Graph Coverage for Specifications

CS 3250
Software Testing

[Ammann and Offutt, "Introduction to Software Testing," Ch. 7.5]

Overview

- Software specification describes aspects of what behavior software should exhibit

- Two types of descriptions
  - Sequencing constraints on class methods
  - State behavior descriptions of software

Sequencing Constraints

- Sequencing constraints are rules that impose constraints on the order in which methods may be called
  - Example: cannot pop an element from a stack until something has been pushed onto it
- Sequencing constraints give an easy and effective way to choose which sequences to use
- Sequencing constraints may be
  - Expressed explicitly
  - Expressed implicitly
  - Not expressed at all
- Sometimes, they can be encoded as preconditions or other specifications

Sequencing Constraints

- If they are not expressed, testers should derive them
  - Look at existing design documents
  - Look at requirement documents
  - Ask the developers
  - Look at the implementation (last choice)

- Testers should share sequencing constraints with designers before designing tests

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
}
```

Implicit sequencing constraints occur between `enQueue()` and `deQueue()`

`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 technique

Sequencing constraints do not capture all behavior, but only abstract certain key aspects

File ADT Example

class FileADT has three methods:

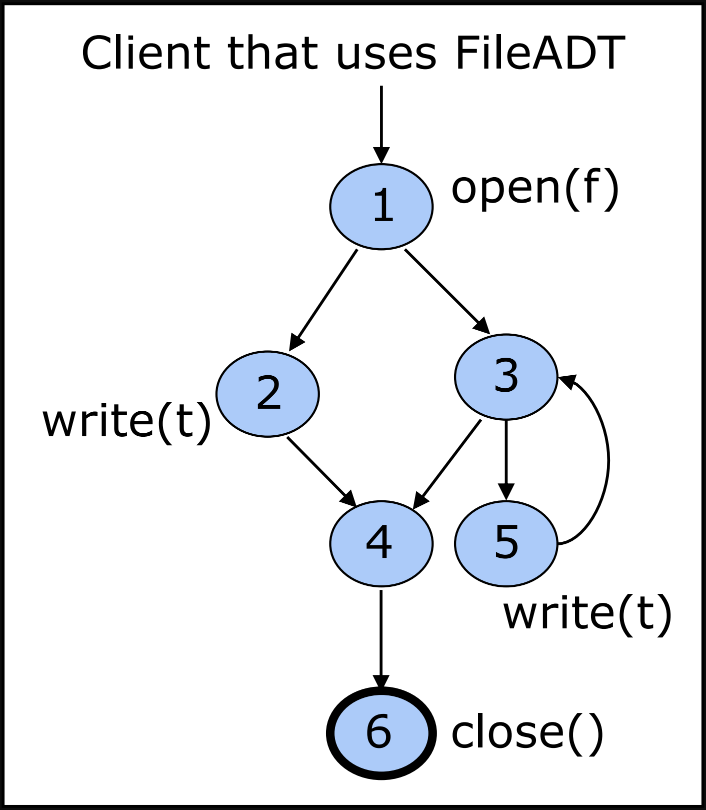
- `open(String fName)` // Opens file with name fName
- `close()` // Closes the file and makes it unavailable
- `write(String textLine)` // Writes a line of text to the file

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)` must not be executed after a `close()` unless there is an `open(f)` in between
4. A `write(t)` should be executed before every `close()`
5. A `close()` must not be executed after a `close()` unless and `open(f)` appears in between
6. An `open(f)` must not be executed after an `open(f)` unless a `close()` appears in between

Constraints are used to evaluate software that uses the class (a ”client”)

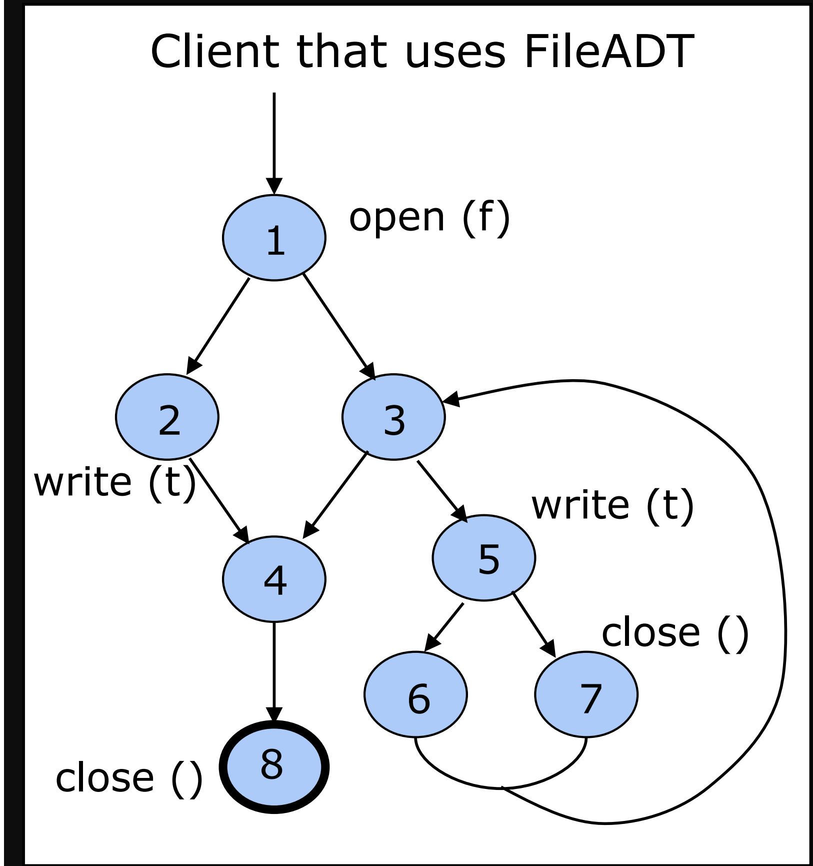
File ADT Example: Client 1



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 at least one write()?
    - Possible problem: path [1,3,4,6]
  - Is there a path from a close() to a close() that does not go through an open()?

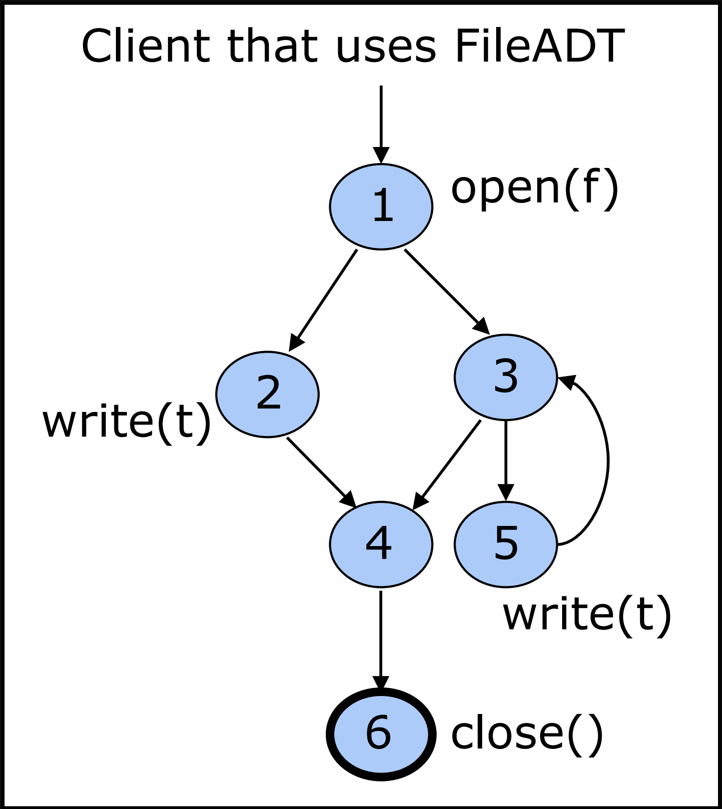
File ADT Example: Client 2



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 at least one write()?
  - Is there a path from a close() to a close() that does not go through an open()?
    - Path [7,3,4], close() before write()

File ADT Example: Client 1



Goal: Violate every sequencing constraint

Dynamic checking

- Consider path [1,3,4,6] where no write() appears
  - It is possible that the logic of the program does not allow the edge (3,4) unless the loop [3,5,3] is taken at least once
  - Deciding whether the path [1,3,4,6] can be taken or not is undecidable
  - This situation can be checked only by executing the program – static checking is not enough
  - Thus, we generate test requirements to try to violate the sequencing constraints

File ADT Example: Test Requirements

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()

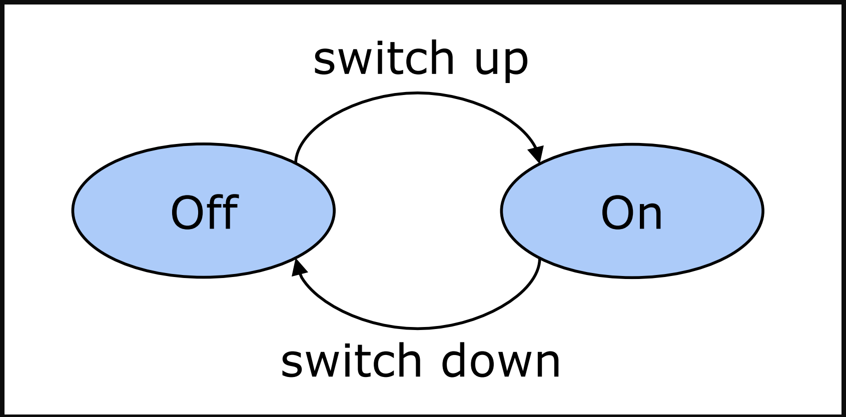
5. Cover every path from every node that contains an open() to every node that contains an open()

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

Testing State Behavior

- Other major method for using graphs based on specifications is to model state behavior of the software using finite state machine

- A finite state machine (FSM) is a graph that describes how software variables are modified during execution
  - Nodes represent states in the execution behavior
    - States represent values of variables
  - Edges represent transitions among the states
    - Transitions represent changes in the state



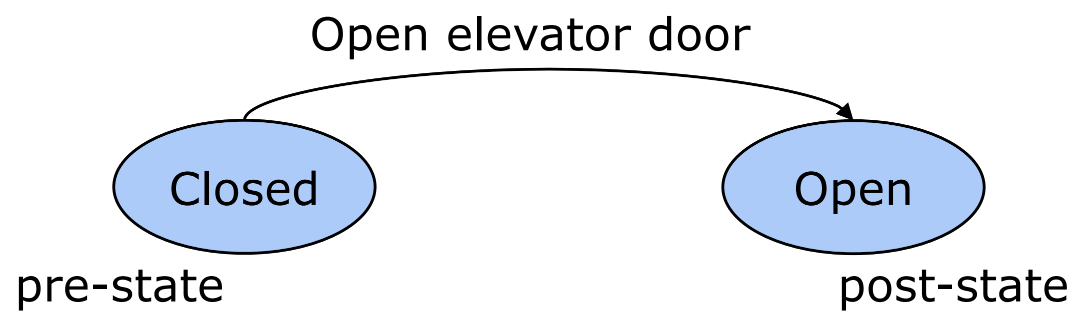
Finite State Machine (FSM)

- FSMs are used to model state behavior of many kinds of software
  - Embedded and control software (cell phones, watches, remote controls, cars, traffic signals, airplane flight guidance)
  - Compilers and operating systems
  - Web applications
- Creating FSMs can help find software problems
- Many languages have been developed to express FSMs
  - UML statecharts, automata, state tables, petri nets
- Limitation
  - "State explosion" – FSMs are not always practical for programs that have lots of states

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
- When the variables change, the software is considered to move from the pre-state to the post-state
  - If a transition's pre-state and post-state are the same, the values of state variables will not change

Annotations on FSMs



pre: elevSpeed = 0

trigger: openButton = pressed

Precondition or guard on transition

Define values that specific variables must have for the transition to be enabled

Trigger event

Change in variable values that cause the transition to be taken

- Before–values: values the triggering event has before the transition
- After-values: values the triggering event has after the transition

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 coverage)
- Data flow coverage:
  - 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

Modeling state variables

- Consider state variables
- 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
- Steps:
  - Identify the state variables
  - Choose which are actually relevant to the FSM

Example: Deriving FSM (`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 ()
```

Example: Deriving FSM (`Watch`)

```java
// 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()
```

```java
// 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()
```

State Variables in `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
```

Constants

Not relevant, really just values

Non-Constant variables

Relevant, affect the changes of state

Consider values

- mode (values: TIME, STOPWATCH, ALARM)

State Variables in `Time`

```
class Time    // inner class
private int hour = 0;
private int minute = 0;

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

Non-Constant variables

Relevant, affect the changes of state

Consider every combination of values

- hour (values: 1 … 12)
- minute (values: 0 … 59)

12 x 60 values = 720 states … too many

Combine values into ranges of similar values

- hour (values: 1…11, 12)
- minute (values: 0, 1…59)

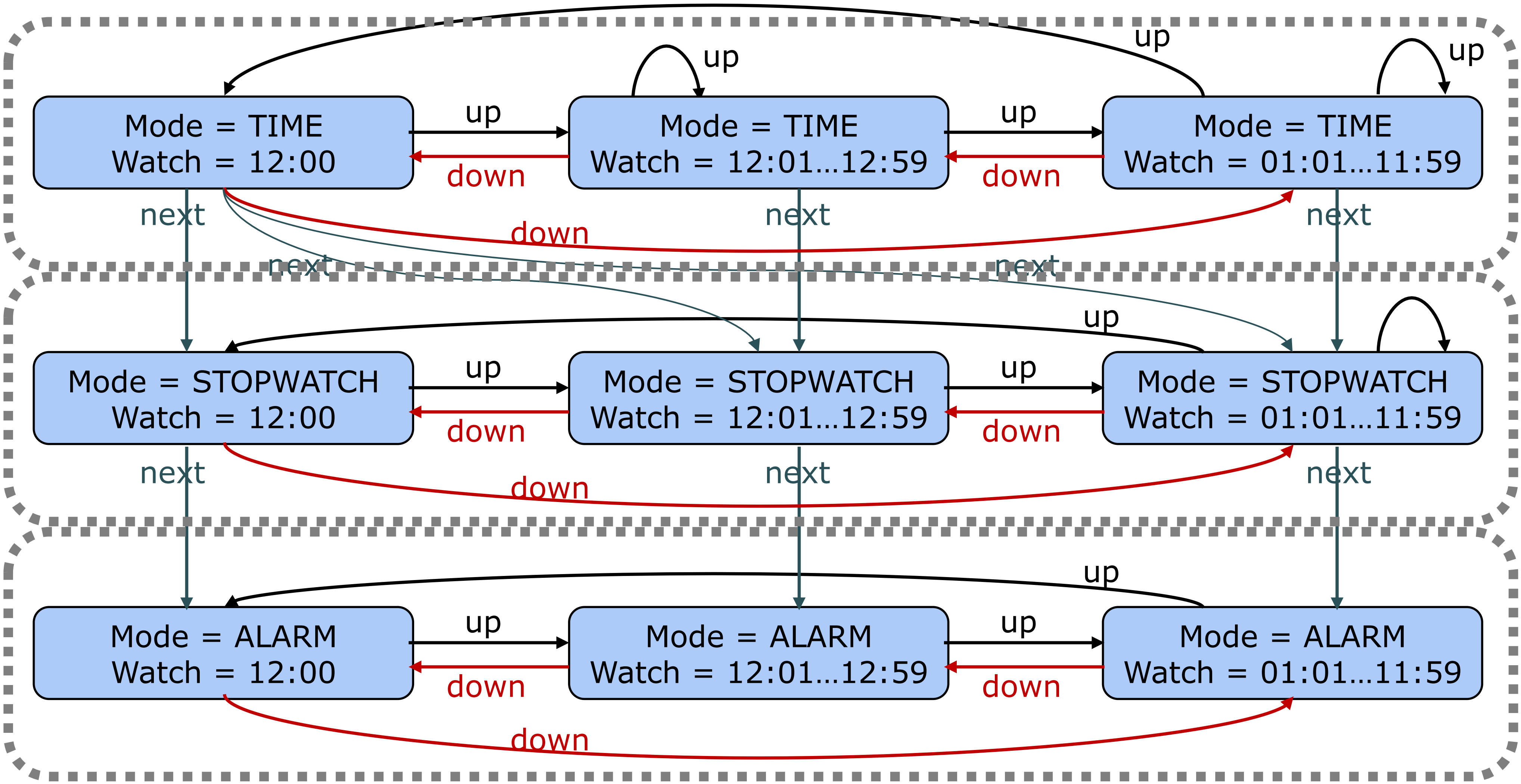
Four states: (1…11, 0), (12, 0), (1…11, 1…59), (12, 1…59) …
Clumsy, not sequential

Combine values in ranges (another way: hour and minute)

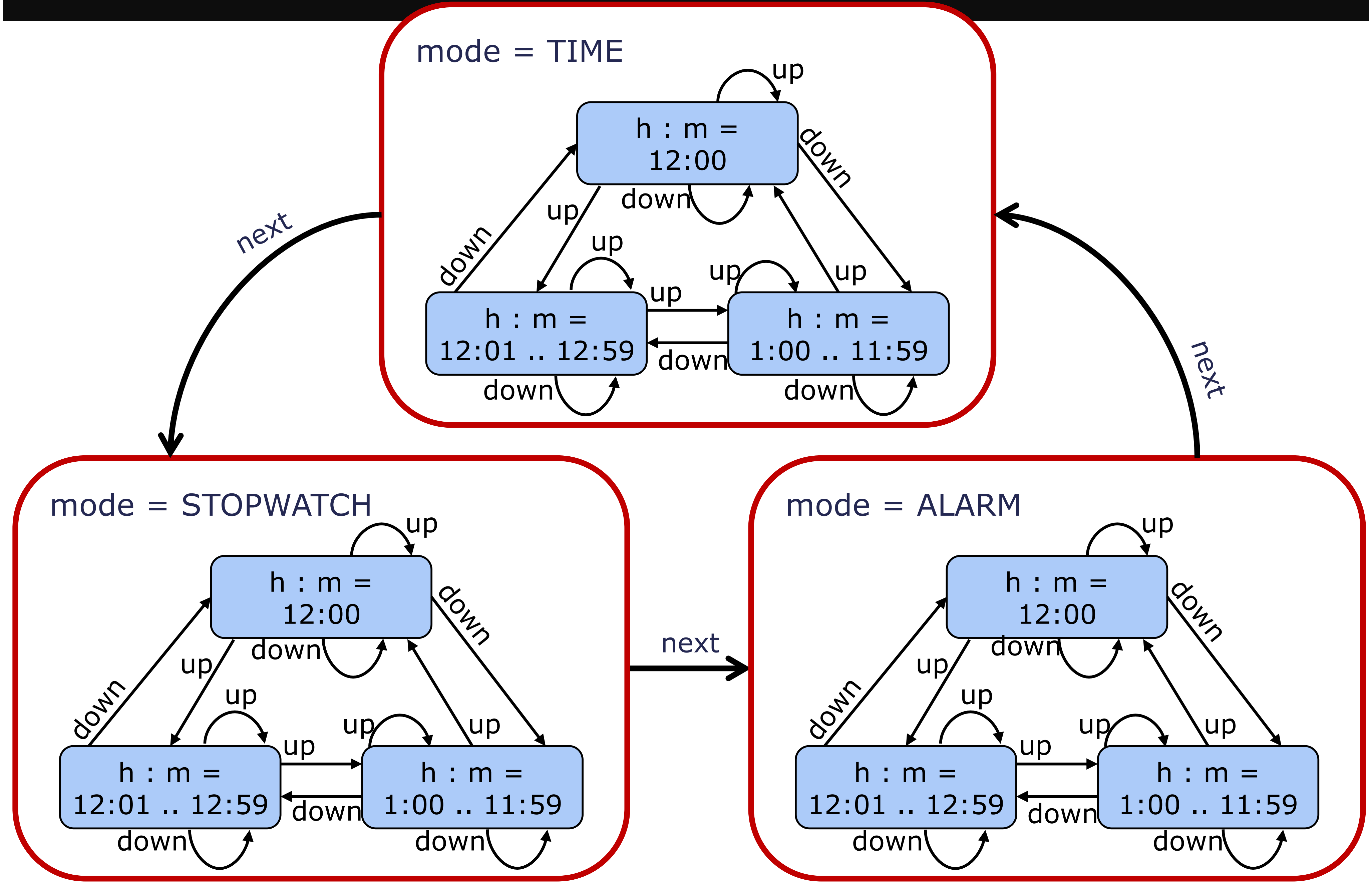
- Time: 12:00, 12:01…12:59, 01:00…11:59)

These require semantic domain knowledge of the program

FSM for `Watch/Time`



Hierarchical FSM for `Watch/Time`



Summary

- Advantages of applying graph coverage criteria to FSMs
  - Tests can be designed before implementation
  - Analyzing FSMs is easier than analyzing source
- Disadvantages of applying graph coverage criteria to FSMs
  - Some implementation decisions are not modeled in the FSM
  - Deriving FSMs may be subjective
  - The names appearing in the FSM may not be the same as the names in the program