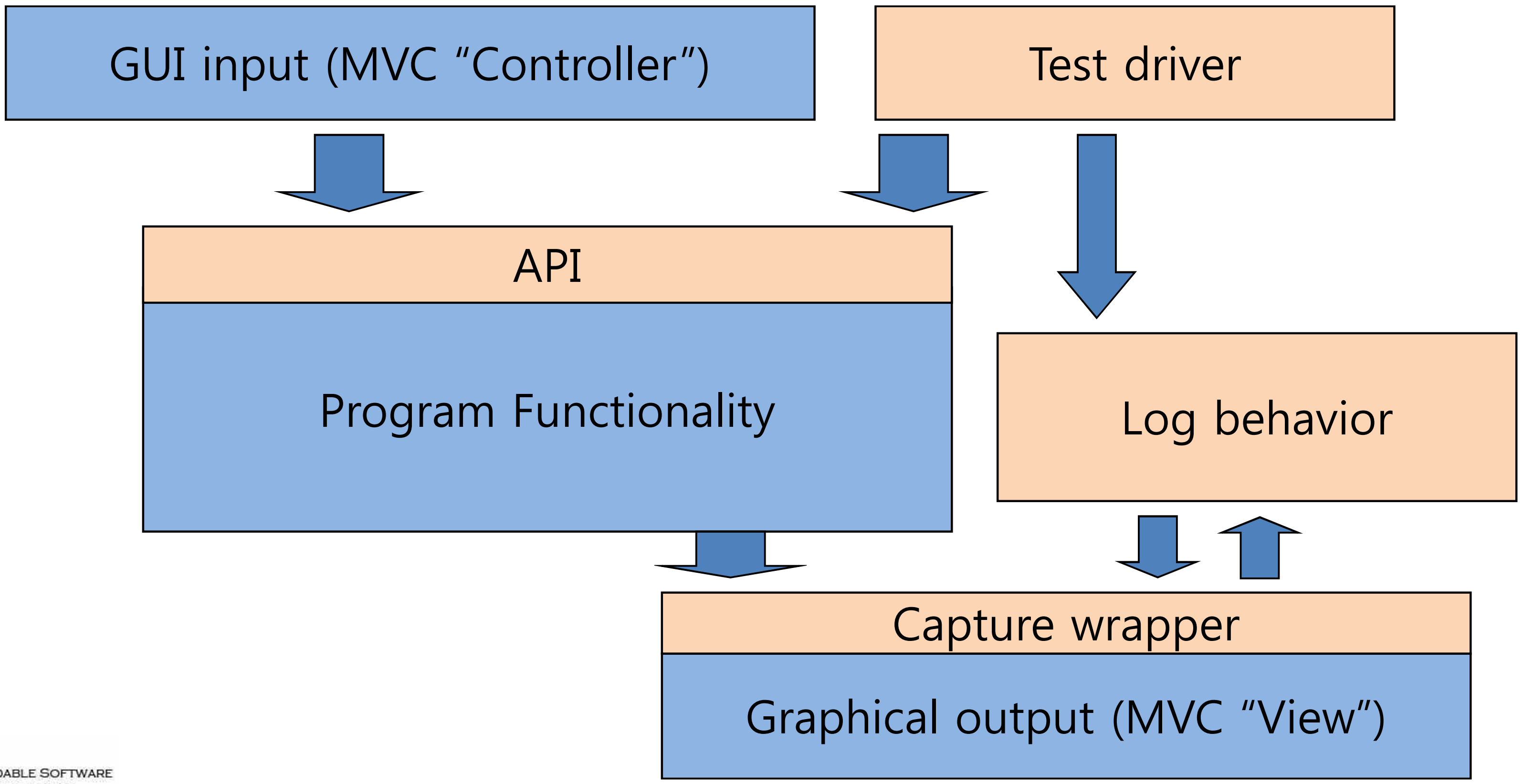
Controllability & Observability

- Example: We want to automate tests,
  - But, interactive input provides limited control and
  - Graphical output provides limited observability.



Controllability & Observability

- Solution: A design for automated test provides interfaces for control (API) and observation (wrapper on output)



Generic vs. Specific Scaffolding

- How general should scaffolding be?
  - We could build a driver and stubs for each test case.
  - Or at least factor out some common code of the driver and test management (e.g. JUnit)
  - Or further factor out some common support code, to drive a large number of test cases from data (as in DDSteps)
  - Or further generate the data automatically from a more abstract model (e.g. network traffic model)

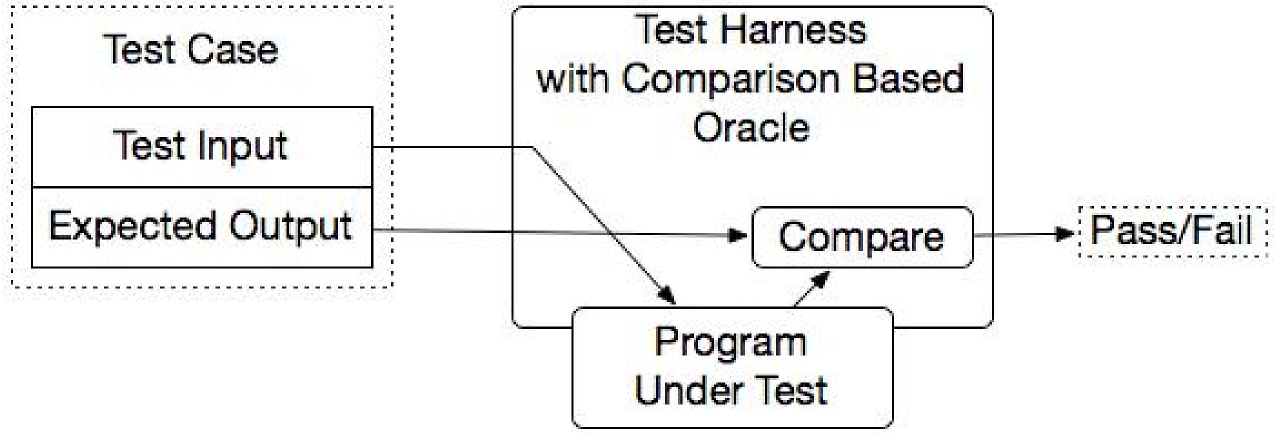
- It's a question of costs and re-use, just as for other kinds of software.

Test Oracles

- No use running 10,000 test cases automatically, if the results must be checked by hand.
- It's a problem of 'range of specific to general', again
  - E.g. JUnit: Specific oracle ("assert") coded by hand in each test case
- Typical approach
  - Comparison-based oracle with predicted output value
  - But, not the only approach

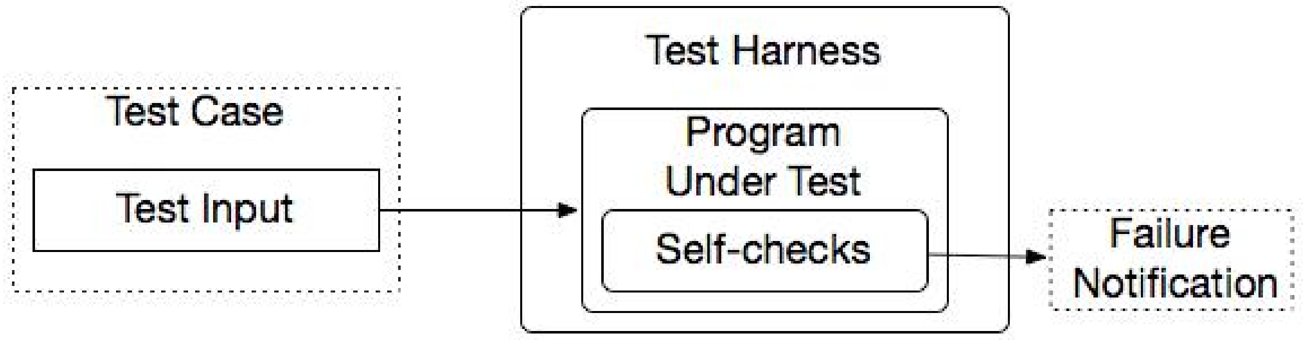
Comparison-based Oracle

- With a comparison-based oracle, we need predicted output for each input.
  - Oracle compares actual to predicted output, and reports failure if they differ.
  - Fine for a small number of hand-generated test cases
  - E.g. for hand-written JUnit test cases



Self-Checks as Oracles

- An oracle can also be written as self-checks.
  - Often possible to judge correctness without predicting results
- Advantages and limits: Usable with large, automatically generated test suites, but often only a partial check
  - E.g. structural invariants of data structures
  - Recognize many or most failures, but not all



Overview

- Automated program analysis techniques complement test and inspection in two ways:
  - Can exhaustively check some important properties
    - Which conventional testing is particularly ill-suited
  - Can extract and summarize information for test and inspection design
    - Replacing or augmenting human efforts

- Automated analysis
  - Replace human inspection for some class of faults
  - Support inspection by
    - Automating extracting and summarizing information
    - Navigating through relevant information

Static vs. Dynamic Analysis

- Static analysis
  - Examine program source code
    - Examine the complete execution space
    - But, may lead to false alarms
- Dynamic analysis
  - Examine program execution traces
    - No infeasible path problem
    - But, cannot examine the execution space exhaustively
- Example:
  - Concurrency faults
  - Memory faults

Extracting Behavior Model from Execution

- Behavior analysis can
  - Gather information from executing several test cases
  - And synthesize a model that characterizes those execution,
  - To the extent that they are the representative of other executions as well.

- Using behavioral models for
  - Testing : validate tests thoroughness
  - Program analysis : understand program behavior
  - Regression testing : compare versions or configurations
  - Testing of component-based software : compare components in different contexts
  - Debugging : Identify anomalous behaviors and understand causes