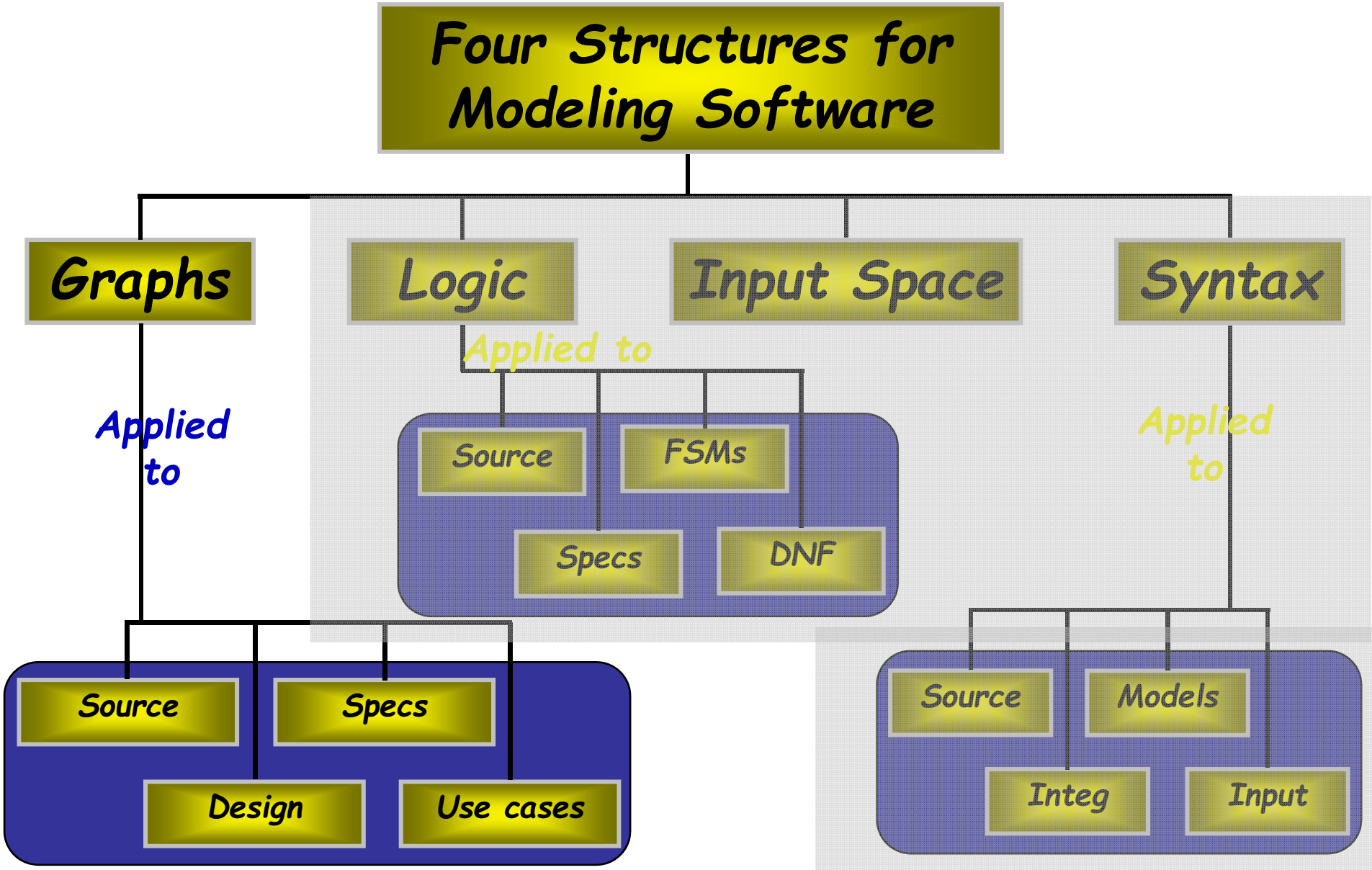
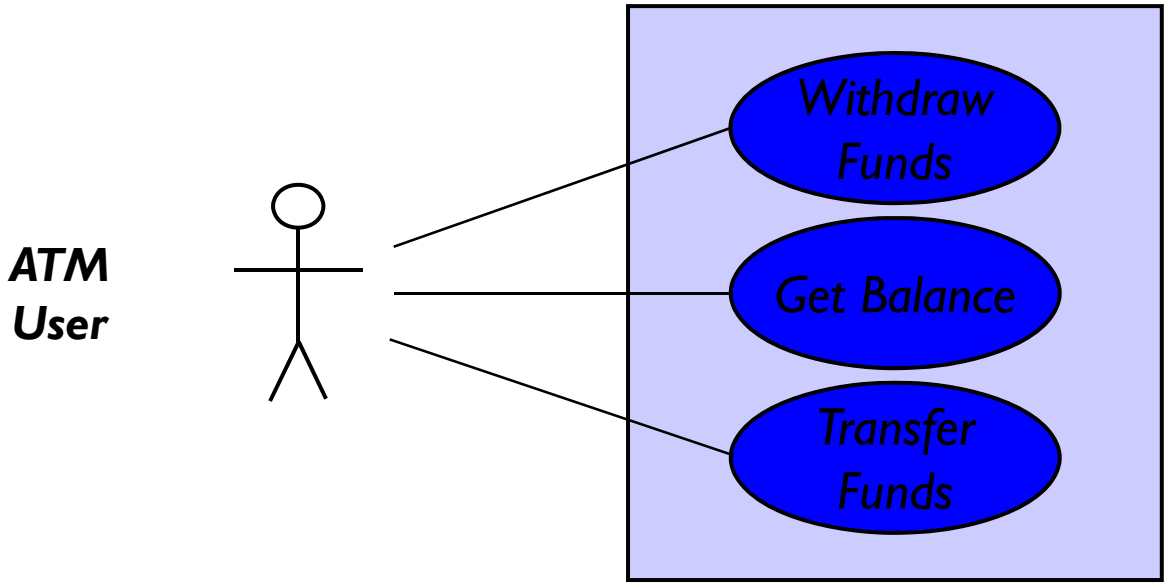

Four Structures for Modeling Software
Graphs
Logic
Input Space
Syntax
Applied to
Applied to
Applied to
Source
FSMs
Specs
DNF
Source
Specs
Design
Use cases
Source
Models
Integ
Input

UML Use Cases

- **UML use cases are often used to express software requirements**
- **They help express computer application workflow**

Simple Use Case Example



- **Actors : Humans or software components that use the software being modeled**
- **Use cases : Shown as circles or ovals**
- **Node Coverage : Try each use case once …**

Use Case graphs, by themselves, are not useful for testing

Elaboration

- Use cases are commonly elaborated (or documented)
- Elaboration is first written textually
  - Details of operation
  - Alternatives model choices and conditions during execution

Elaboration

- **Use Case Name : Withdraw Funds**
- **Summary : Customer uses a valid card to withdraw funds from a valid bank account.**
- **Actor : ATM Customer**
- **Precondition : ATM is displaying the idle welcome message**
- **Description :**
  - **Customer inserts an ATM Card into the ATM Card Reader.**
  - **If the system can recognize the card, it reads the card number.**
  - **System prompts the customer for a PIN.**
  - **Customer enters PIN.**
  - **System checks the card's expiration date and whether the card has been stolen or lost.**
  - **If the card is valid, the system checks if the entered PIN matches the card PIN.**
  - **If the PINs match, the system finds out what accounts the card can access.**
  - **System displays customer accounts and prompts the customer to choose a type of transaction. There are three types of transactions, Withdraw Funds, Get Balance and Transfer Funds. (The previous eight steps are part of all three use cases; the following steps are unique to the Withdraw Funds use case.)**

Elaboration of ATM Use Case—(2/3)

- **Description (continued) :**
  - Customer selects Withdraw Funds, selects the account number, and enters the amount.
  - System checks that the account is valid, makes sure that customer has enough funds in the account, makes sure that the daily limit has not been exceeded, and checks that the ATM has enough funds.
  - If all four checks are successful, the system dispenses the cash.
  - System prints a receipt with a transaction number, the transaction type, the amount withdrawn, and the new account balance.
  - System ejects card.
  - System displays the idle welcome message.

Elaboration of ATM Use Case—(3/3)

- **Alternatives** :
  - If the system cannot recognize the card, it is ejected and the welcome message is displayed.
  - If the current date is past the card's expiration date, the card is confiscated and the welcome message is displayed.
  - If the card has been reported lost or stolen, it is confiscated and the welcome message is displayed.
  - If the customer entered PIN does not match the PIN for the card, the system prompts for a new PIN.
  - If the customer enters an incorrect PIN three times, the card is confiscated and the welcome message is displayed.
  - If the account number entered by the user is invalid, the system displays an error message, ejects the card and the welcome message is displayed.
  - If the request for withdraw exceeds the maximum allowable daily withdrawal amount, the system displays an apology message, ejects the card and the welcome message is displayed.
  - If the request for withdraw exceeds the amount of funds in the ATM, the system displays an apology message, ejects the card and the welcome message is displayed.
  - If the customer enters Cancel, the system cancels the transaction, ejects the card and the welcome message is displayed.
- **Postcondition** :
  - Funds have been withdrawn from the customer’s account.

Wait A Minute …

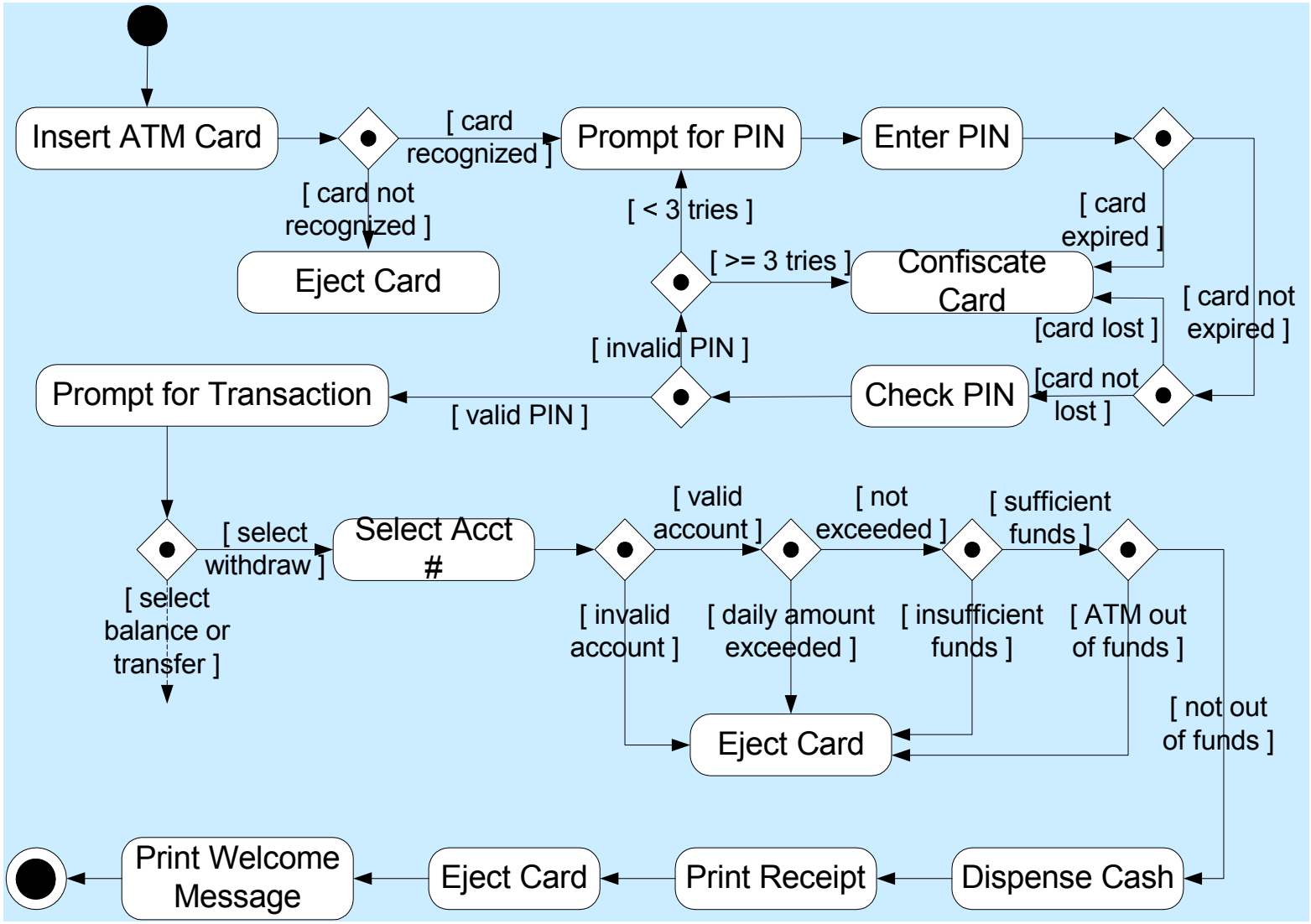
- **What does this have to do with testing ?**
- **Specifically, what does this have to do with graphs ???**
- **Remember our admonition : Find a graph, then cover it!**
- **UML has something very similar :**

Activity Diagrams

Use Cases to Activity Diagrams

- Activity diagrams indicate flow among activities
- Activities should model user level steps
- Two kinds of nodes:
  - Action states
  - Sequential branches
- Use case descriptions become action state nodes in the activity diagram
- Alternatives are sequential branch nodes
- Flow among steps are edges
- Activity diagrams usually have some helpful characteristics:
  - Few loops
  - Simple predicates
  - No obvious DU pairs

ATM Withdraw Activity Graph



Covering Activity Graphs

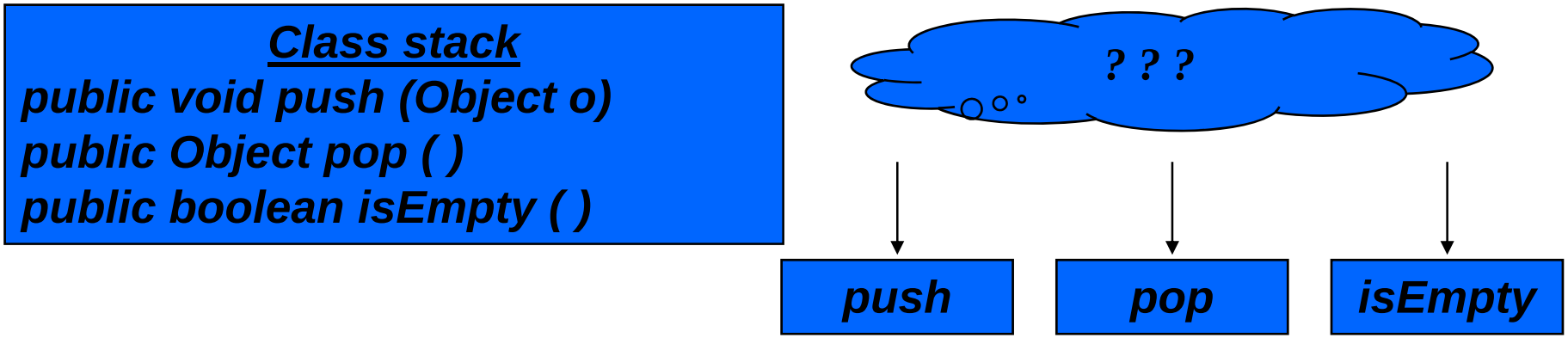
- Node Coverage
  - Inputs to the software are derived from labels on nodes and predicates
  - Used to form test case values
- Edge Coverage
- Data flow techniques do not apply
- Scenario Testing
  - Scenario : A complete path through a use case activity graph
  - Should make semantic sense to the users
  - Number of paths often finite
  - If not, scenarios defined based on domain knowledge
  - Use “specified path coverage”, where the set S of paths is the set of scenarios
  - Note that specified path coverage does not necessarily subsume edge coverage, but scenarios should be defined so that it does

Design Specifications

- A design specification describes aspects of what behavior software should exhibit
- A design specification may or may not reflect the implementation
  - More accurately – the implementation may not exactly reflect the spec
  - Design specifications are often called models of the software
- Sequencing constraints on class methods

Sequencing Constraints

- Sequencing constraints are rules that impose constraints on the order in which methods may be called
- They can be encoded as preconditions or other specifications



- *Tests can be created for these classes as sequences of method calls*
- *Sequencing constraints give an easy and effective way to choose which sequences to use*

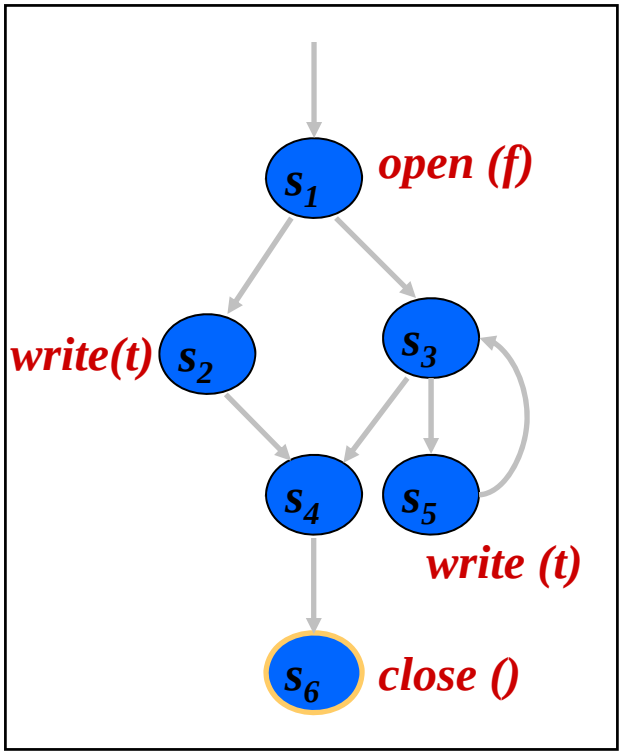
File ADT Example

class FileADT has three methods:

- ***open (String fName)*** *// Opens file with name fName*
- ***close ()*** *// Closes the file and makes it unavailable*
- ***write (String textLine)*** *// Writes a line of text to the file*

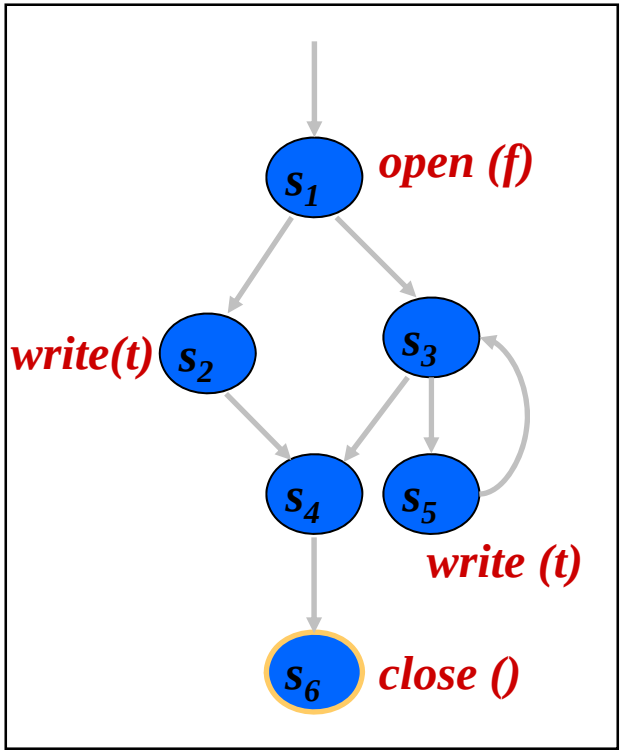
Valid sequencing constraints on FileADT:

1. *An open (f) must be executed before every write (t)*
2. *An open (f) must be executed before every close ()*
3. *A write (f) may not be executed after a close () unless there is an open (f) in between*
4. *A write (t) should be executed before every close ()*



Static Checking

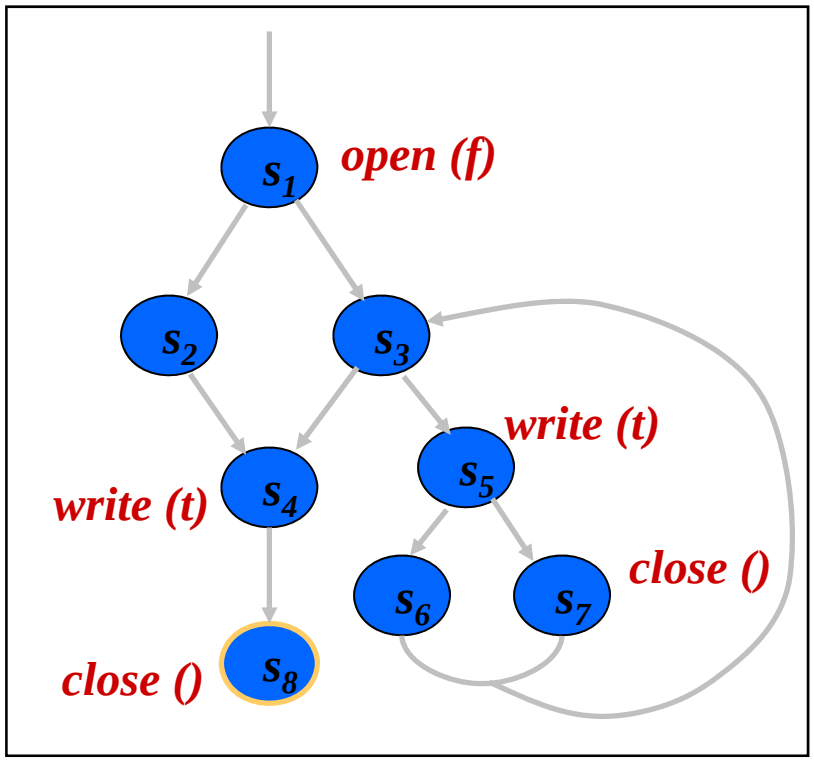
Is there a path that violates any of the sequencing constraints ?



- Is there a path to a write() that does not go through an open() ?
- Is there a path to a close() that does not go through an open() ?
- Is there a path from a close() to a write()?
- Is there a path from an open() to a close() that does not go through a write() ? (“write-clear” path)

Static Checking

Consider the following graph :



[7, 3, 4] – close () before write () !

Generation vs. Recognition

- **Generation** of tests based on coverage means producing a test suite to achieve a certain level of coverage
  - As you can imagine, generally very hard
  - Consider: generating a suite for 100% statement coverage easily reaches "solving the halting problem" level
  - Obviously hard for, say, mutant-killing
- **Recognition** means seeing what level of coverage an existing test suite reaches

Coverage and Subsumption

- Sometimes one coverage approach *subsumes* another
  - If you achieve 100% coverage of criteria A, you are guaranteed to satisfy B as well
    - For example, consider node and edge coverage
      - (there's a subtlety here, actually – can you spot it?)
- What does this mean?
  - Unfortunately, not a great deal
  - If test suite X satisfies "stronger" criteria A and test suite Y satisfies "weaker" criteria B
    - **Y may still reveal bugs that X does not!**
    - **For example, consider our running example and statement vs. branch coverage**
  - *It means we should take coverage with a grain of salt, for one thing*

Testing “for” Coverage

- Never seek to improve coverage *just for the sake of increasing coverage*
  - Well, unless it’s a command from-on-high
- Coverage is not the goal
  - Finding failures that expose faults is the goal
  - No amount of coverage will prove that the program cannot fail

“Program testing can be used to show the presence of bugs, but never to show their absence!” *– E. Dijkstra, Notes On Structured Programming*

The Purpose of Testing

"Program testing can be used to show the presence of bugs, but never to show their absence!" – *E. Dijkstra, Notes On Structured Programming*

- Dijkstra meant this as a criticism of testing and an argument in favor of more disciplined and total approaches (proving programs correct)
- But he also points out *what testing is good for:* exposing errors
- Coverage is valuable if and only if test sets with higher coverage are more likely to expose failures

The Purpose of Testing

"Program testing can be used to show the presence of bugs"

- When we first start "testing," we often want to "see that the program works"
  - Try out some scenarios and watch the program "do its stuff"
  - Surprised (annoyed) when (if) the program fails
  - *This is not really testing*: *testing is not the same as a demonstration*
  - Aim to break (your) code, if it can be broken

What's So Good About Coverage?

- Consider a fault that causes failure *every time the code is executed*
- Don't execute the code: cannot possibly find the fault!
- That's a pretty good argument for statement coverage

```
int findLast (int a[], int n, int x) {
    // Returns index of last element
    // in a equal to x, or -1 if no
    // such.  n is length of a

   int i;
   for (i = n-1; i >= 0; i--) {
      if (a[i] = x)
        return i;
   }
   return 0;
}
```

What's So Good About Coverage?

- We should have an *argument* for any kind of coverage:
  - "If I don't cover *this*, then there is more chance I'll miss a fault like *that*"
  - Backed with empirical data, preferably!

```
int findLast (int a[], int n, int x) {
    // Returns index of last element
    // in a equal to x, or -1 if no
    // such.  n is length of a

    int i;
    for (i = n-1; i >= 0; i--) {
      if (a[i] = x)
        return i;
    }
    return 0;
}
```