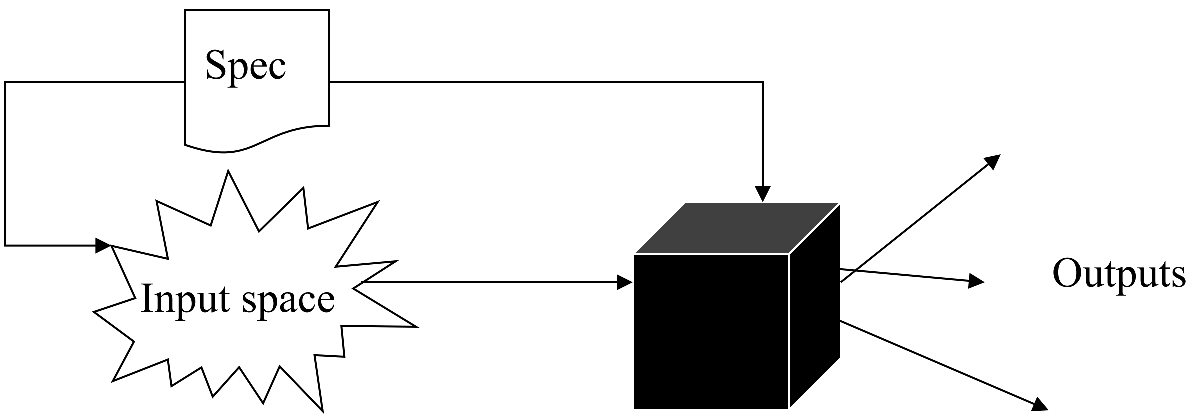
# Software Verification and Validation

## Functional Testing

# Functional Testing

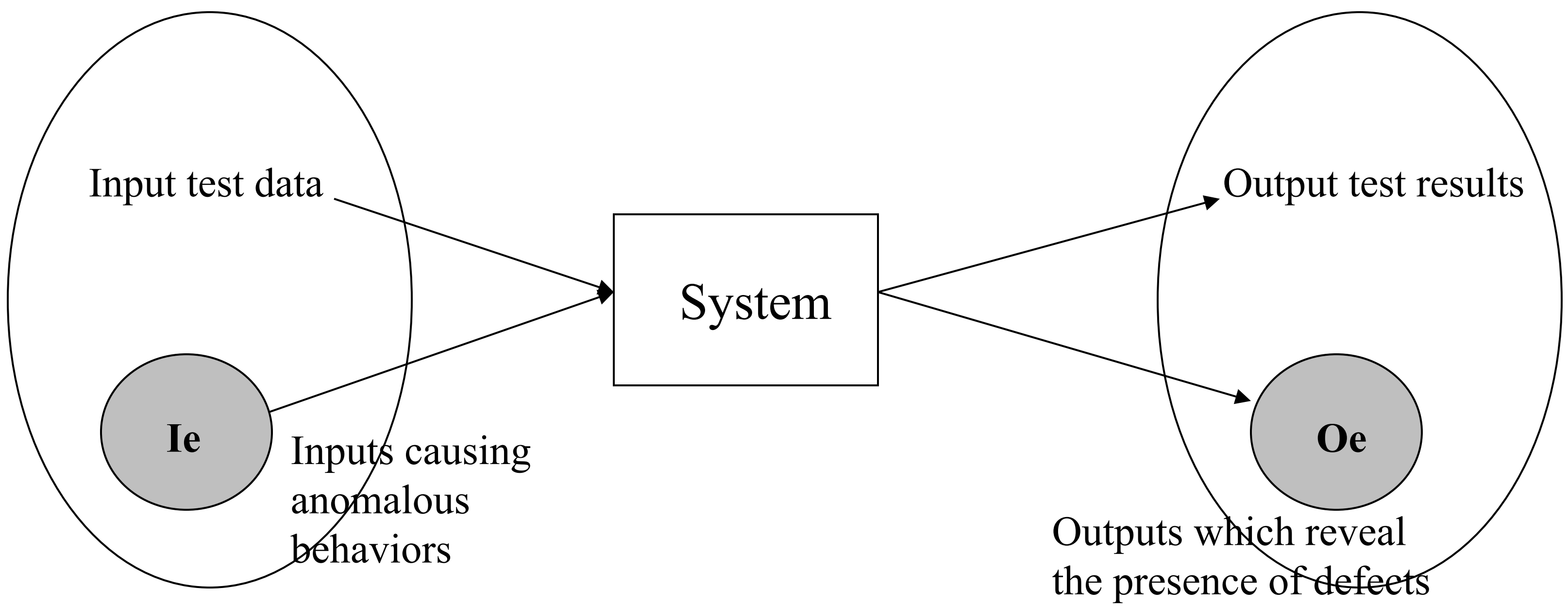
- Focuses on the functional requirements (i.e., specification) of software
- Also known as:
  - Black box testing, specification-based testing, behavioral testing
- Attempts to find errors in the following categories:
  1. Incorrect or missing functions
  2. Interface errors
  3. Errors in data structures or external data based access
  4. Behavior of performance errors
  5. Initialization and termination errors



# Functional Testing

- Test cases are designed to answer the following questions:
  - How is functional validity tested?
  - How is system behavior and performance tested?
  - What classes of input will make good test cases?
  - How are the boundaries of a data class isolated

# Functional (Black Box) Testing

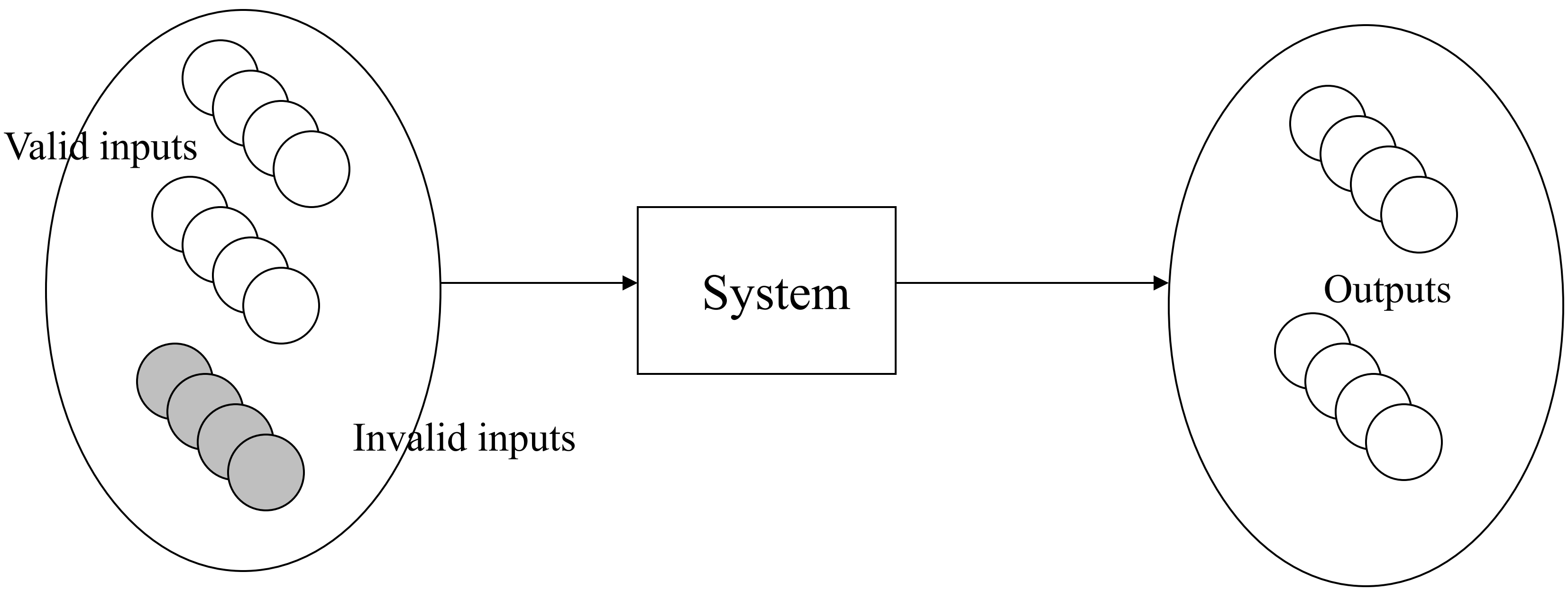


# Major Methods for Generating Functional Test Cases

- Equivalence Partitioning (EP)
- Boundary Value Analysis (BVA)
- Error Guessing (EG)
- Pair-wise combination testing
- Graph-based testing methods
- Syntax Checking

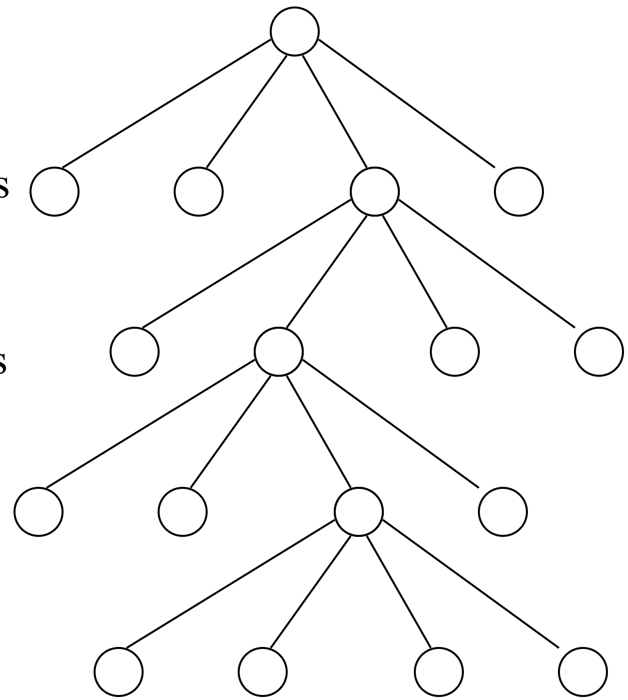
# Equivalence Partitioning (EP)

- Partition input data into a number of classes
  - E.g., positive numbers, negative numbers, string without bank, etc.
- Basic idea – Find EP of values for input and output variables
- Once identified a set of partitions, choose test cases from each of these partitions

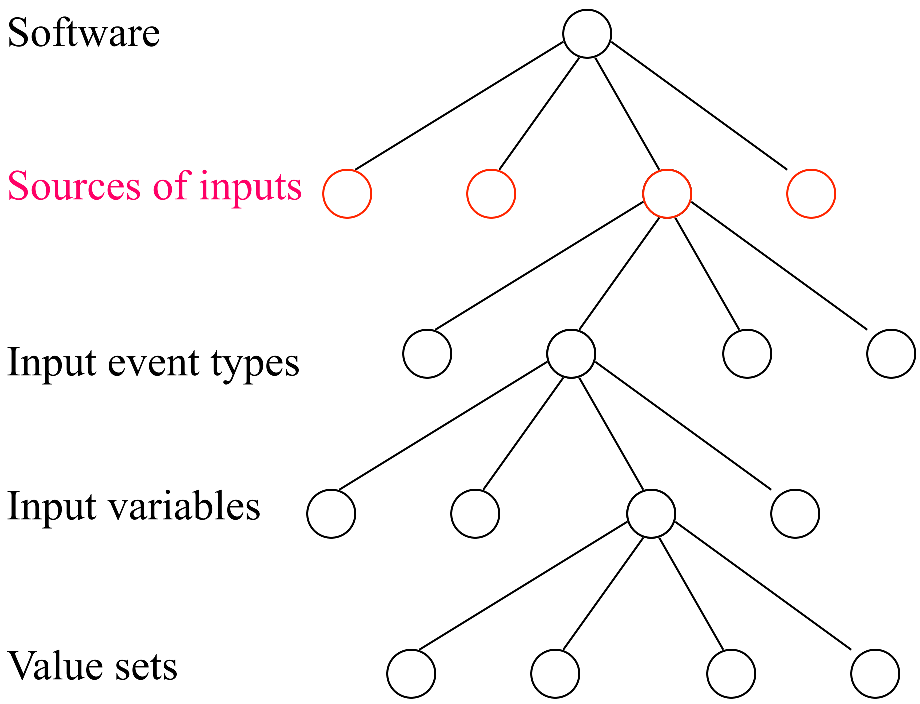


# Equivalence Partitioning (EP) on Input and Outputs Spaces

| ***Input*** | ***Output*** |
| --- | --- |
| Software | Software |
| Sources of inputs | Output Destinations |
| Input event types | Output event types |
| Input variables | Outputs variables |
| Value sets | Value sets |

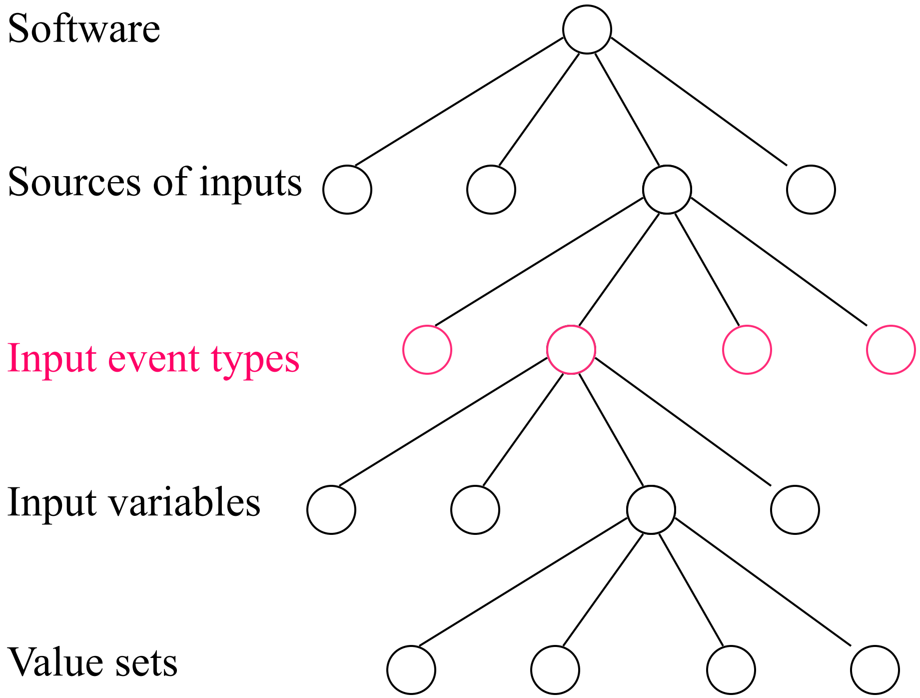


# Identify Source of Inputs



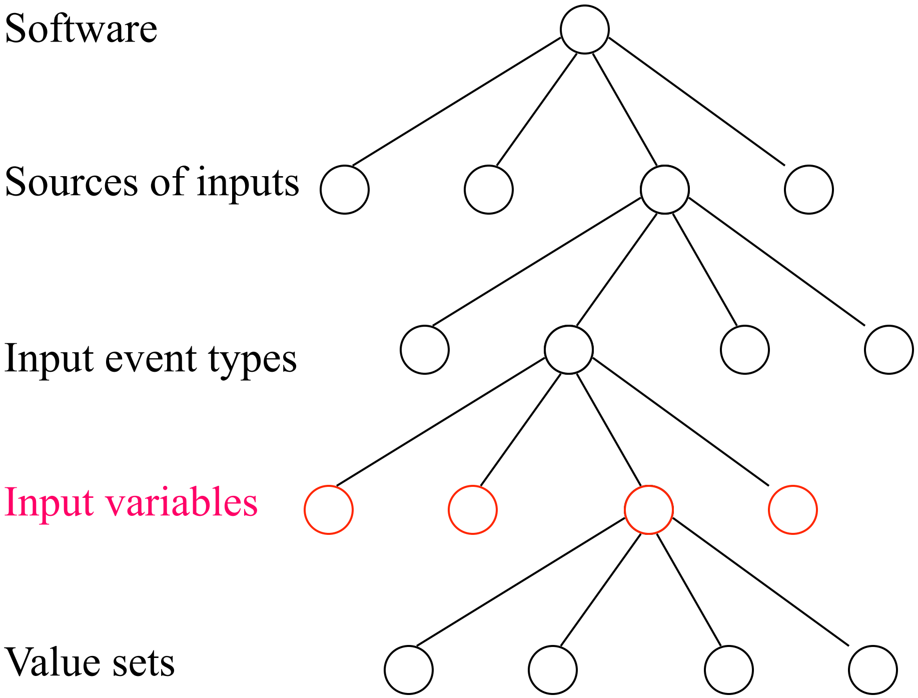

- Anything that is external to the Software Under Test (SUT)
- Examples:
  - Data typed by the end user
  - GUI interface
  - Disk files
  - Data coming through the network
  - Etc.

# Identify Input Event Types



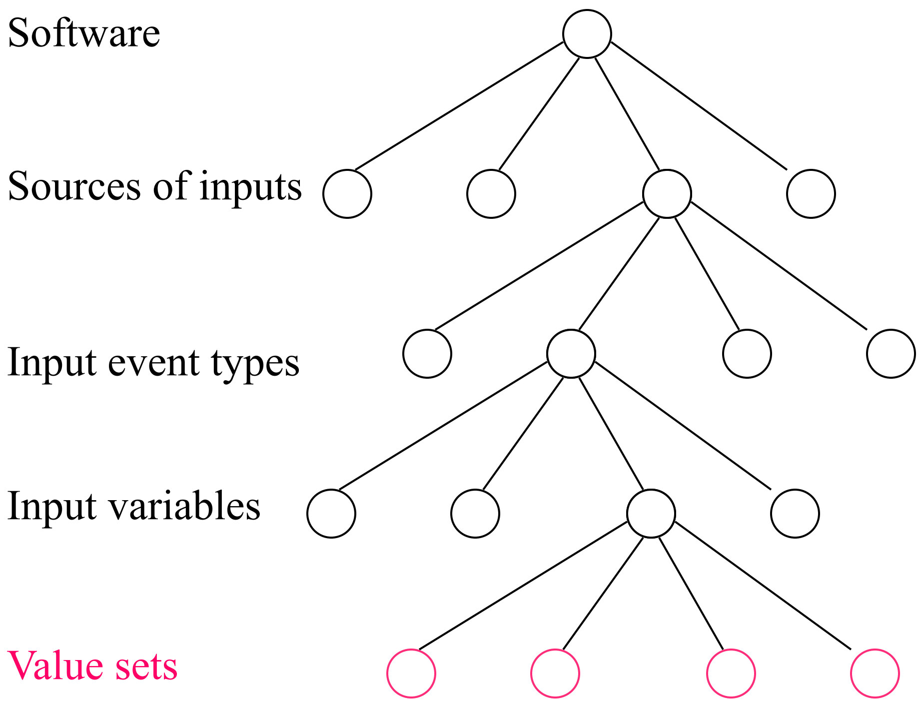

- Examples:
  - Mouse movement
  - Interrupt
  - Pop up menu
  - Completion of a GUI interface
  - Etc.

# Identify Input Variables




- Examples:
  - Any scalar or string variables such as phone number, driver's license number, name
  - Some inputs might not have any associated variables (e.g., menu selection)

# Identify Value Sets




- The stage where actually defining classes are taking place
- Input variables can take values as:
  - Valid – System proceeds normally
  - Invalid – System gives error message (e.g., 25 as the month number)
- Break up the possible values into valid and invalid
- Choose one or more items of the valid class
- Choose one or more items of the invalid class

# Value Sets – Some Examples

- Height of an object
  - Valid: {x|x > 0}, Invalid: {x| x <= 0}, Invalid: {x| x is not a number}
- Name of a city
  - Valid: {x| x is alphabetic}, Invalid: {x|x is alphanumeric}, Invalid: {x| everything else}
- File name argument to Unix *lpr* command
  - Valid: {x| an existing file}, Invalid: {x| everything else}
- Usually number of invalid sets are greater than number of valid sets

# How to Construct Test Cases Using EP

- If an input condition specifies a *range*, one valid and two invalid equivalence classes are defined
- If an input condition requires a specific *value*, one valid and two invalid equivalence classes are defined
- If an input condition specifies *a member of a set*, one valid and one invalid equivalence class are defined
- If an input condition is *Boolean*, one valid and one invalid class are defined

# How to Construct Test Cases Using EP

- For each input event type, at least one test case that includes that event type (e.g., pop-up menu, mouse right click, etc.)
- For each valid data set, at least one test case that include a value from that set and all the values are valid
  - E.g. (MM/DD/YY) (03/12/2009)
- For each invalid data set, at least one test case that include a value from that set and no other values are invalid
  - E.g. (MM/DD/YY) (25/12/2009)
- For each invalid set of combination of valid values, at least one test case that includes a combination from that set
  - E.g. (MM/DD/YY) (09/31/2009)

# Another Example Using EP

- Two obvious equivalence partition (from specification)
  - A. Inputs where the key element is a member of the sequence (FOUND=true)
  - B. Inputs where the key element is not a sequence member (FOUND=false)
- Some Testing Guidelines
  - Test software with sequences that have only a single value
  - Use different sequences of different sizes
  - Derive tests so that first, middle, and last elements of the sequence are accessed
- Therefore, additional partitions:
  - C. The input sequence has a single value
  - D. The number of elements in the input sequence is greater than 1 (more than one partition)

The specification of a search routine:

**procedure** Search (key: ELEM;
T: SEQ of ELEM;
Found: **in out** BOOLEAN;
L: **in out** ELEM_INDEX);

**Pre-condition**
--the sequence has at least one element
T^FIRST <= T^LAST

**Post-condition**
--the element is found and is referenced by L
(Found and T (L) = Key)

**Or**
--the element is not in the sequence
(**not** Found **and**
**not** (**exists** i; T^FIRST >= i <= T^LAST, T(i) = key))

# Another Example Using EP

- Therefore, equivalence partitions for search routine

| **Array** | **Element** |
|---|---|
| Single value | In sequence |
| Single value | Not in sequence |
| More than 1 value | First element in sequence |
| More than 1 value | Last element in sequence |
| More than 1 value | Middle element in sequence |
| More than 1 value | Not in sequence |

The specification of a search routine:

**procedure** Search (key: ELEM;
T: SEQ of ELEM;
Found: **in out** BOOLEAN;
L: **in out** ELEM_INDEX);

**Pre-condition**

--the sequence has at least one element
T^FIRST <= T^LAST

**Post-condition**

--the element is found and is referenced by L
(Found and T (L) = Key)

**Or**

--the element is not in the sequence
(**not** Found **and**
**not** (**exists** i; T^FIRST >= i <= T^LAST, T(i) = key))

# Another Example Using EP

- Therefore, possible test cases are

| **Input sequence(T)** | **Key (Key)** | **Output (FOUND,L)** |
|---|---|---|
| 17 | 17 | True, 1 |
| 17 | 0 | False, ?? |
| 17, 29, 21, 23 | 17 | True, 1 |
| 41, 18, 9, 31, 30, 16, 45 | 45 | True, 7 |
| 17, 18, 21, 23, 29, 41, 38 | 23 | True, 4 |
| 21, 23, 29, 33, 38 | 25 | False, ?? |

The specification of a search routine:

**procedure** Search (key: ELEM;
T: SEQ of ELEM;
Found: **in out** BOOLEAN;
L: **in out** ELEM_INDEX);

**Pre-condition**

--the sequence has at least one element
T^FIRST <= T^LAST

**Post-condition**

--the element is found and is referenced by L
(Found and T (L) = Key)

**Or**

--the element is not in the sequence
(**not** Found **and**
**not** (**exists** i; T^FIRST >= i <= T^LAST, T(i) = key))

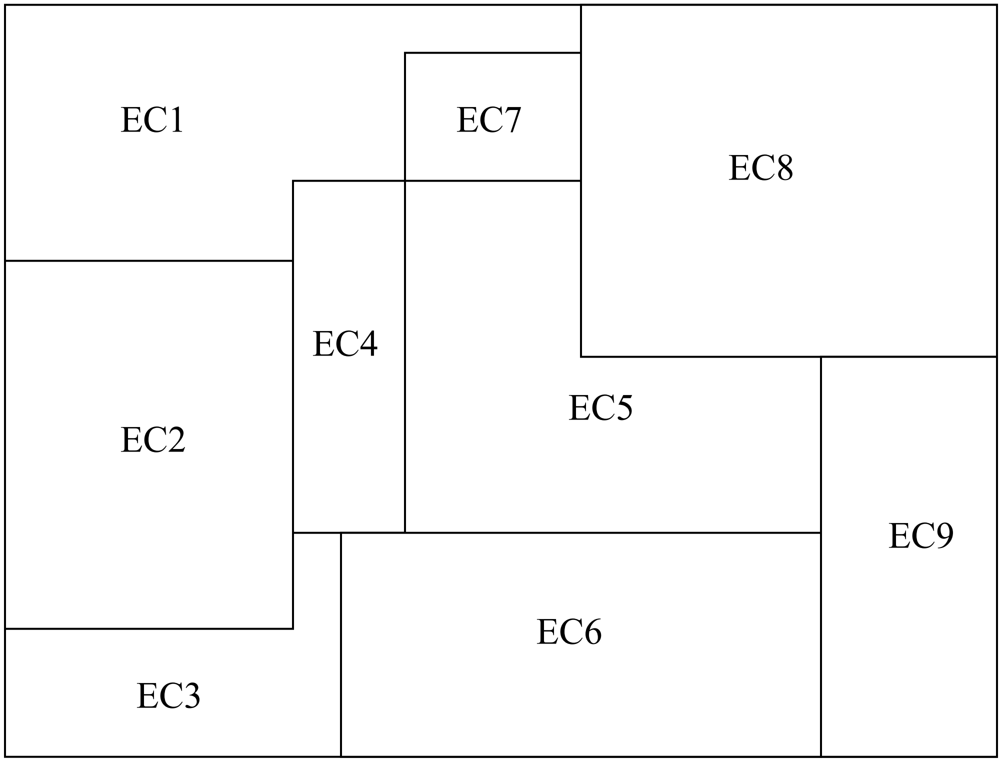
# How About Outputs

- Similar to inputs
- Output destinations
  - E.g., paper, GUI, data files, network
- Output event types
  - E.g., message to users, data sent to network
- Output even variables
  - E.g., any data returned can be a variable
- Value sets
  - A and B are in the same value set, if we guess that the result of processes will be approximately the same
- Output values
  - Not “valid” or “invalid”
  - Only values

# An Example Using EP

- Online banking application
  - Password - A six-digit password
  - Area code – Blank or three-digit number
  - Prefix – Three-digit number not beginning with 0 or 1
  - Suffix - Four-digit number
- Input variables and conditions
  - Area code
    - Boolean, the area code may or may not be present
    - Range, values defined between 200 and 999 with specific exception
  - Prefix
    - Range, specified value > 200
    - Value, four-digit length
  - Password
    - Boolean, may or may not be present
    - Value, six character length

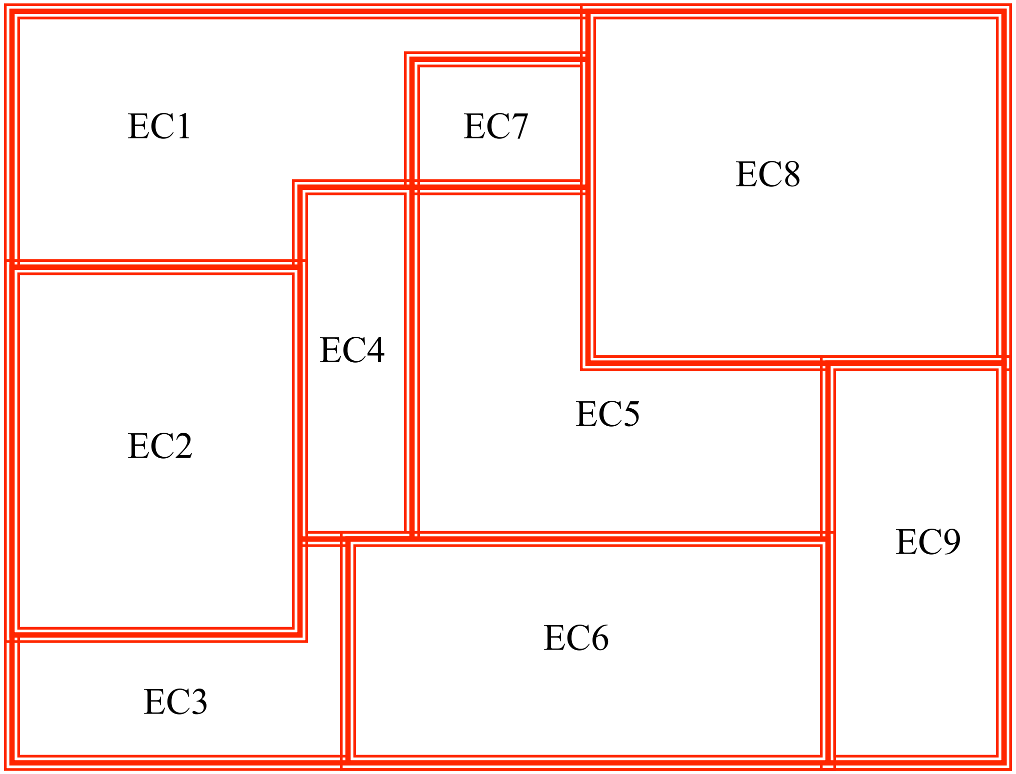
# Boundary Value Analysis (BVA)




Input Domain Space

# Boundary Value Analysis (BVA)

BVA extends equivalence partitioning by focusing on data at the edges of a class




Input Domain Space

# Boundary Value Analysis (BVA)

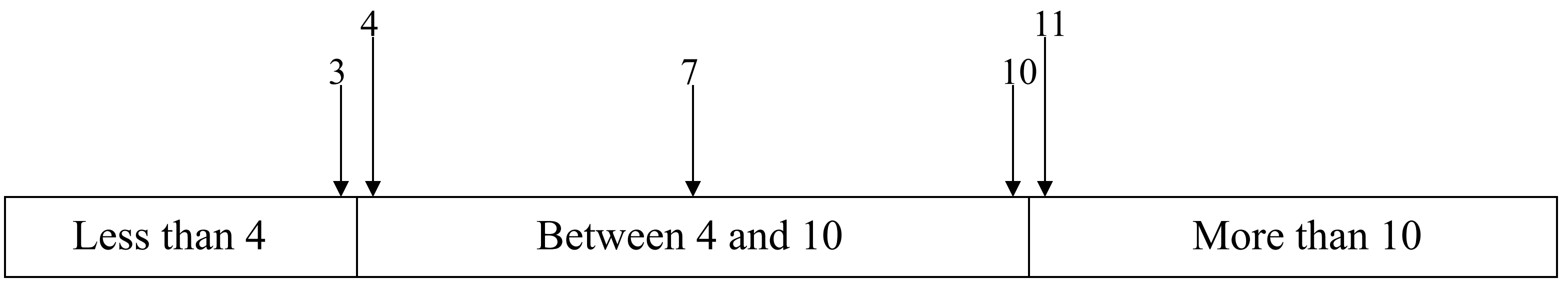
- Suppose that x1, x2, …, xn are numeric variables of an input even
- Some possible problems:
  - Boundary shift: where is the boundary between ECs
  - Missing boundary: Does the software implements the boundary?
  - Extra boundary: Do we have any hidden or extra boundary not implemented by the software
  - Closure problem: To which EC the boundary points belong to?
- Basic Technique
  - Choose values close to boundaries
- Simple example (MM/DD)
  - (0, 21), (1, 21), (12, 21), (13, 21), (7, 0), (7, 1), (7, 31), (7, 32)
- Extreme Value Analysis (EVA)
  - Select “extreme” values for input/output
    - E.g., very large or very small numbers

# Boundary Value Analysis (BVA) – A Guideline

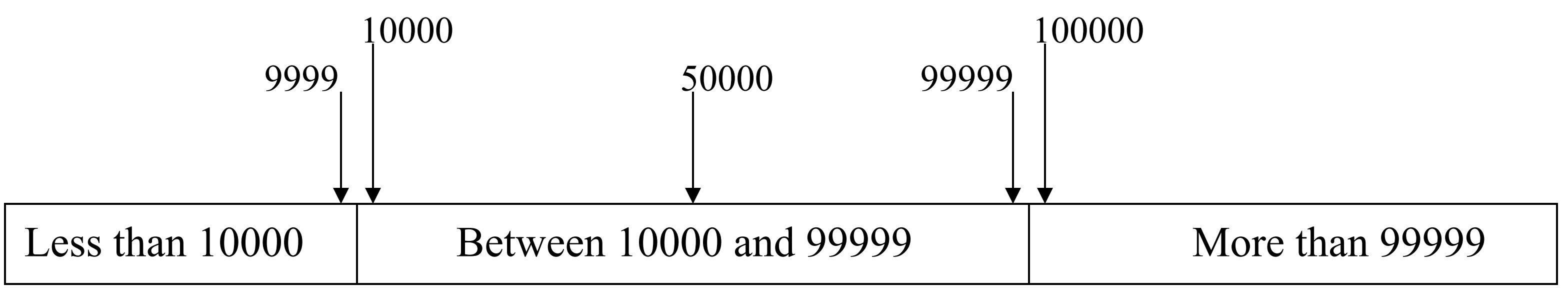
1. If an input condition specifies a range bounded by values a and b, design test cases that exercise a, b, and just above and below of these bounds
2. If an input condition specifies a number of values, design test cases that exercise the min and max numbers as well as values above and below the min and max
3. Apply the above guidelines to the output space
4. Apply the above guidelines to any data structures used in the intenal logic of program (e.g., an array SEQ defined as SEQ[100])

# Boundary Value Analysis (BVA) – Another Example

- A program spec states that:
  - The program accepts 4 to 10 inputs which are five-digit integers greater than 10000. Then the boundary values could be:




**Number of inputs values**




**Inputs values**

# Error Guessing (EG)

- Needs imagination and thinking about how to break a system
- Having previous experience is an asset
  - E.g., working with link list
    - Add node
    - Delete first node and add a node as a first one
    - Delete last node and add a last node
    - Delete all nodes and then add them again
  - Unusual characters
  - Negative numbers
  - Big numbers
- Particular useful in testing GUIs
  - Trying to type a numeric value into name field

# Pair-wise combination testing

- A specification-based testing criterion
- Problem with exhaustive enumeration
  - An exhaustive coverage of all pairs between parameters and their values might be very expensive

| **Parameter A** | **Parameter B** | **Parameter C** |
|---|---|---|
| A1 | B1 | C1 |
| A2 | B2 | C2 |
| A3 | B3 | C3 |

Exhaustive enumeration: Number of test cases required is 3 * 3 * 3 = 27

# Pair-wise combination testing – Horizontal and Vertical Grow

- Idea – Generating test cases more intelligently
- Also called: In-Parameter Order (IPO)
  - Only nine test cases needed

| Parameter A | Parameter B | Parameter C |
|---|---|---|
| A1 | B1 | C1 |
| A2 | B2 | C2 |
| A3 | B3 | C3 |

| A | B | C |
|---|---|---|
| A1 | B1 | C1 |
| A1 | B2 | C2 |
| A1 | B3 | C3 |
| A2 | B1 | C2 |
| A2 | B2 | C3 |
| A2 | B3 | C1 |
| A3 | B1 | C3 |
| A3 | B2 | C1 |
| A3 | B3 | C2 |

# Pair-wise combination testing – Horizontal Grow Algorithm

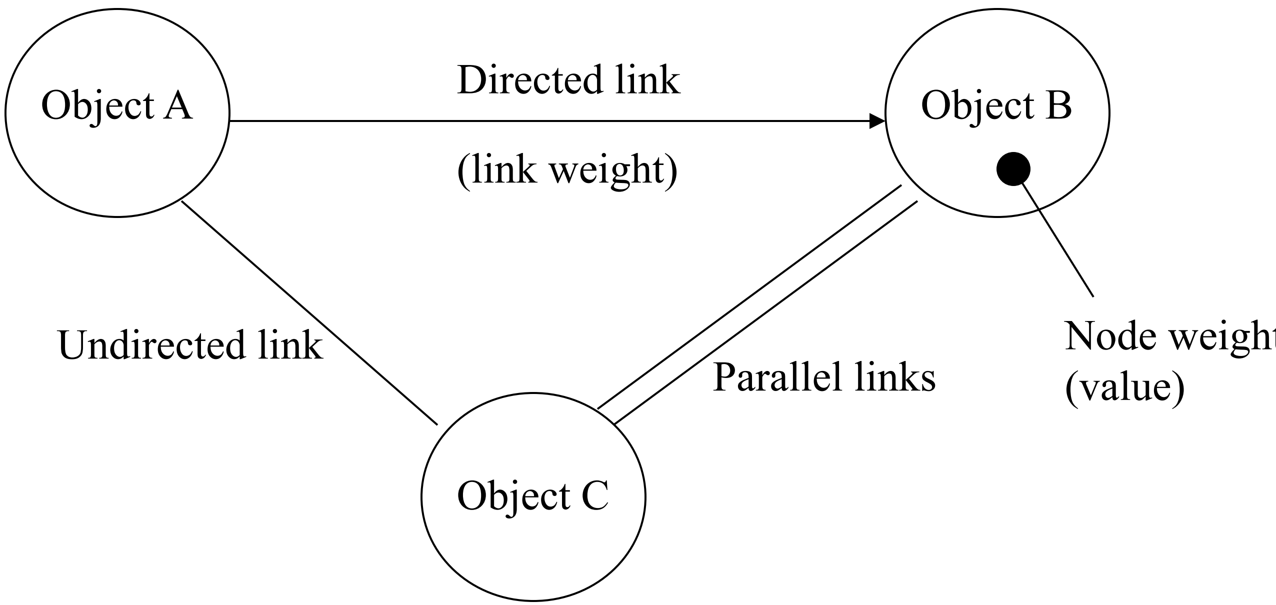
**Algorithm** $IPO\_H(\mathcal{T}, p_i)$
// $\mathcal{T}$ is a test set. But $\mathcal{T}$ is also treated as a list with elements in arbitrary order
{ assume that the domain of $p_i$ contains values $v_1, v_2, \ldots,$ and $v_q$;
  $\pi = \{$ pairs between values of $p_i$ and values of $p_1, p_2, \ldots,$ and $p_{i-1}$ $\}$;
  **if** $(|\mathcal{T}| \leq q)$
  { for $1 \leq j \leq |\mathcal{T}|$, extend the $j$th test in $\mathcal{T}$ by adding value $v_j$ and
    remove from $\pi$ pairs covered by the extended test;
  }
  else
  { for $1 \leq j \leq q$, extend the $j$th test in $\mathcal{T}$ by adding value $v_j$ and
    remove from $\pi$ pairs covered by the extended test;
    for $q < j \leq |\mathcal{T}|$, extend the $j$th test in $\mathcal{T}$ by adding one value of $p_i$
    such that the resulting test covers the most number of pairs in $\pi$, and
    remove from $\pi$ pairs covered by the extended test;
  }
}

# Pair-wise combination testing – Vertical Grow Algorithm

**Algorithm** $IPO\_V(\mathcal{T}, \pi)$
{ let $\mathcal{T}'$ be an empty set;
  **for** each pair in $\pi$
  { assume that the pair contains value $w$ of $p_k$, $1 \leq k < i$, and value $u$ of $p_i$;
    **if** ($\mathcal{T}'$ contains a test with "$-$" as the value of $p_k$ and $u$ as the value of $p_i$)
      modify this test by replacing the "$-$" with $w$;
    **else**
      add a new test to $\mathcal{T}'$ that has $w$ as the value of $p_k$, $u$ as the value of $p_i$,
      and "$-$" as the value of every other parameter;
  };
  $\mathcal{T} = \mathcal{T} \cup \mathcal{T}'$;
};

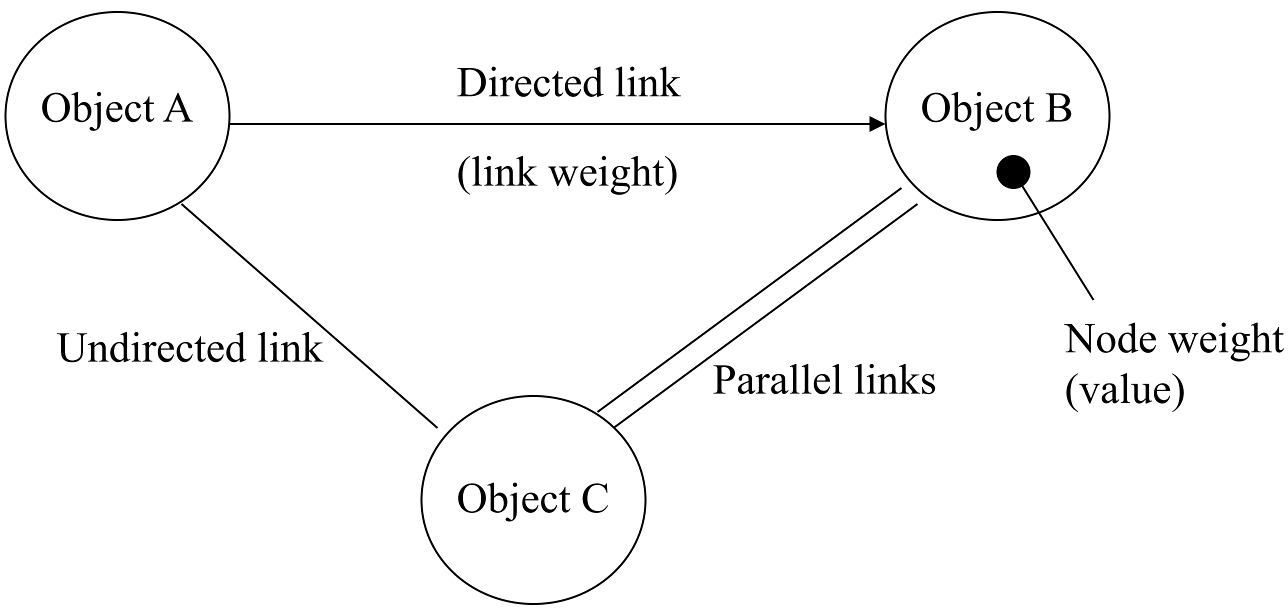
# Graph-based Testing Methods

- A graph represents the relationship between data objects and program objects, enabling us to derive test cases that search for errors associated with these relationships



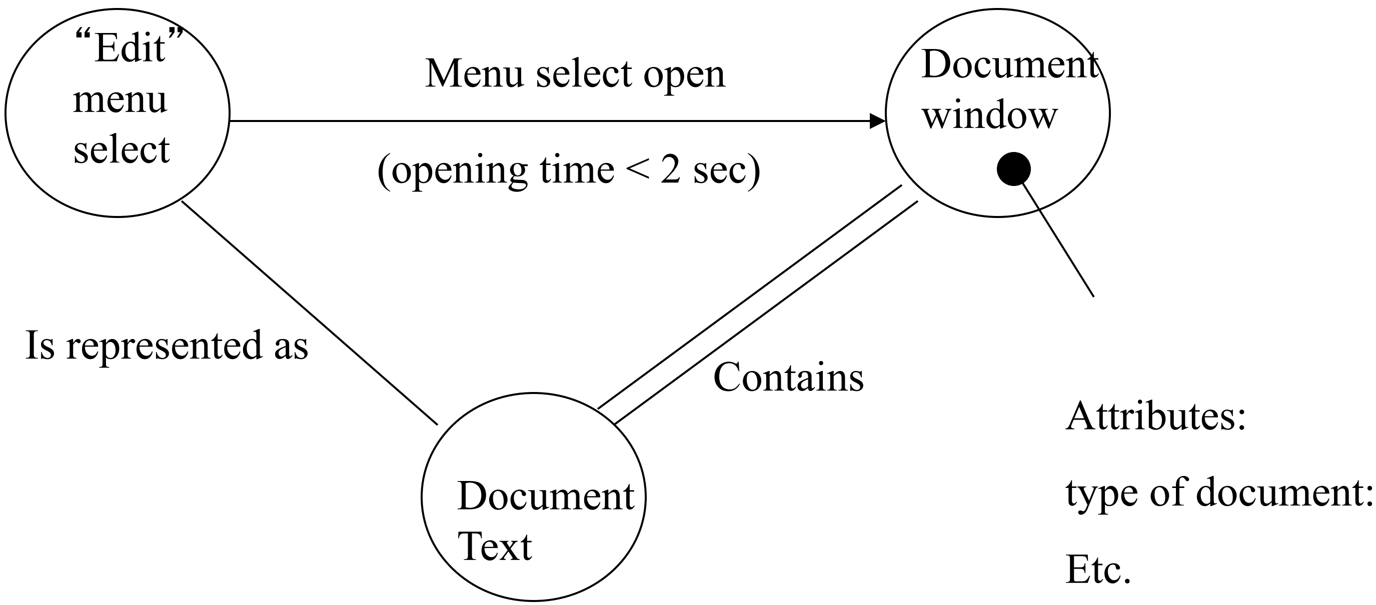
# Graph-based Testing Methods – How?

- Create a graph
  - A collection of *nodes* that represent objects, *links* that represent the relationships between nodes, *node weight* that describe the properties of a node, and *link weight* that describe some characteristic of a link
  - A directed link – the relationship moves in only one direction
  - A bidirectional link – The relationship moves in both direction
  - A parallel link – A number of different relationships are established



# Graph-based Testing Methods – An Example

- Editing a file
  - Edit – Opens a document window
  - Document window – provides a list of the window attributes (type of file) that are to be expected when the window is opening
    - Opening time should be less than 2 sec.



# Graph-based Testing Methods

- Drive test cases by traversing the graph and covering each of the relationships shown
- The test cases are designed in an attempt to find errors in any of the relationships
- Important Note:
  - Any equivalence relationships between objects must be tested
    - Transitivity to determine how the impact of relationships propagates across objects
      - x R y, y R z => x R z , derive tests to find errors in the calculation of z must consider a variety of values for both x and y
      - x R y, y R x must also be tested if the link is bidirectional
      - x R x should also be tested (e.g. “null action” or “no action”)
  - Node Coverage
    - No node has been left without exercising (test case)
  - Link coverage
    - No link has been left without exercising (test case)

# Syntax Checking

- Testing syntax of programs and not necessarily program functions
  - All acceptable (erroneous) inputs must be accepted (rejected)
- Explicit syntax testing
  - E.g. Programming languages
- Implicit syntax testing
  - E.g. Command line or arguments of Unix commands

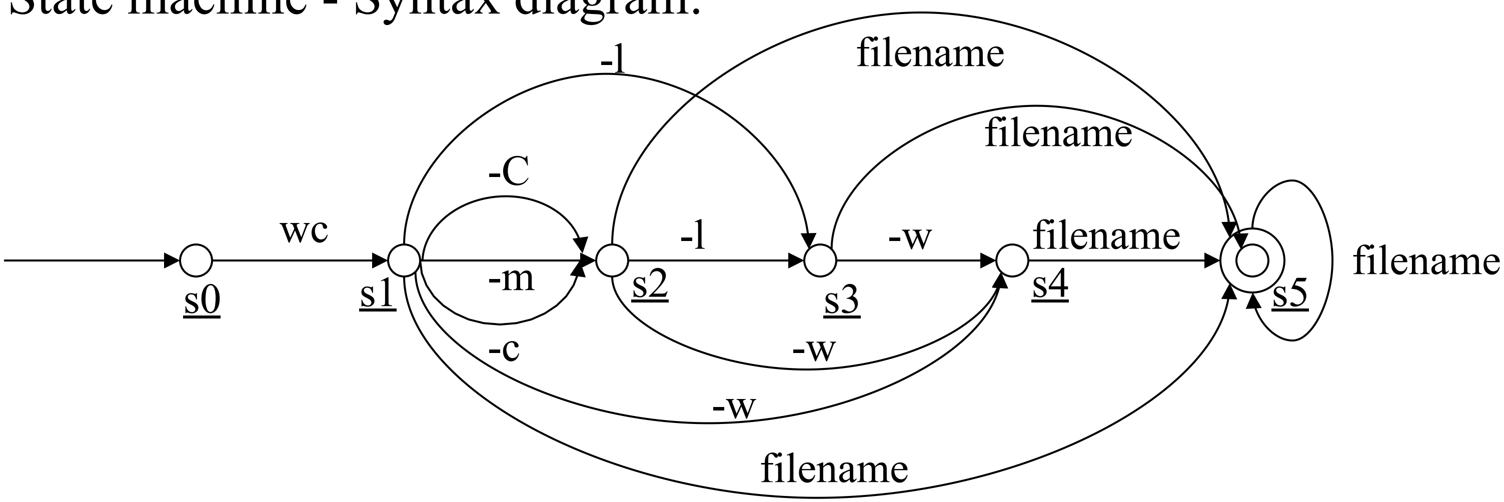
- Two possible levels:
  - Token level
    - E.g., reserve words, numbers, keywords, etc.
  - Lexical level
    - Characters

# Syntax Checking – Basic Steps

1. Identify the syntax of the target programming language
   - Drawing state machine may help
2. Write test cases to cover
   - All acceptable kinds of inputs
   - Every possible syntax error

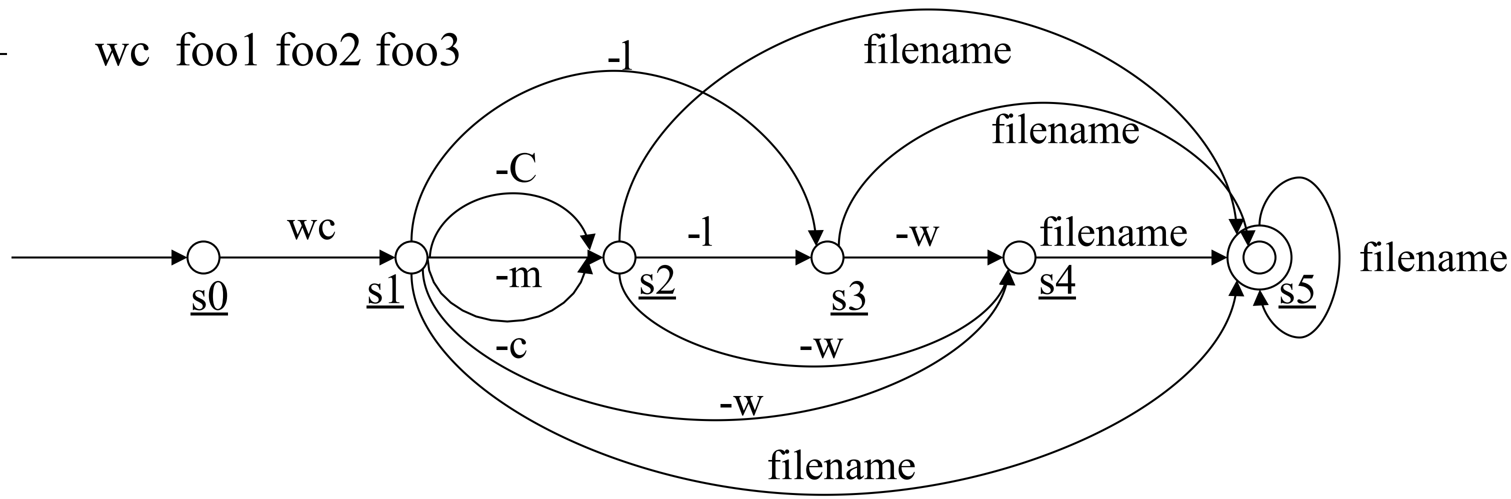
# Syntax Checking – An Example

- Unix "wc" word count command: *wc [-c | -m | -C] [-l] [-w] filename …*
  - -c Count bytes
  - -m Count characters
  - -C Same as –m
  - -l Count lines
  - -w Count words delimited by white space or new lines
  - … means repeat (e.g., file1 file2 file3)
  - [ ] means optional
- State machine - Syntax diagram:



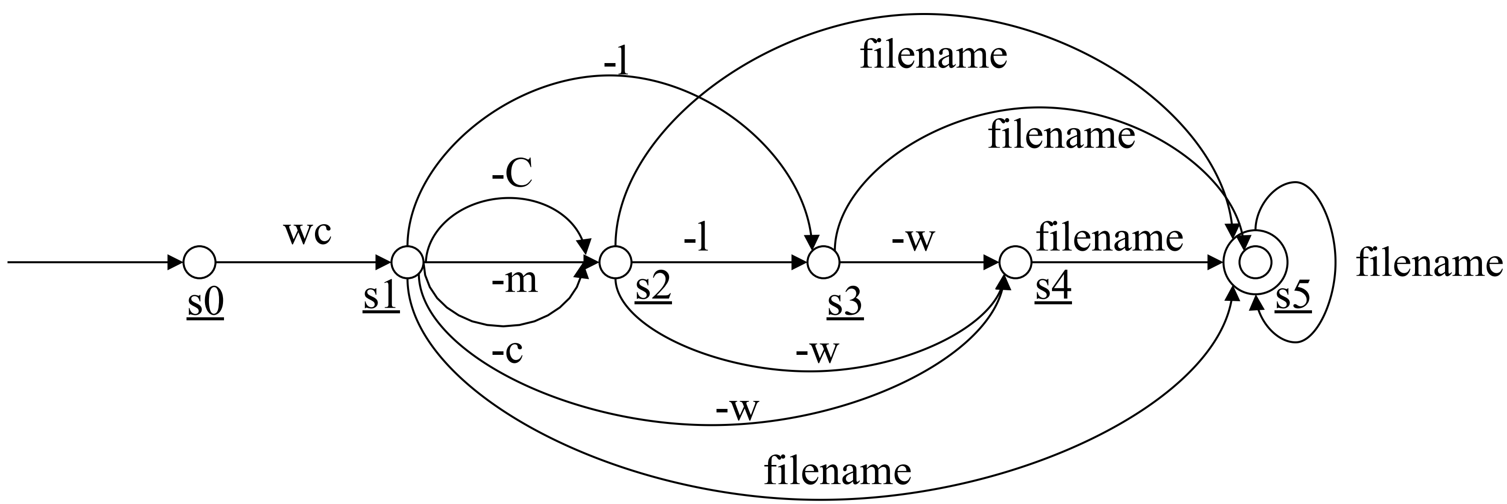
# Syntax Checking – Valid Syntax: All Transitions

- Cover all transitions
  - Each transition is covered by at least one test case
  - Covering transitions not necessarily means covering paths
    - There might be loop making it impossible to cover all loops
- Each test case must end up with the final state
- Possible set of valid test cases:
  - wc foo
  - wc  foo1 foo2
  - wc  foo1 foo2 foo3



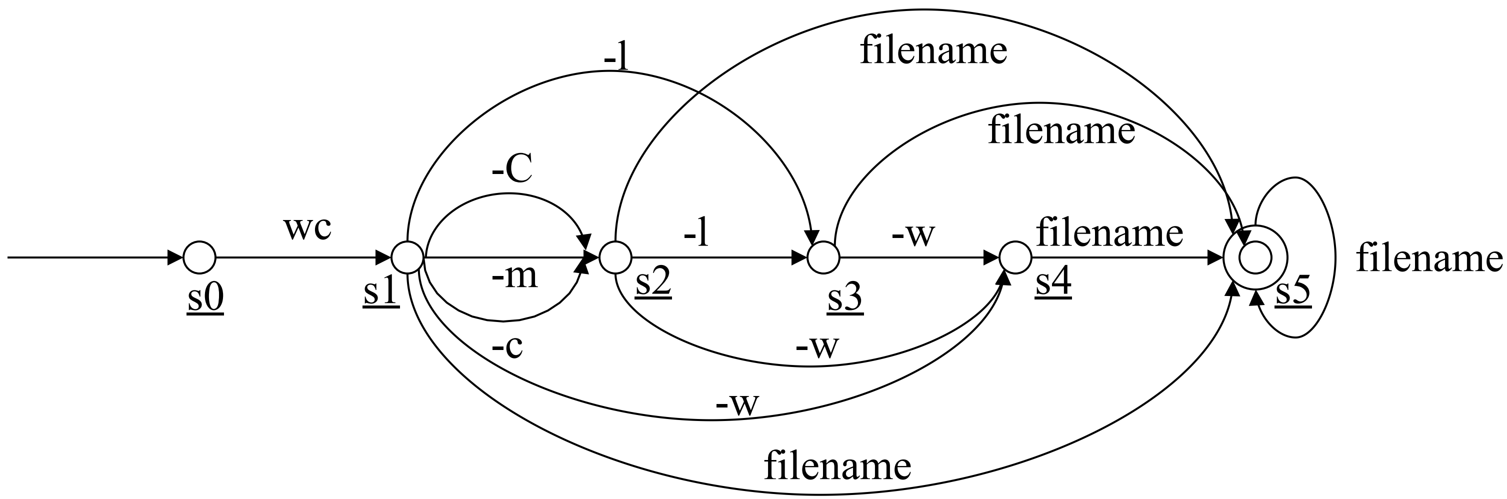
# Syntax Checking – Valid Syntax: All Transition Pairs

- Cover all transition pairs of the form (a, b) such that state b can follow state a
  - To make sure that we cover all optional flags (-c, -w, etc. )
- Possible set of valid test cases:
  - wc –c foo
  - wc -C –l foo
  - wc -m –w foo
  - wc -w foo



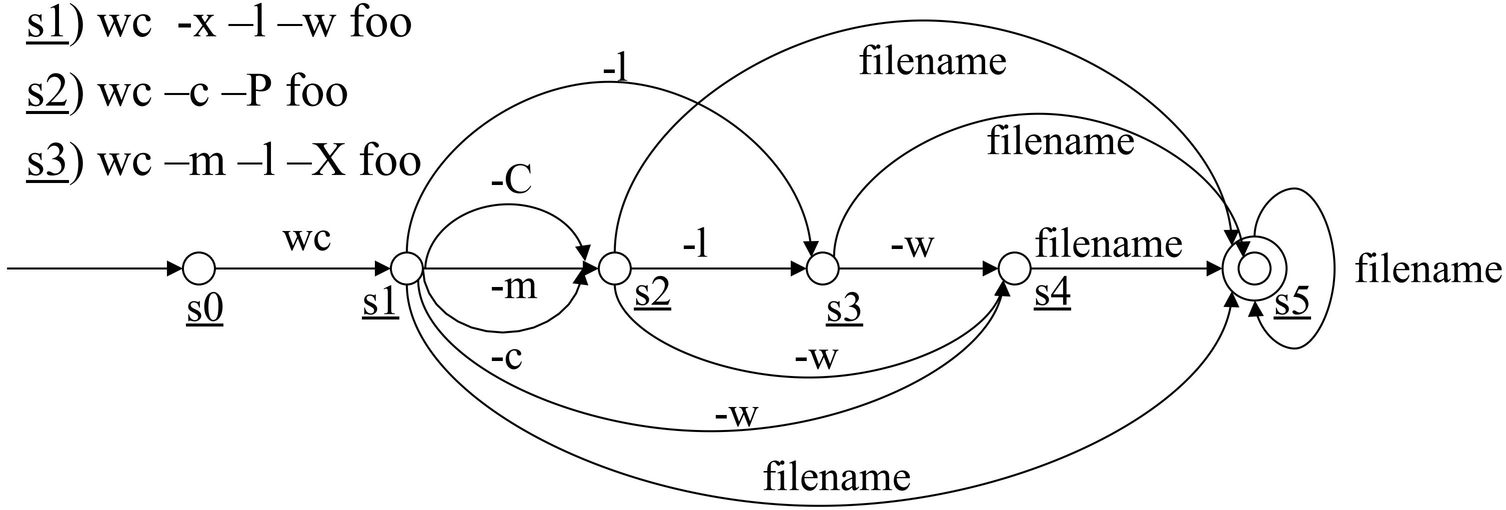
# Syntax Checking – Invalid Syntax

- To test invalid syntax, identify all "reaching strings"
  - Reaching String – For any state s, a reaching string is a string S that makes the state machine reaches that state s.
- Some examples
  - s1) wc
  - s2) wc –m
  - s3) wc –l
  - s4) wc -w



# Syntax Checking – Invalid Syntax – Test Cases

- Test each non-final state by using a reaching string
- Test invalid tokens
  - For each state si
    1. Start with reaching string
    2. Add a token not acceptable by state si
    3. Pretend that one of the leaving transition (leaving si if there is any) has been followed
    4. Add more token to get to the final state
- Some Examples
  - s1) wc -x –l –w foo
  - s2) wc –c –P foo
  - s3) wc –m –l –X foo

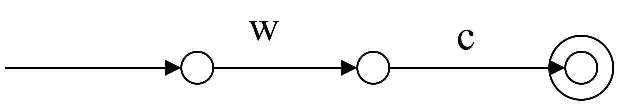


# Syntax Checking – Implicit Syntax

- Valid filename? Valid option?
  - Not explicitly referenced in syntax
  - We assume:
    - Any token starting with “-” is an option
    - Any token not starting with “-” is a filename

# Syntax Checking – Lexical Level

- Lexical syntax of "wc" command




- Invalid test cases can be generated using valid test cases and replacing "wc" with:
  - w
  - wcc
  - wcp
  - w
- The same strategy for the options
  - Test case sensitive options