Introduction to Software Testing
(2nd edition)
Chapter 7.5

Graph Coverage for Specifications

Paul Ammann & Jeff Offutt

http://www.cs.gmu.edu/~offutt/softwaretest/

Design Specifications

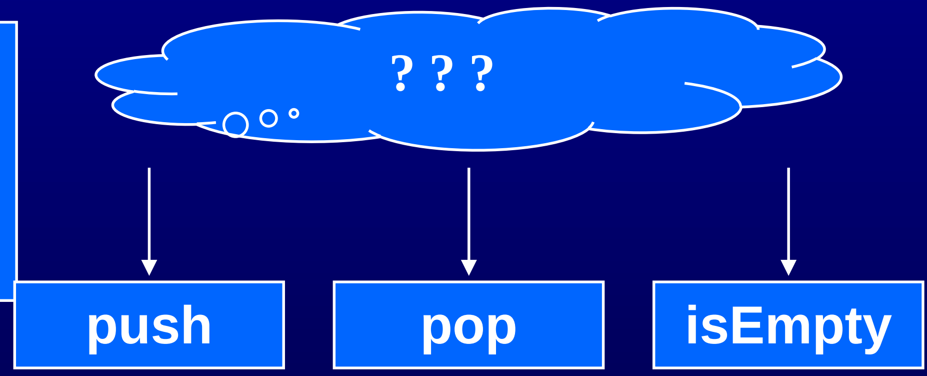
- A design specification describes aspects of what behavior software should exhibit
- A design specification may or may not reflect the implementation
  – More accurately – the implementation may not exactly reflect the spec
  – Design specifications are often called models of the software
- Two types of descriptions are used in this chapter
  1. Sequencing constraints on class methods
  2. State behavior descriptions of software

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
- Section 7.4 said that classes often have methods that do not call each other

Class stack

public void push (Object o)
public Object pop ( )
public boolean isEmpty ( )



- 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

Sequencing Constraints Overview

- Sequencing constraints might be
  - Expressed explicitly
  - Expressed implicitly
  - Not expressed at all
- Testers should derive them if they do not exist
  - Look at existing design documents
  - Look at requirements documents
  - Ask the developers
  - Last choice : Look at the implementation
- If they don't exist, expect to find more faults !
- Share with designers before designing tests
- Sequencing constraints do not capture all behavior

Queue Example

```
public int deQueue()
{
   // Pre: At least one element must be on the queue.
   … …

public enQueue (int e)
{
   // Post: e is on the end of the queue.
```

- Sequencing constraints are implicitly embedded in the pre and postconditions
  - enQueue () must be called before deQueue ()
- Does not include the requirement that we must have at least as many enQueue () calls as deQueue () calls
  - Can be handled by state behavior techniques

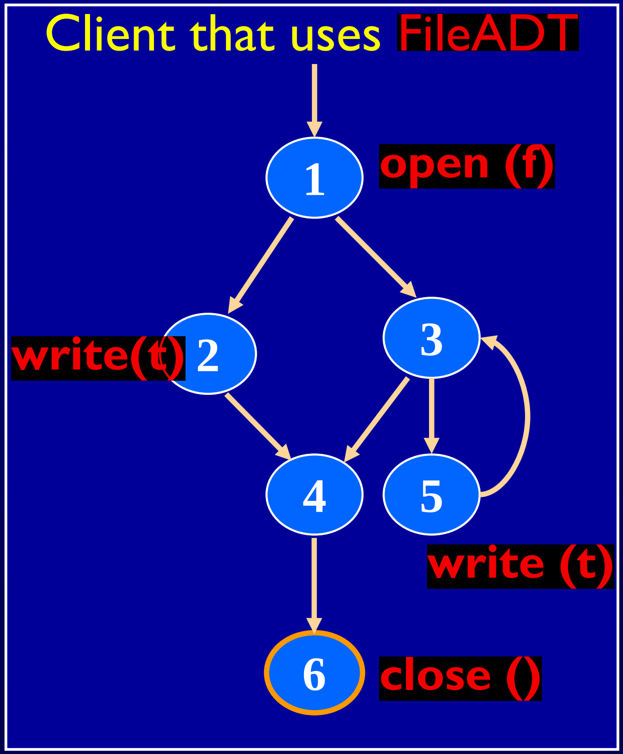
File ADT Example

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

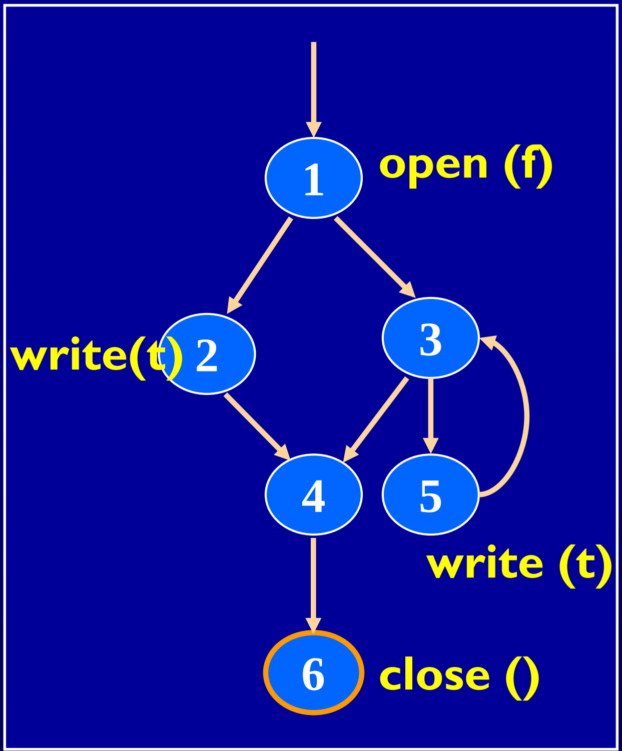
Valid sequencing constraints on FileADT:

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 ?

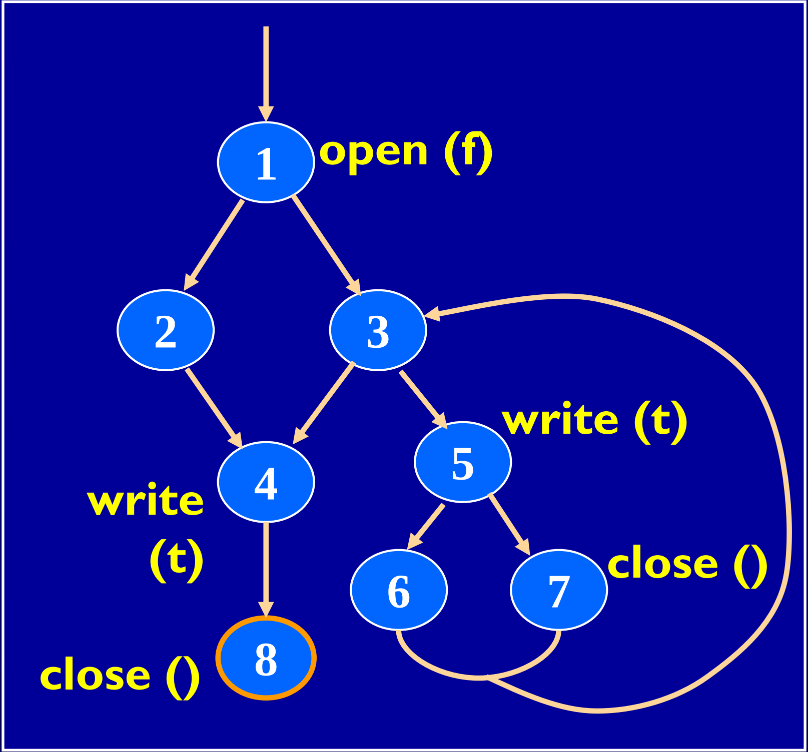


- Is there a path to a write() that does not go through an open() ?
- Is there a path to a close() that does not go through an open() ?
- Is there a path from a close() to a write()?
- Is there a path from an open() to a close() that does not go through a write() ? ("write-clear" path)

[1, 3, 4, 6] – ADT use anomaly!

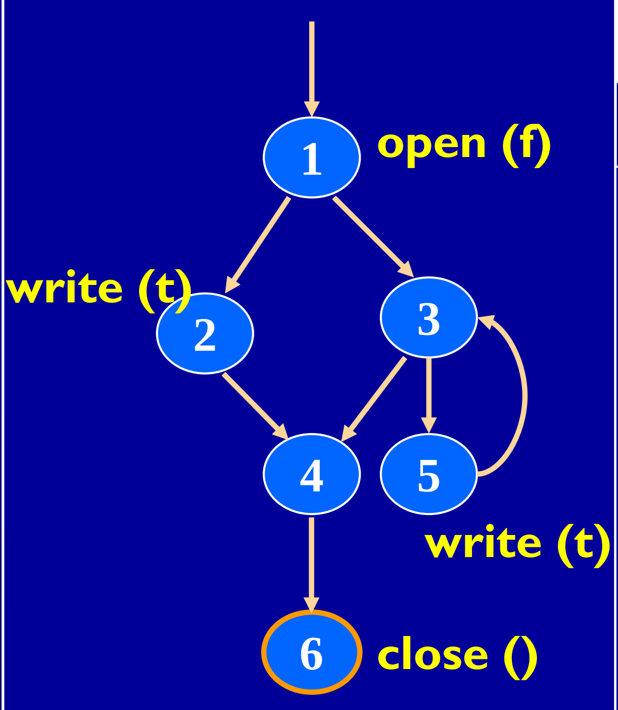
Static Checking

Consider the following graph :



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

Generating Test Requirements



[1, 3, 4, 6] – ADT use anomaly!

- But it is possible that the logic of the program does not allow the pair of edges [1, 3, 4]
- That is – the loop body must be taken at least once
- Determining this is undecidable – so static methods are not enough

- Use the sequencing constraints to generate test requirements
- The goal is to violate every sequencing constraint

Test Requirements for FileADT

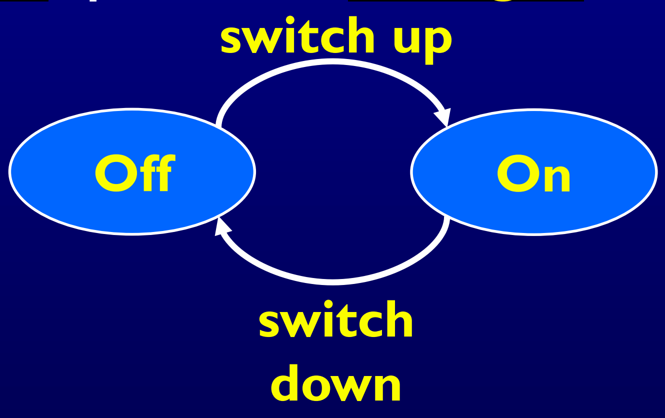
Apply to all programs that use FileADT

1. Cover every path from the start node to every node that contains a write() such that the path does not go through a node containing an open()
2. Cover every path from the start node to every node that contains a close() such that the path does not go through a node containing an open()
3. Cover every path from every node that contains a close() to every node that contains a write()
4. Cover every path from every node that contains an open() to every node that contains a close() such that the path does not go through a node containing a write()

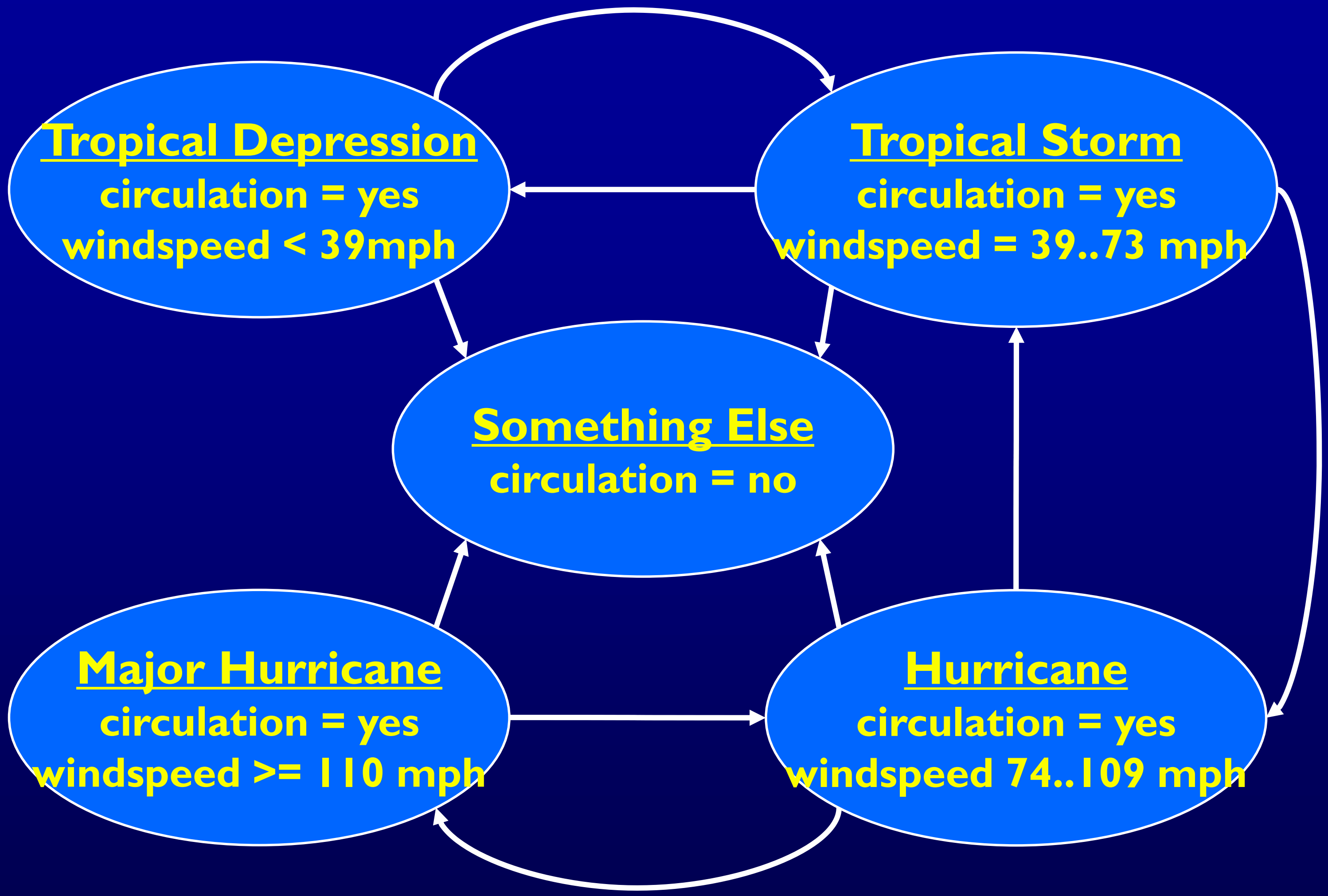
- If program is correct, all test requirements will be infeasible
- Any tests created will almost definitely find faults

Testing State Behavior

- A finite state machine (FSM) is a graph that describes how software variables are modified during execution
- Nodes : States, representing sets of values for key variables
- Edges : Transitions, possible changes in the state



Finite State Machine—Two Variables



Other variables may exist but not be part of state

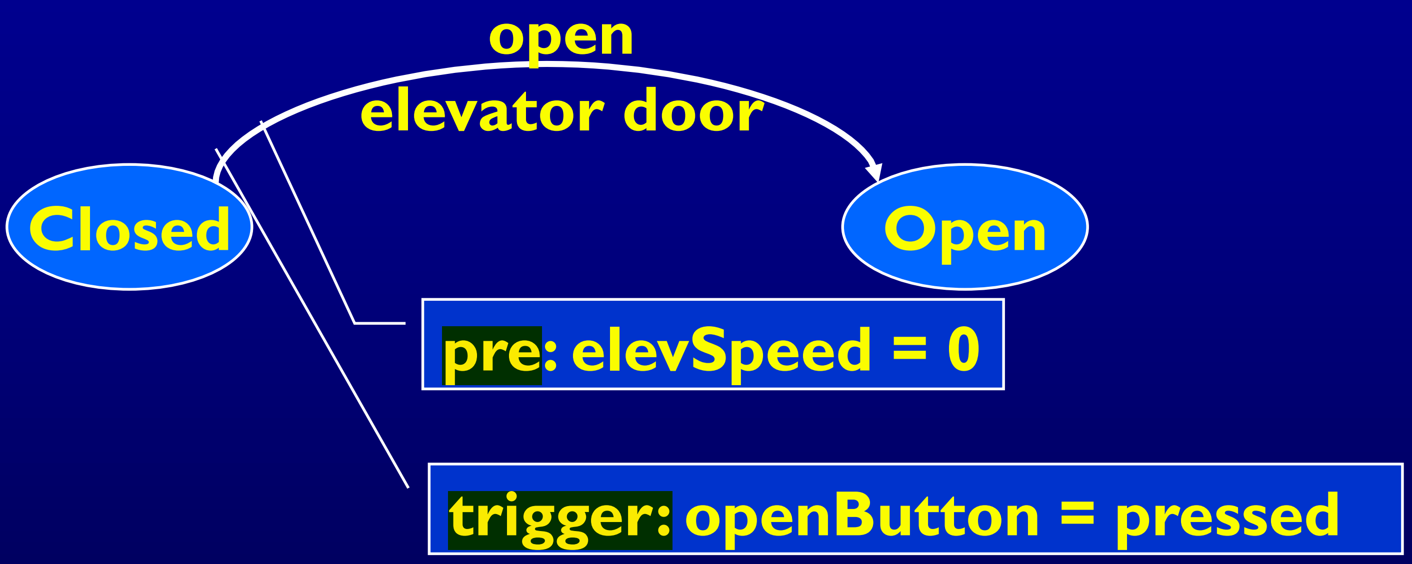
Finite State Machines are Common

- FSMs can accurately model many kinds of software
  - Embedded and control software (think electronic gadgets)
  - Abstract data types
  - Compilers and operating systems
  - Web applications
- Creating FSMs can help find software problems
- Numerous languages for expressing FSMs
  - UML statecharts
  - Automata
  - State tables (SCR)
  - Petri nets
- Limitation : FSMs are not always practical for programs that have lots of states (for example, GUIs)

Annotations on FSMs

- FSMs can be annotated with different types of actions
  - Actions on transitions
  - Entry actions to nodes
  - Exit actions on nodes
- Actions can express changes to variables or conditions on variables
- These slides use the basics:
  - Preconditions (guards) : conditions that must be true for transitions to be taken
  - Triggering events : changes to variables that cause transitions to be taken
- This is close to the UML Statecharts, but not exactly the same

Example Annotations



Covering FSMs

- Node coverage : execute every state (*state coverage*)
- Edge coverage : execute every transition (*transition coverage*)
- Edge-pair coverage : execute every pair of transitions (*transition-pair*)
- Data flow:
  - Nodes often do not include defs or uses of variables
  - Defs of variables in triggers are used immediately (the next state)
  - Defs and uses are usually computed for guards, or states are extended
  - FSMs typically only model a subset of the variables
- Generating FSMs is often harder than covering them …

Deriving FSMs

- With some projects, an FSM (such as a statechart) was created during design
  - Tester should check to see if the FSM is still current with respect to the implementation
- If not, it is very helpful for the tester to derive the FSM
- Strategies for deriving FSMs from a program:
  1. Combining control flow graphs (*wrong*)
  2. Using the software structure (*wrong*)
  3. Modeling state variables
- Example based on a digital watch …
  - Class Watch uses class Time

Class Watch

```
class Watch
// Constant values for the button (inputs)
private static final int NEXT = 0;
private static final int UP   = 1;
private static final int DOWN = 2;
// Constant values for the state
private static final int TIME      = 5;
private static final int STOPWATCH = 6;
private static final int ALARM     = 7;
// Primary state variable
private int mode = TIME;
// Three separate times, one for each state
private Time watch, stopwatch, alarm;

public Watch () // Constructor
public void doTransition (int button) // Handles inputs
public String toString () // Converts values
```

```
class Time   ( inner class )
private int hour   = 0;
private int minute = 0;

public void changeTime (int button)
public String toString ()
```

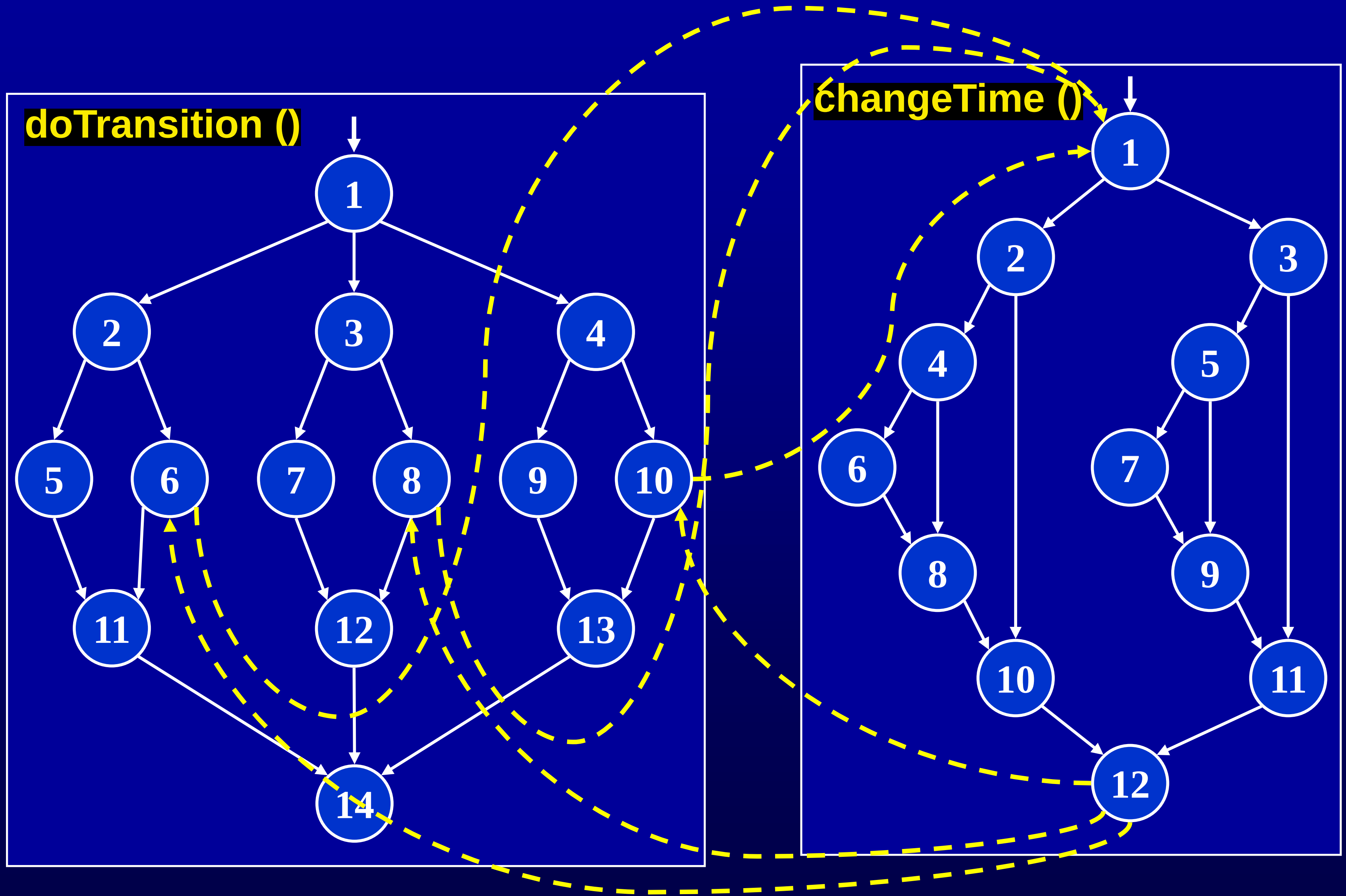
```
// Takes the appropriate transition when a button is pushed.
public void doTransition (int button)
{
   switch ( mode )
   {
      case TIME:
         if (button == NEXT)
            mode = STOPWATCH;
         else
            watch.changeTime (button);
         break;
      case STOPWATCH:
         if (button == NEXT)
            mode = ALARM;
         else
            stopwatch.changeTime (button);
         break;
      case ALARM:
         if (button == NEXT)
            mode = TIME;
         else
            alarm.changeTime (button);
         break;
      default:
         break;
   }
}  // end doTransition()
```

```
// Increases or decreases the time.
// Rolls around when necessary.
public void changeTime (int button)
{
   if (button == UP)
   {
      minute += 1;
      if (minute >= 60)
      {
         minute = 0;
         hour += 1;
         if (hour > 12)
            hour = 1;
      }
   }
   else if (button == DOWN)
   {
      minute -= 1;
      if (minute < 0)
      {
         minute = 59;
         hour -= 1;
         if (hour <= 0)
            hour = 12;
      }
   }
}  // end changeTime()
```

1. Combining Control Flow Graphs

- The first instinct for inexperienced developers is to draw CFGs and link them together
- This is really not an FSM
- Several problems
  - Methods must return to correct callsites—implicit nondeterminism
  - Implementation must be available before graph can be built
  - This graph does not scale up
- Watch example …

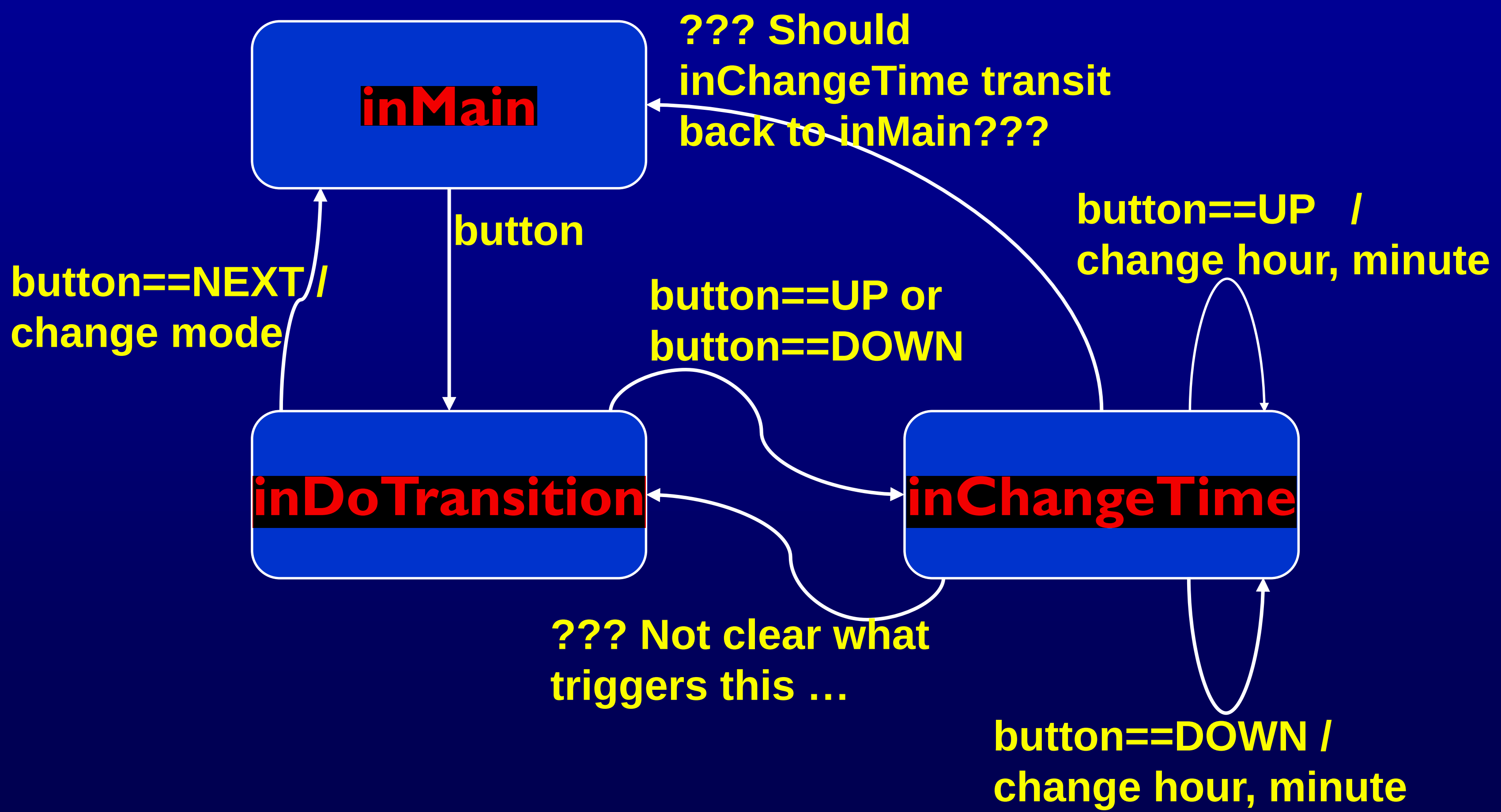
CFGs for Watch



2. Using the Software Structure

- A more experienced programmer may map methods to states
- These are really not states
- Problems
  - Subjective—different testers get different graphs
  - Requires in-depth knowledge of implementation
  - Detailed design must be present
- Watch example …

Software Structure for Watch



3. Modeling State Variables

- More mechanical
- State variables are usually defined early
- First identify all state variables, then choose which are relevant
- In theory, every combination of values for the state variables defines a different state
- In practice, we must identify ranges, or sets of values, that are all in one state
- Some states may not be feasible

State Variables in Watch

Constants

- ~~NEXT, UP, DOWN~~
- ~~TIME, STOPWATCH, ALARM~~

Not relevant, really just values

Non-constant variables in class Watch

- int mode (values: TIME, STOPWATCH, ALARM)
- Time watch, stopwatch, alarm

State Variables in Time

Non-constant variables in class Time

- int hour (values: 1..12)
- int minute (values: 0 .. 59)

12 X 60 values is 720 states

Clearly, that is too many

Combine values into ranges of similar values :

- hour : 1.. 11, 12
- minute : 0, 1.. 59

Clumsy … Not sequential …

let's combine hour and minute …

Four states : (1..11, 0); (12, 0); (1..11, 1.. 59); (12, 1 .. 59)

Time : 12:00, 12:01..12:59, 01:00 .. 11:59

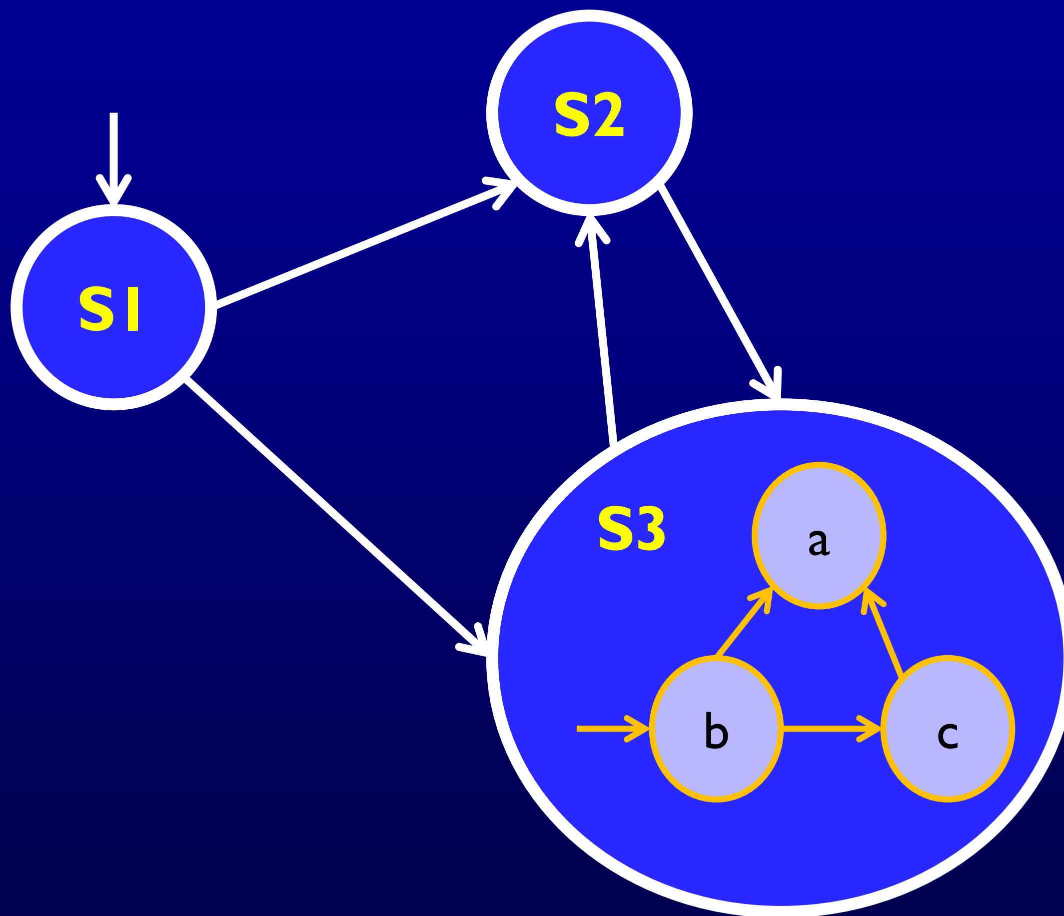
These require lots of thought and semantic domain knowledge of the program

Hierarchical FSMs

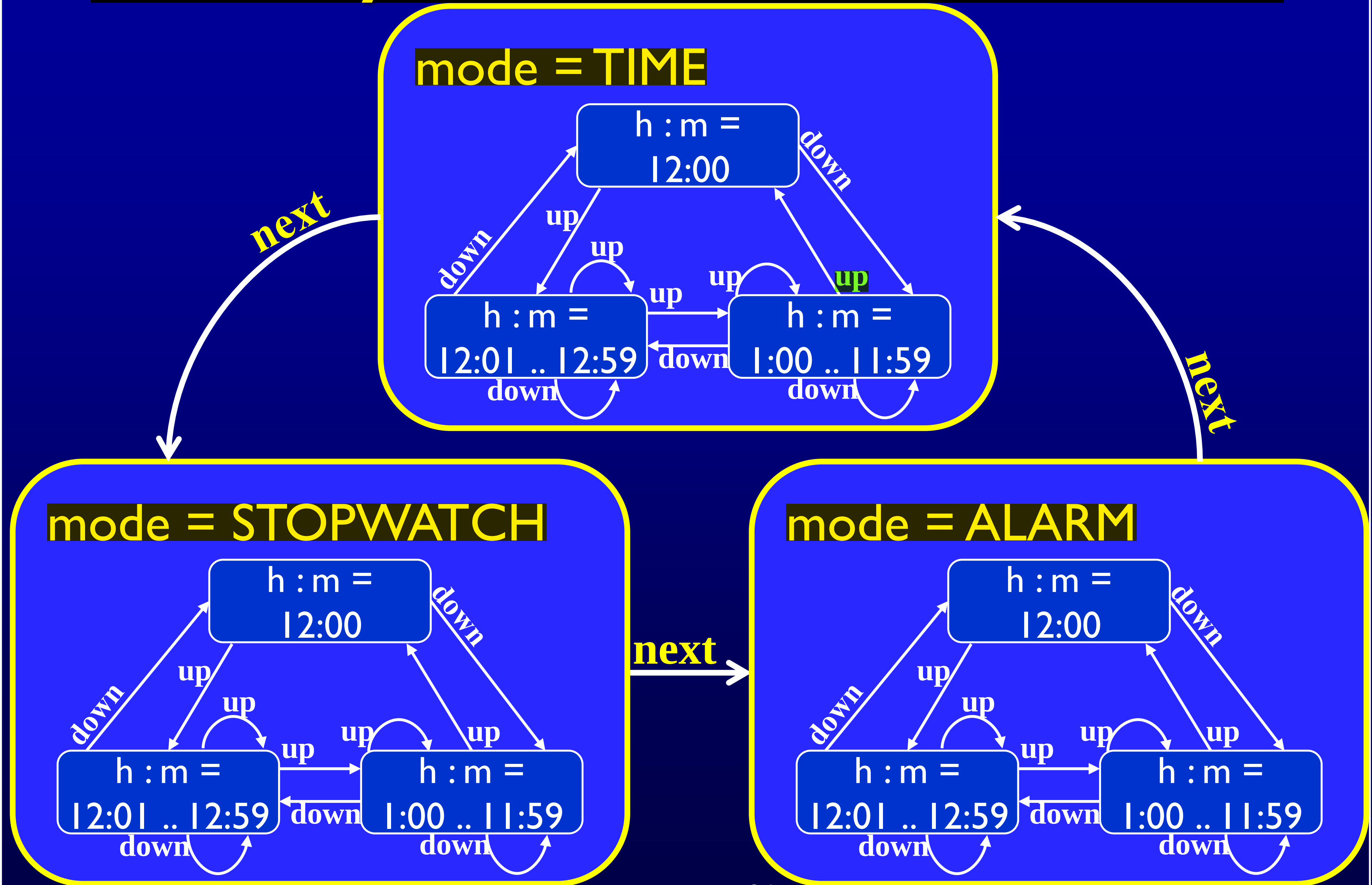
One FSM is contained within the other

Class Watch uses class Time

How can we model two classes—one that uses another?



Watch / Time Hierarchical FSM



Summary–Tradeoffs in Applying Graph Coverage Criteria to FSMs

- Two advantages
  1. Tests can be designed before implementation
  2. Analyzing FSMs is much easier than analyzing source

- Three disadvantages
  1. Some implementation decisions are not modeled in the FSM
  2. There is some variation in the results because of the subjective nature of deriving FSMs
  3. Tests have to be "mapped" to actual inputs to the program – the names that appear in the FSM may not be the same as the names in the program