Software Testing and Quality Assurance

Lecture 2

Agenda

- Software Testing from theoretical standpoint
  - Verification Vs Validation
  - Static Vs Dynamic Testing
  - Adequacy
- Graph Based Testing

Input and Output Space

- Let **U** be the set of numbers and characters that can be represented on some computer system.
- Let **I** be the <u>input space</u>, and **O** be the <u>output space</u>.
  - Both **I** and **O** are equal to the set of finite sequences of **U**.

Specifications

- A *specification* is a relation $\mathbf{S} \subseteq \mathbf{I} \times \mathbf{O}$
- The *input domain of specification* $\boldsymbol{S}$ is the set
  $\mathbf{D}^{\mathbf{S}} = \{ \, i \in \mathbf{I} \mid \exists o \in \mathbf{O}, (i, o) \in \mathbf{S} \, \}$
- Ideally, a specification is a *total* function, which describes an intended behavior of the program for every possible input.
- In practice, a specification is a *partial* function, whose domain is the set of all values for which $\mathbf{S}$ is defined.

Programs

- A program defines a partial function
  $$\mathbf{P} : \mathbf{I} \rightarrow \mathbf{O}$$
- The *input domain of program* $\boldsymbol{P}$ is the set
  $\mathbf{D}^{\mathbf{P}} = \{\boldsymbol{i} \in \mathbf{I} \mid \mathbf{P}$ halts on input $\boldsymbol{i}\ \}$
  $= \{\boldsymbol{i} \in \mathbf{I} \mid \mathrm{P}(\boldsymbol{i})$ is defined $\}$
- Those inputs which cause program $\mathbf{P}$ to crash or go into an infinite loop are not in $\mathbf{D}^{\mathbf{P}}$

Program Correctness

- Program **P** is correct w.r.t. specification **S**, iff

  $\mathbf{D}^{\mathbf{S}} \subseteq \mathbf{D}^{\mathbf{P}}$ and $\forall i \in \mathbf{DS}, (i, \mathbf{P}(i)) \in \mathbf{S}$

- That is, **P** is correct w.r.t. **S** iff for each element i on the input domain of **S**, **P** halts on input i, returning a value which is in accordance with the specification.

Testing and Correctness

- In order to determine correctness by testing it is necessary to be able to generate a finite set $\mathbf{T} \subset \mathbf{D}$ with the following properties:
  - For each $t \in \mathbf{T}$ there is a computable procedure for determining whether or not $\mathbf{P}$ terminates for t.
  - $\{\forall t \in \mathbf{T}, \mathbf{P}(t) = \mathbf{S}(t)\} \Rightarrow \{\forall t \in \mathbf{D}, \mathbf{P}(t) = \mathbf{S}(t)\}$

Verification vs. Validation

- Verification:
  "Are we building the product right?"
  - The software should conform to its specification.
- Validation:
  "Are we building the right product?"
  - The software should do what the user really requires.

Verification vs. Validation

- Verification:
  "Are we building the product right?"
  - The software should conform to its specification.
- Validation:
  "Are we building the right product?"
  - The software should do what the user really requires.

Verification and Validation

- **Validation**: *process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements*
  - Validation: Are we building the right product?

- **Verification**: *the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase*
  - Verification: Are we building the product right?

- IEEE definitions

The V & V Process

- Is a whole life-cycle process - V & V must be applied at each stage in the software process.
  - Example: Peer document reviews
- Has two principal objectives
  - The discovery of defects in a system
  - The assessment of whether or not the system is usable in an operational situation

V & V Goals

- Verification and validation should establish confidence that the software is fit for its purpose.
  - This does NOT mean completely free of defects.
  - Rather, it must be good enough for its intended use. The type of use will determine the degree of confidence that is needed.
- This leads us to definition of testing
  - Software testing is the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item

V & V Confidence

- Depends on the system's purpose, user expectations, and marketing environment
  - System purpose
    - The level of confidence depends on how critical the software is to an organization (e.g., safety critical).
  - User expectations
    - Users may have low expectations of certain kinds of software
  - Marketing environment
    - Getting a product to market early may be more important than finding defects in the program

Static vs. Dynamic V & V

- <u>Code and document inspections</u> - Concerned with the analysis of the static system representation to discover problems (*static v & v*)
  - May be supplement by tool-based document and code analysis
- <u>Software testing</u> - Concerned with exercising and observing product behaviour (*dynamic v & v*)
  - The system is executed with test data and its operational behaviour is observed

Static vs. Dynamic V & V

- Code and document inspections - Concerned with the analysis of the static system representation to discover problems (*static v & v*)
  - May be supplement by tool-based document and code analysis
- Software testing - Concerned with exercising and observing product behaviour (*dynamic v & v*)
  - The system is executed with test data and its operational behaviour is observed

What, when, how and how much…

- What (or what against what) to test?
- Is it the code, specification, features …
  - Answer: at different levels, we bear different perspectives
  - This could be code, model against requirements
  - What are requirements?
- When to test?
- How and how much?
  - We want to find out when to stop testing

Test Selection and Adequacy Criteria

- Attention has focused on two important areas:
  - *Test selection criteria* i.e., conditions that must be fulfilled by a test.

    For example, a criterion for a numerical program whose input domain is the integers might specify that each test contain one positive integer, one negative integer, and zero. { 3, 0, -7 }, { 122, 0, -11 }, { 1, 0, -1 } are three of the tests selected by this criterion.
  - *Test adequacy criteria* i.e., the properties of a program that must be exercised to constitute a thorough testing.

    For example, we consider that our software has been adequately tested if the test suite causes each method to be executed at least once. This provides a measurable objective for completeness.

What is adequacy?

- Given a program P written to meet a set of functional requirements R = {R1, R2, …, Rn}.
  Let T contain k tests for determining whether or not P meets all requirements in R.

- Assume that P produces correct behavior for all tests in T.
  We now ask:
  Has P been tested thoroughly? or: Is T adequate?

In software testing, the terms "thorough" and "adequate" have the same meaning.

Measurement of adequacy

- Adequacy is measured for a given test set and a given criterion.

- A test set is considered adequate wrt criterion C when it satisfies C.

Example

Program Sum must meet the following requirements:

R1 Input two integers, x and y, from standard input.

R2.1 Print to standard output the sum of x and y if $x<y$.

R2.2 Print to standard output the Sum of x and y if $x \geq y$.

Example (contd.)

Suppose now that the test adequacy criterion C is specified as:

C : A test set T for program (P, R) is considered adequate if, for each r in R there is a test case in T that tests the correctness of P with respect to r.

T={t: <x=2, y=3> is *inadequate* with respect to C for program Sum.

The lone test case t in T tests R1 and R2.1, but not R2.2.

Black-box and white-box criteria

Given an adequacy criterion C, we derive a finite set Ce known as the coverage domain.
A criterion C is a white-box test adequacy criterion if the corresponding Ce depends solely on the program P under test.

A criterion C is a black-box test adequacy criterion if the corresponding Ce depends solely on the requirements R for the program P under test.

Coverage

Measuring adequacy of T:

T covers Ce if, for each e' in Ce, there is a test case in T that tests e'. T is adequate wrt C if it covers all elements of Ce.

T is inadequate with respect to C if it covers k<n elements of Ce.

k/n is the coverage of T wrt C.

Example

Consider criterion:
"A test T for (P, R) is adequate if , for each requirement r in R, there is at least one test case in T that tests the correctness of P with respect to r."

The coverage domain is Ce={R1, R2.1, R2.2}.
T covers R1 and R2.1 but not R2.2.
Hence, T is inadequate with respect to C.
The coverage of T wrt C is 2/3=0.66.

Another Example

Consider the following criterion:
"A test T for program (P, R) is considered adequate if each path in P is traversed at least once."

Assume that P has exactly two paths, p1 and p2, corresponding to condition $x<y$ and condition $x \geq y$, respectively.

For the given adequacy criterion C we obtain the coverage domain Ce to be the set {p1, p2}.

Another Example (contd.)

To measure the adequacy of T of Sum against C,
we execute P against each test case in T .

As T contains only one test for which x<y, only path p1 is executed.

Thus, the coverage of T wrt C is 0.5. T is not adequate with respect to C.

We can also say that p1 is tested and p2 is not tested.

Code-based coverage domain

In the previous example, we assumed that P contains two paths. When the coverage domain must contain elements from the code, these elements must be derived by analyzing the code.

Errors in the program and incomplete/incorrect requirements can cause the coverage domain to be different from the expected. Question: What if we have 20 paths….

However

An adequate test set might not reveal even the most obvious error in a program.

This does not diminish in any way the need for the measurement of test adequacy, as increasing coverage might reveal an error!

Program Representation

Graph based representation

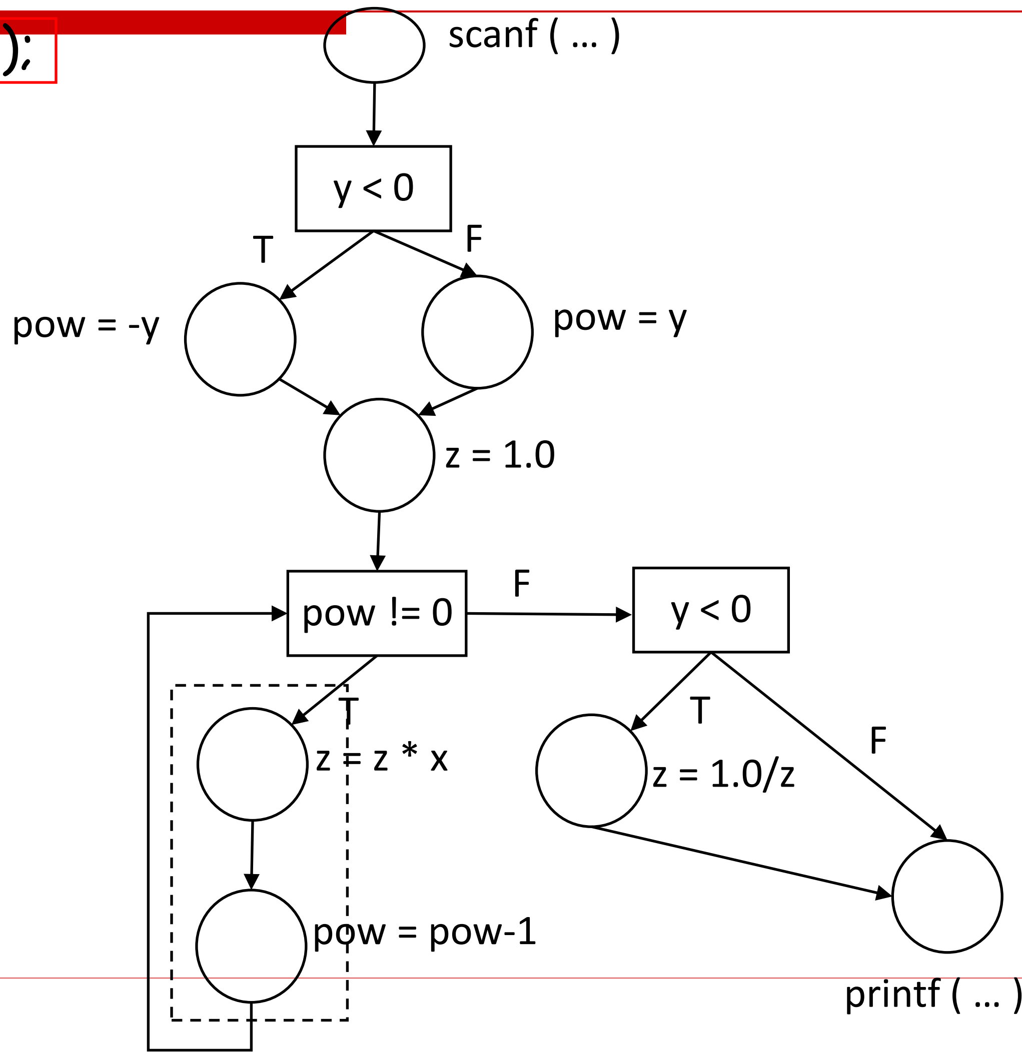
- We can represent a program as a set of nodes connected with vertices
- We can define graph coverage criterion
- We can based our sufficiency measurement or adequacy of testing
- Program information
  - Control-flow
  - Data-flow
  - Dependence
  - ...

Control Flow Graphs

- A Control Flow Graph (CFG) is a *static*, abstract representation of a program.
- A CFG is a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$
  - Each node, in the set $\mathcal{N}$, is either a *statement node* or a *predicate node*.
  - A *statement node* represents a simple statement. Alternatively, a statement node can be used to represent a basic block.
  - A *predicate node* represents a conditional statement.
  - Each *edge*, in the set $\mathcal{E}$, represents the flow of control between statements.
  - Optionally, we use circles to represent statement nodes, and rectangles to represent predicate nodes.

Example of a CFG

```
scanf ("%d, %d", &x, &y);
if (y < 0)
        pow = -y;
else
        pow = y;
z = 1.0;
while (pow != 0) {
        z = z * x;
        pow = pow - 1;
}
if (y < 0)
        z = 1.0 / z ;
printf("%f", z);
```



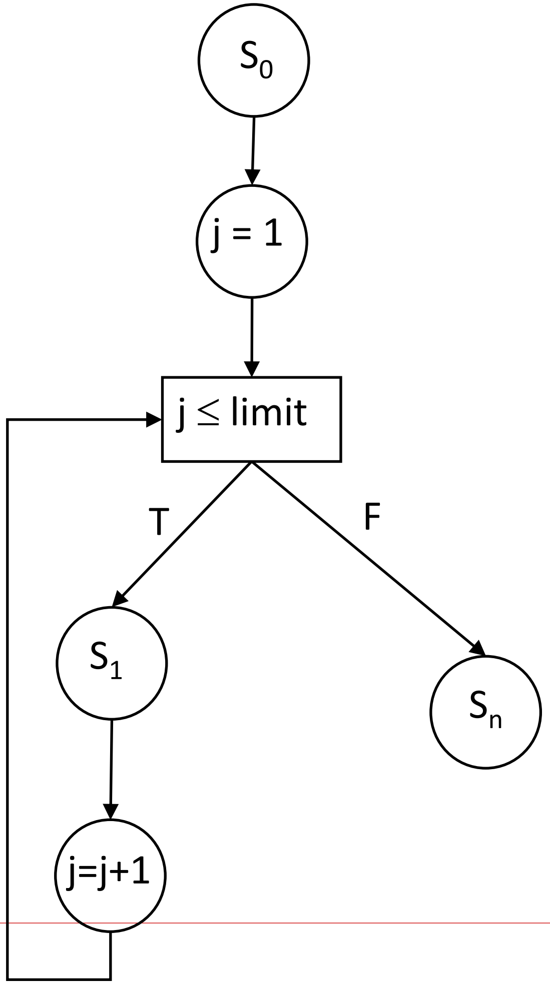
CFG: The **for** Loop

What does the CFG for the following
code fragment look like?

```
S0;
for ( j = 1; j <= limit; j=j+1 )
{
    S1;
}
Sn;
```

CFG: The **for** Loop

```
S0;
for ( j = 1; j <= limit; j=j+1 )
{
      S1;
}
Sn;
```



CFGs: Switch-Case

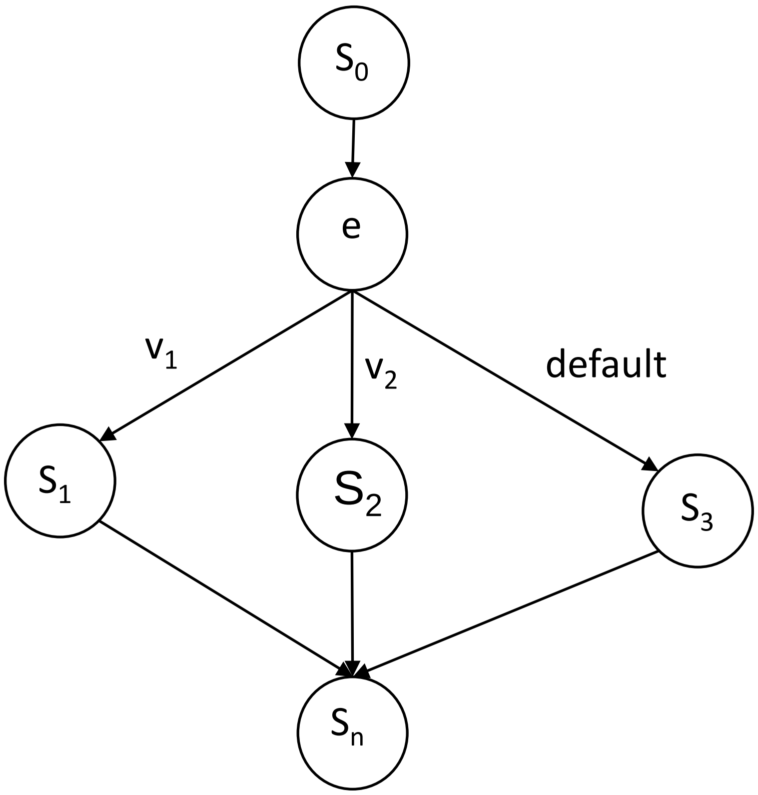
What does the CFG for the following code fragment look like?

```
S0;
switch ( e )
{
    case v1:
        S1;
        break;
    case v2:
        S2;
        break;
    default:
        S3;
}
Sn;
```

CFGs: Switch-Case

Intuitively, the diagram shown on the right...

However, *e* is not a predicate.



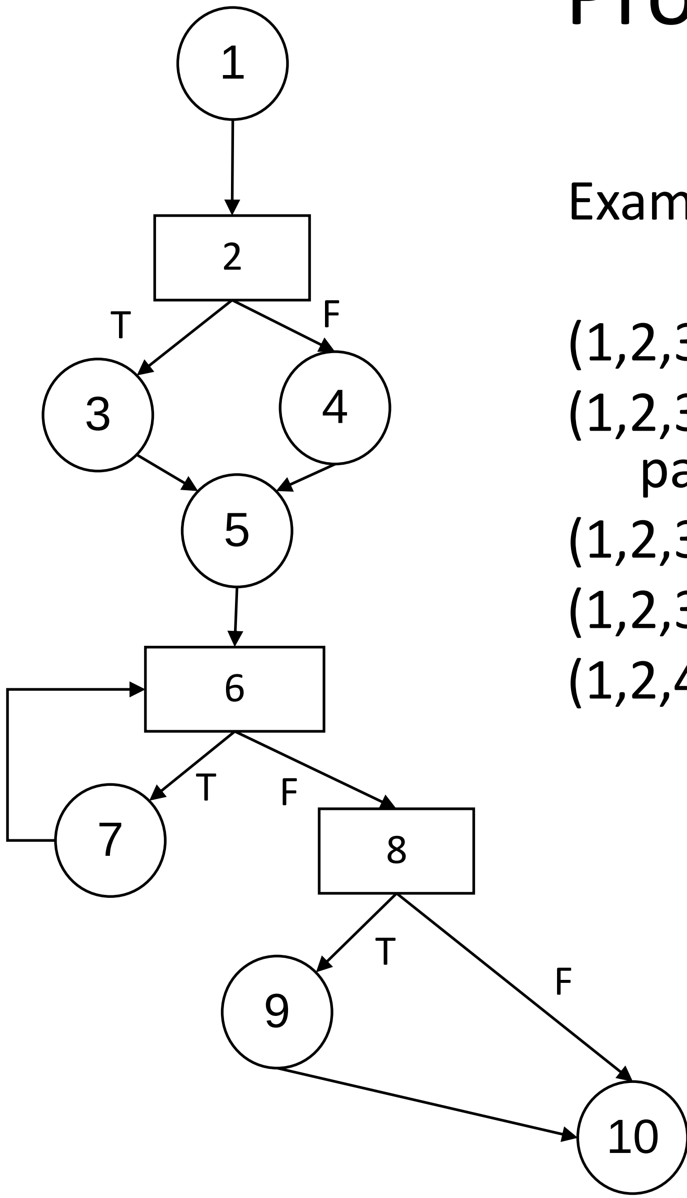
Program Paths

- A *path* is a unique sequence of executable statements from one point of the program to another point.
- In a graph, a path is a sequence $(n_1, n_2, \ldots, n_t)$ of nodes such that $\langle n_i, n_{i+1} \rangle$ is an edge in the graph, $\forall$ i=1, 2,.., t-1 (t > 0).

Program Paths

- *Complete path* starts with the first node and ends at the last node of the graph.
- *Execution path* a complete path that can be exercised by some input data.
- *Subpath* a subsequence of the sequence, e.g.,
  - $n_1, n_2, \ldots, n_t$
- *Elementary path* all nodes are unique.
- *Simple path* all edges are unique

Program Paths



Example:

(1,2,3,5,6,7,6,8,10) is a *complete* path

(1,2,3,5,6,7,6,7,6,8,10) is a different *complete* path

(1,2,3,5) is a *subpath*

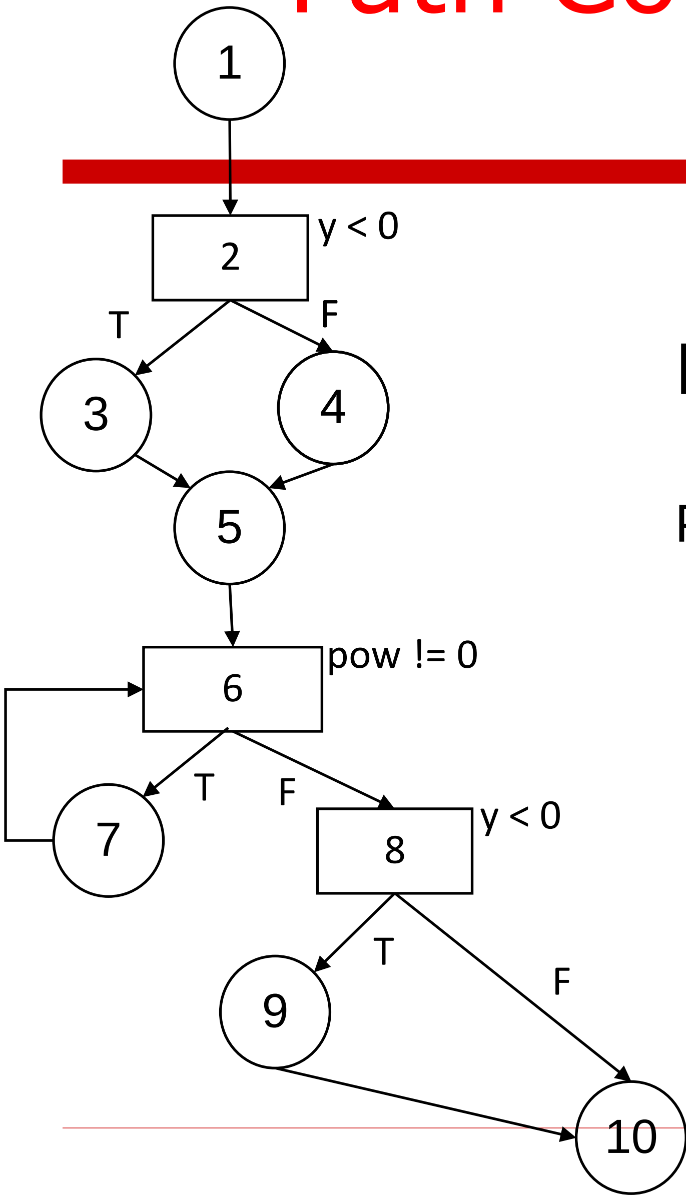
(1,2,3,5,6,8,10) is an *elementary* path

(1,2,4,5,6,7,6,8,10) is a *simple* path

Path Condition

- *Path condition*: the conjunction of the individual predicate conditions that are generated at each branch point along the path.
- The path condition must be satisfied by the input data in order for the path to be executed.

Path Condition



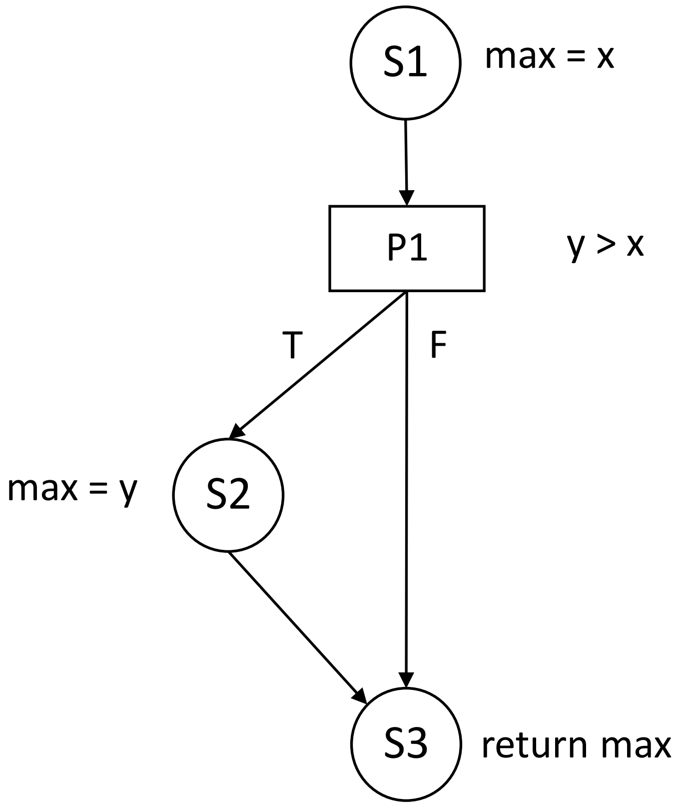
Example:

PC(1,2,4,5,6,8,10) = ($y \geq 0 \wedge pow == 0 \wedge y \geq 0$)

Number of Paths

The simple CFG shown here has 2 different paths:

(S1, P1, S2, S3)
(S1, P1, S3)

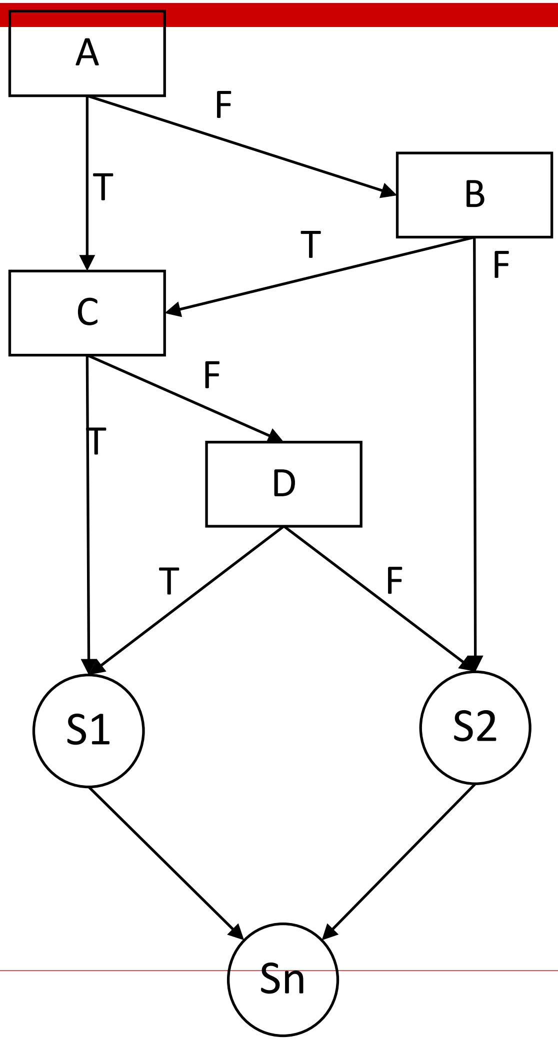


Number of Paths

How many paths?

```
if ((A||B) && (C||D)
{
    S1;
}
else
{
    S2;
}
```

Number of Paths



There are:

  2 statements + Sn

  4 branches

7 paths (4T + 3 F)

(A, C, S1)
(A, C, D, S1)
(A, C, D, S2)
(A, B, C, S1)
(A, B, C, D, S1)
(A, B, C, D, S2)
(A, B, S2)

Infinite Paths

- Since the value for the variable a is entered by the user, we consider this program to have an infinite number of paths.
- In general, programs that contain loops with control variables whose value is supplied by the user have infinite number of paths.

Infeasible Paths

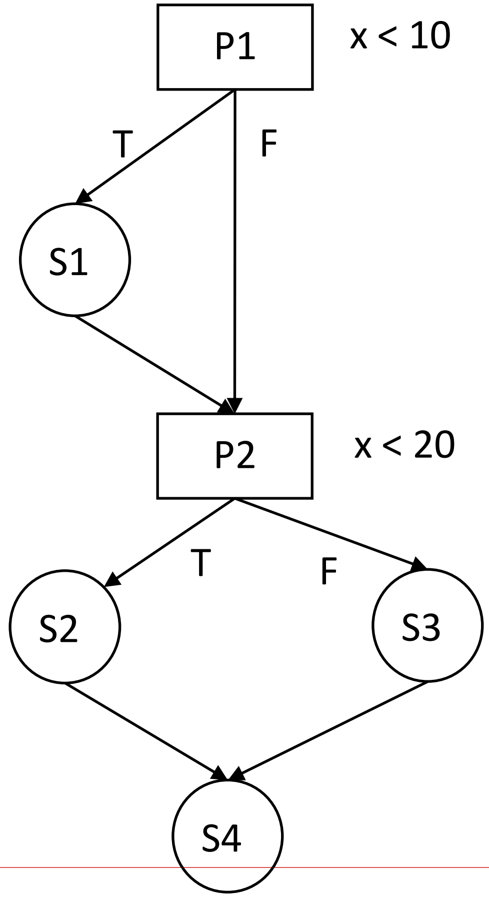
- A path is said to be feasible if it can be exercised by some input data; otherwise the path is said to be infeasible.
- Infeasible paths are the result of contradictory predicates.

Infeasible Paths

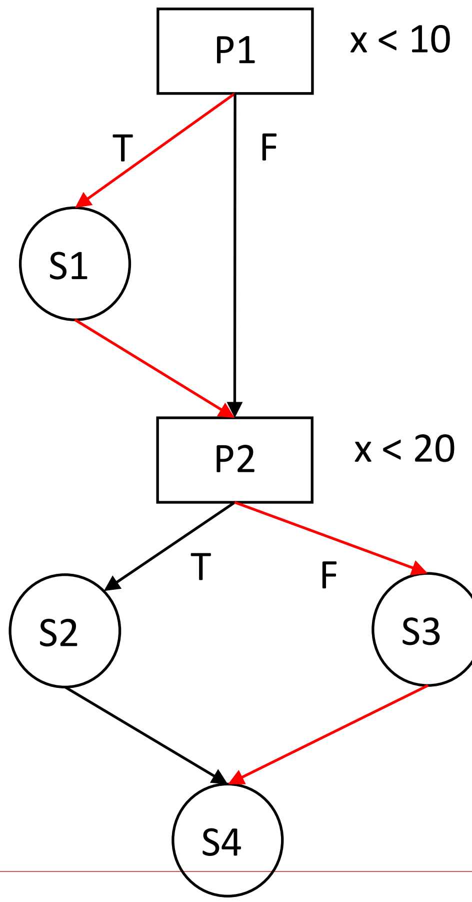
Are there any

infeasible paths?

How many?



Infeasible Paths



The path

(P1, S1, P2, S3, S4)

is not feasible.

Unless S1 changes the value of x …

Flow graphs Consist of Three Primitives

- A **decision** is a program point at which the control can diverge.
  - (*e.g.,* if and *case* statements).
- A **junction** is a program point where the control flow can merge.
  - (*e.g., end if, end loop, goto label*)
- A **process block** is a sequence of program statements uninterrupted by either decisions or junctions. (*i.e.,* straight-line code).
  - A process has one entry and one exit.
  - A program does not jump into or out of a process.

Path Selection Criteria

- There are many paths between the entry and exit points of a typical routine.
- Even a small routine can have a large number of paths.
- Examples
  - Exercise every path from entry to exit.
  - Exercise every statement at least once.
  - Exercise every branch (in each direction) at least once.
- Clearly, 1 implies 2 and 3
- However, 1 is impractical for most routines.
- Also, 2 is not equal to 3 in languages with `goto` statements.

Effectiveness of Control-flow Testing

- Studies show that control-flow testing catches 50% of all bugs caught during unit testing.
  - About 33% of all bugs.
- Control-flow testing is more effective for unstructured code than for code that follows structured programming.
- Experienced programmers can bypass drawing flowgraphs by doing path selection on the source.

Limitations of Control-flow Testing

- Control-flow testing as a sole testing technique is limited:
  - Interface mismatches and mistakes are not caught.
  - Not all initialization mistakes are caught by control-flow testing.
  - Specification mistakes are not caught.

Path Predicates

- Every path corresponds to a succession of **true** or **false** values for the predicates traversed on that path.
- A **Path Predicate Expression** is a Boolean expression that characterizes the set of input values that will cause a path to be traversed.
- **Multiway branches** (*e.g., case/switch* statements) are treated as equivalent *if then else* statements.

Path Predicate Expressions

- Any set of input values that satisfies ALL of the conditions of the path predicate expression will force the routine through that path.
- If there is no such set of inputs, the path is not achievable.
- Process of creating path expression:
  - Write down the predicates for the decisions you meet along a path.
  - The result is a set of path predicate expressions.
  - All of these expressions must be satisfied to achieve a selected path.

Process (In)dependent and correlated Predicates

- Process independent predicates
  - A predicate whose truth value cannot/can change as a result of the processing is said to be **Process Independent/Dependent**, respectively.
  - If all the variables on which a predicate is based are process independent, the predicate must be process independent.
- Correlated predicates
  - A pair of predicates whose outcomes depend on one or more variables in common are said to be **Correlated Predicates.**
  - Every path through a routine is achievable only if all predicates in that routine are uncorrelated.

Path Sensitization

- The act of finding a set of solutions to the path predicate expression is called **path sensitization**.
- This yields set of values when given as input allow us to traverse that path
- We manually compute expected outputs
- Set of inputs together with expected outputs, they form test cases

Test Outcomes

- The **outcome** of test is what we expect to happen as a result of the test.
- Test outcomes include anything we can observe in the computer's memory that should have (not) changed as a result of the test.
- Since we are not "kiddie testing" we must predict the outcome of the test as part of the test design process.

Testing Process

- run the test
- observe the actual outcome
- compare the actual outcome to the expected outcome.

Summary

- Control flow graphs
- Path selection criteria
- Input vector
- Predicates
- Path sensitization
- Test outcomes and process