Model-Based Testing

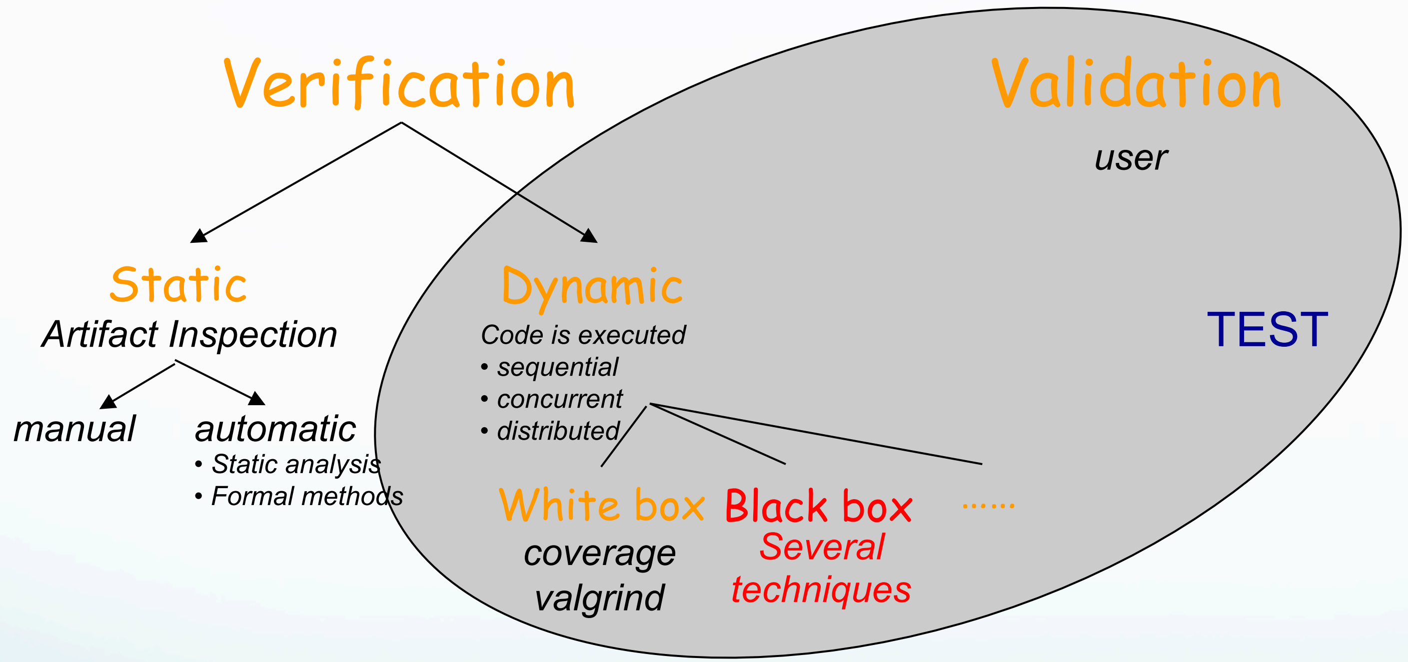
(DIT848 / DAT260)
Spring 2013

Lecture 2
Specification and Black Box Testing

Gerardo Schneider
Dept. of Computer Science and Engineering
Chalmers | University of Gothenburg

(Some slides based on material by Magnus Björk)

Terminology



Test cases

- Clear description of tests to be performed
  - Possible for a colleague or a computer to perform

- Manual, scripted or automatic:
  - Manual:
    - Clear text description for humans
    - Typically used for system properties
  - Scripted:
    - Executable description for computers
    - Typically used for lower level properties (unit tests, etc)
    - E.g. in xUnit, DejaGnu, bash or perl
    - Must still be readable for developers!
      - Serves a documentation purpose
  - Automatic:
    - Automatically generated and executed by computer
    - Based on formal specification

Test case 'elements'

- **Fundamental** components
  - Action: what to do in the test
  - Expected outcome: how the system should respond
    - **Good** expected outcomes are specific
      - "The function call returns 3"
      - "'Hello World!' is printed on the screen"
    - **Bad** expected outcomes:
      - "The expected number is returned"
- **Optional** components:
  - Id/name
  - Description/purpose
  - Reference to requirement/ part of specification
  - Preconditions
  - Initialization
  - More: see IEEE 829

- **Typically...**
  - Id/name
  - Description/purpose
  - Precondition
  - Initialization
  - Action
  - Expected outcome

Example: Test case for CD player

Id: CDP-Vol-1

Purpose: checking volume control

Precondition: Music is playing and heard in speakers

- **Action a**: Turn volume control, try both directions
- **Expected outcome**: Turning volume control clockwise increases volume of music, counterclockwise decreases volume
- **Action b**: Turn volume control counterclockwise as far as possible
- **Expected outcome**: Music playback completely silent at stop
- **Action c**: Turn volume control slightly clockwise
- **Expected outcome**: Music is audible within 3mm of silent position

Test cases and specifications

- A test case verifies fulfilment of some particular aspect of the specification

- Test cases are usually (initially) derived from the specification
  - Coverage techniques help to direct focus

- A bunch of test cases is **not** a specification

Software specification

- What is the software supposed to do?
- Functional and non-functional properties
  - Functional: Specific behaviour of system
    - If user does X, then system does Y
    - Normally only refers to interface of the component being specified
  - Non-functional
    - Other properties, such as:
      - Efficiency
      - Usability
      - Performance
      - Coding standards
      - Licensing

Different kinds of specifications

Informal specifications:

- Descriptive text

(Semi-) formal specifications:

- UML-diagrams
  - Behavior diagrams (use-case diagrams, state diagrams)
  - Interaction diagrams (sequence diagrams, communication diagrams)

Formal specifications:

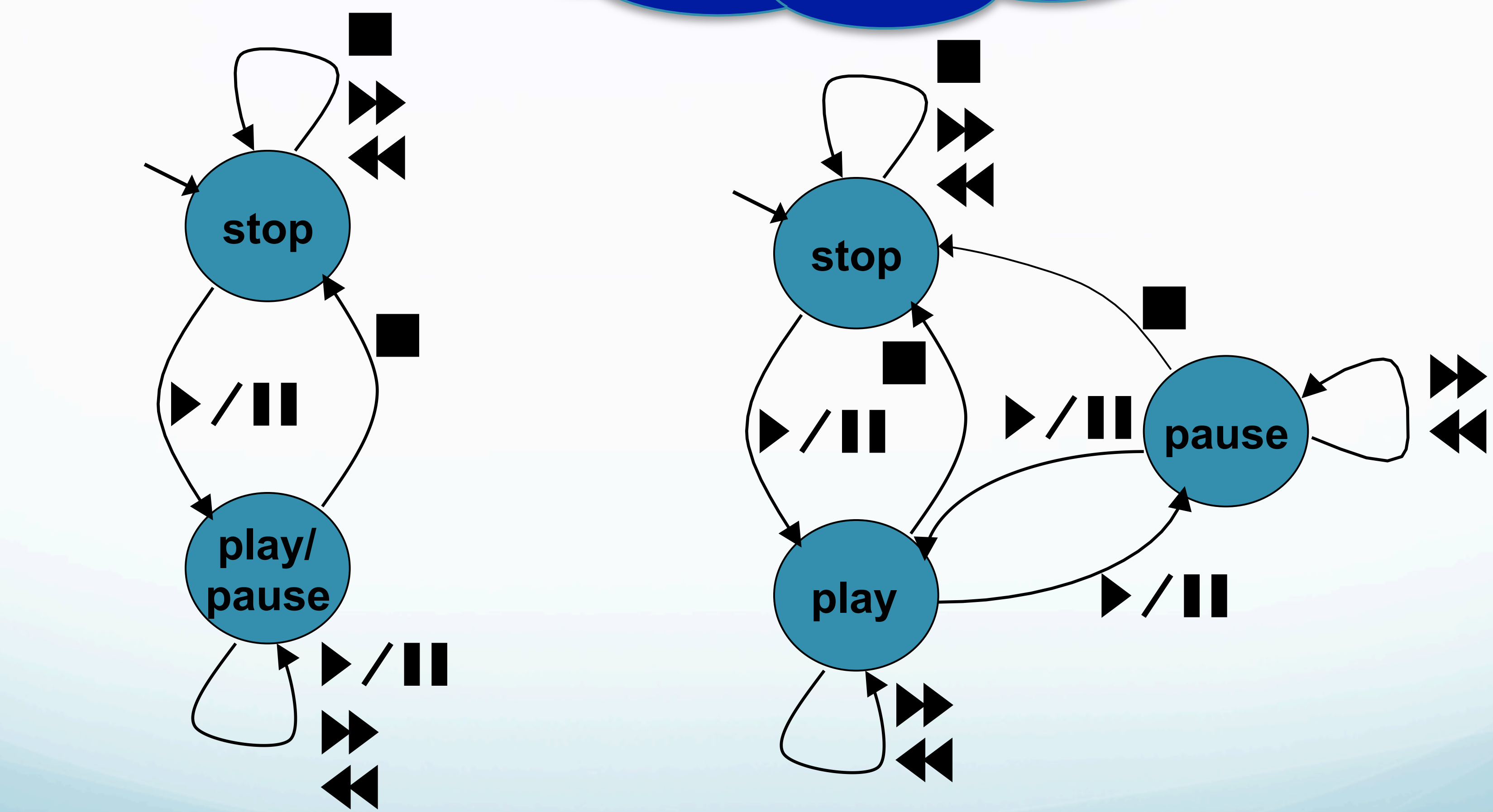
- State-machine models
  - Finite-State Machines (FSM)
  - Extended Finite-State Machines (EFSM)
- Algebraic specifications
  - Equations relating functions to each other
- Temporal logic

Finite State Machines (FSM)

- Powerful description mechanism
- Variants of FMSs are used in various places (some UML diagrams, formal specification languages, …)
- Consists of
  - A finite number of states
  - Transitions between the states
  - (Very often we also use labels on states/transitions)

Example: FSM of a CD-player

Is it possible to obtain a more precise FSM?



CD-player FSMs

- The first version is a "***model***" of the second
  - Alternative (better) terminology: **abstraction**
  - Warning! Depends who you ask and how "precise" you want to be: abstraction and model are not the same!
  - A model is an abstract representation of the reality
    - It provides a simpler view of a property/system
    - In some cases information is lost, in others is not
      - First FSM adequate for "motor running"
      - First FSM **not** adequate for "music is playing"
  - Abstraction is the process of taking away characteristics from something in order to reduce it to a set of essential ones
    - It typically only retains information which is relevant for a particular purpose
    - There is a lost of information
    - Ex: "Odd-Even" is an abstraction of the natural numbers
- Both FSM versions are models of the whole (real) player
  - Do not include e.g. current track and position of laser pickup

Manual test case (example)

Name: CD Play 1

Purpose: Checking basic playback

Preparation: Turn on CD-player with CD in tray

- Action: Press Play/Pause
- Expected outcome: CD starts playing

More manual test cases (example)

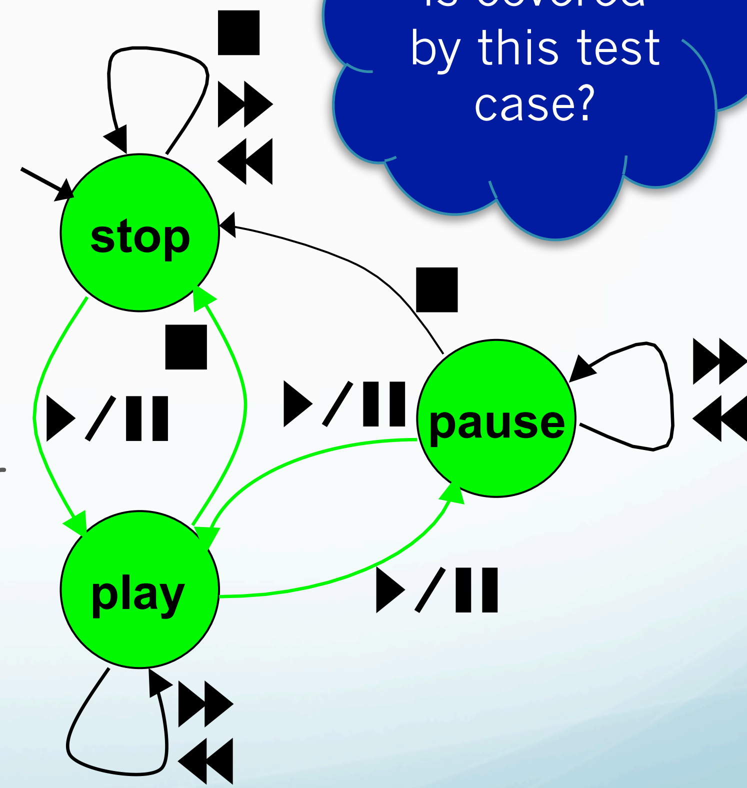
Name: CD Play 2

Purpose: Checking playing, pausing, and stopping

Preparation: Turn on CD-player with CD in tray

- Actions:
  - a: Press Play/Pause
    - Expected outcome: CD starts playing
  - b: Press Play/Pause
    - Expected outcome: CD stops playing
  - c: Press Play/Pause
    - Expected outcome: CD starts playing where it stopped
  - d: Press Stop
    - Expected outcome: CD stops playing
  - e: Press Play/pause
    - Expected outcome: CD starts playing from beginning of first track

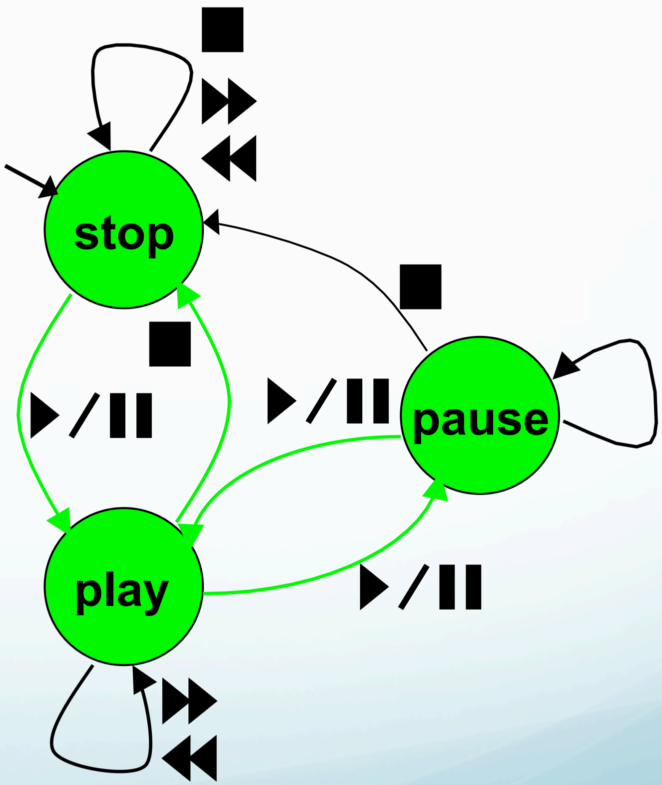
How much is covered by this test case?



More manual test cases (example)

Specification coverage:

- Covered states:
  - 3 of 3
- Covered transitions:
  - 4 of 12
    - Some transitions superpositioned in figure
- Uncovered transitions can give a hint of missing test cases



FSMs: To think about…

- About transitions
  - One transition out of each node for each possible event
  - Some transitions missing
    - Cannot happen
    - Error if happens
    - Ignored if happens
  - No labels? Maybe the transition is not needed
  - Deterministic/nondeterministic
    - Deterministic FSM
      - It's in exactly one state at any time
      - Only one transition possible to take
    - Nondeterministic FSM
      - Several states may be active at a time
      - Several transitions may be enabled under same input

Random testing against FSMs

- Representing an FSM (Implementation)
  - Set of states (e.g. enum type, bounded integers)
    - One initial state
  - Set of events
  - Transition function: State -> Event -> State
- Useful functions for verifying against FSM
  - Precondition: State -> Event -> Bool
    - Which events can happen now?
  - Postcondition (expected outcome):

    State -> Event -> SystemState -> Bool
    - Test that the actual system is in a correct state after the transition
    - Looks at the actual system to test
  - Generate events randomly
    - Make sure they respect precondition
- You can implement this in your favourite test framework
- …or get QuickCheck, which does it automatically

Model-based verification

- Writing test cases can be a very tedious task
- State transition systems can be used to automate test case generation (later in this course)
  - Model-based testing
- Advanced tools that automatically check whether a system is modelled by a given FSM
  - Model checking

Extended Finite-State Machines

- Finite-State Machines have concrete state spaces
- Extended Finite-State machines (EFSMs)
  - State space represented by structures like integers, lists, tuples, strings, and enumeration types
  - May have infinite state space
- Random testing against EFSMs:
  - Just as for FSMs

Example EFSM: CD player

- State representation: **(S,T)**
  - S: Playing state (stopped, playing, paused)
  - T: Track number
- Initial state:
  - (stopped, 1)
- Events:
  - Play/pause pressed
  - Stop pressed
  - Skip forwards pressed
  - Skip backwards pressed
  - End of track reached (eot)
  - End of disc reached (eod)

CD player preconditions

- Almost all events are possible at all times
  - Buttons can be pressed at any time
  - But *eot* and *eod* can only happen during playback

- *precondition((stopped, t), eot)*
  - Is it possible to be in a state where the CD player is not playing and it is "reading" any track t and "react" to the eond-of-track event?

precondition((stopped, t), eot) -> false

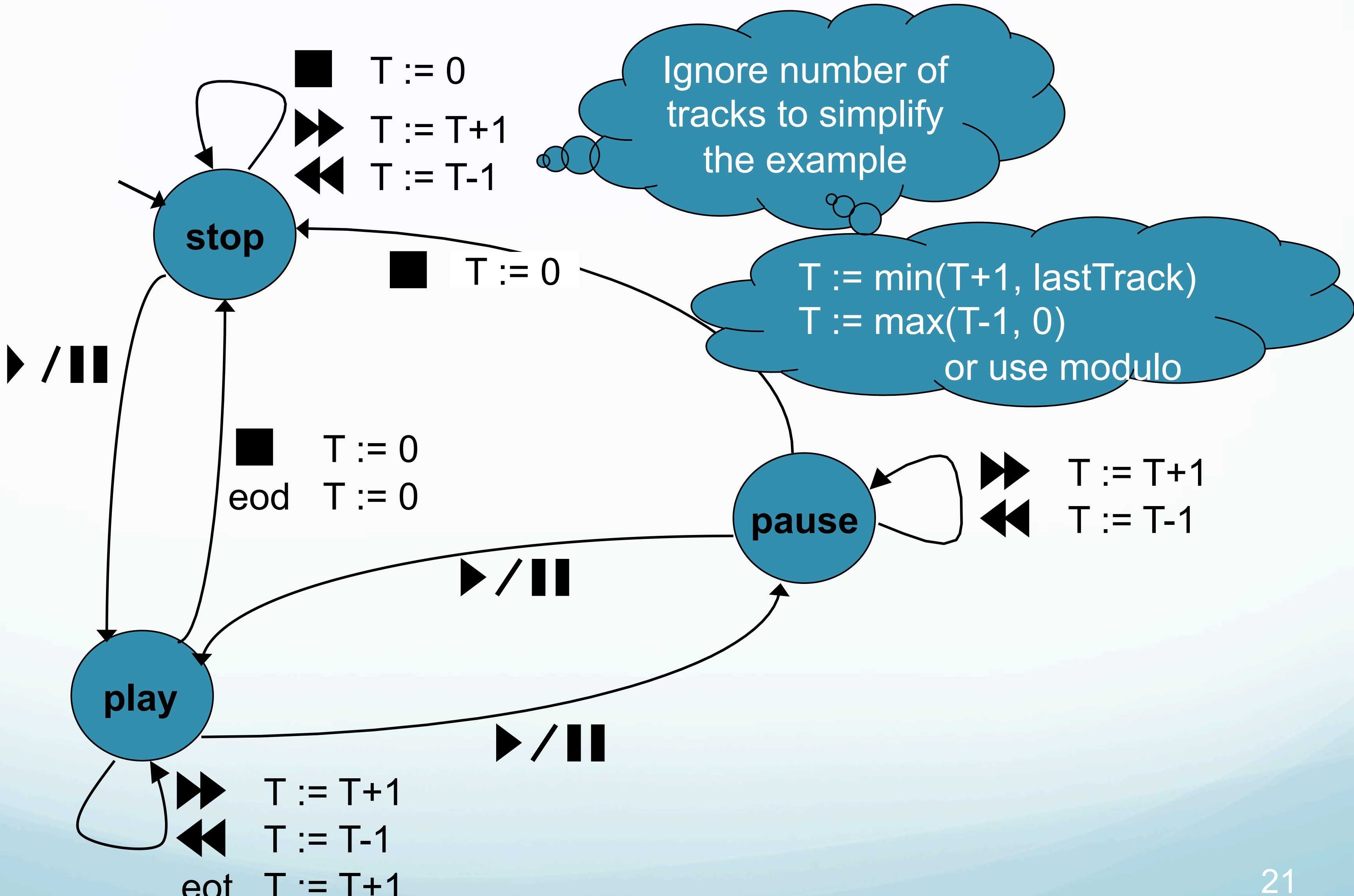
precondition((paused, t), eot) -> false

precondition((stopped, t), eod) -> false

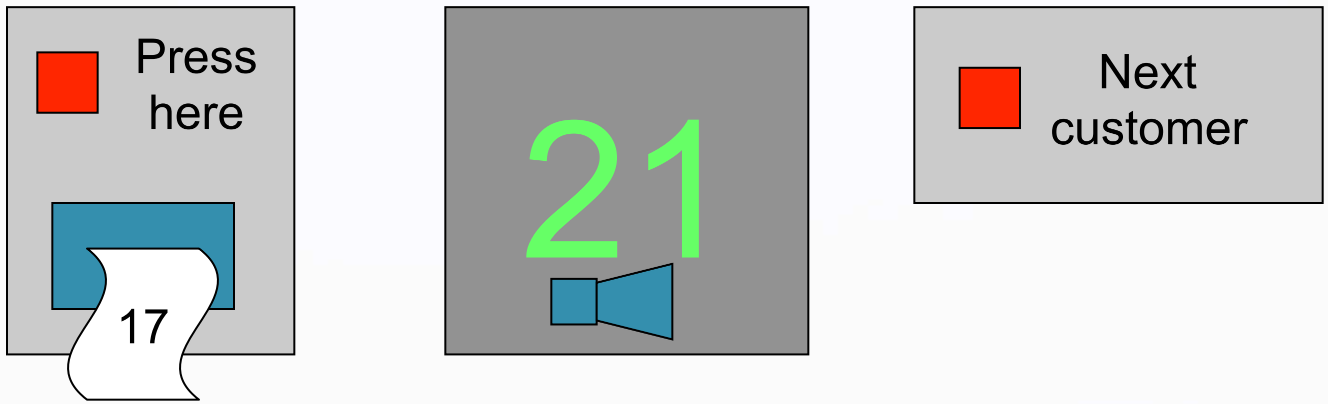
precondition((paused, t), eod) -> false

precondition((s,t), e) -> true
(for any other state s, track t, any other event e may happen?)

CD Player transitions

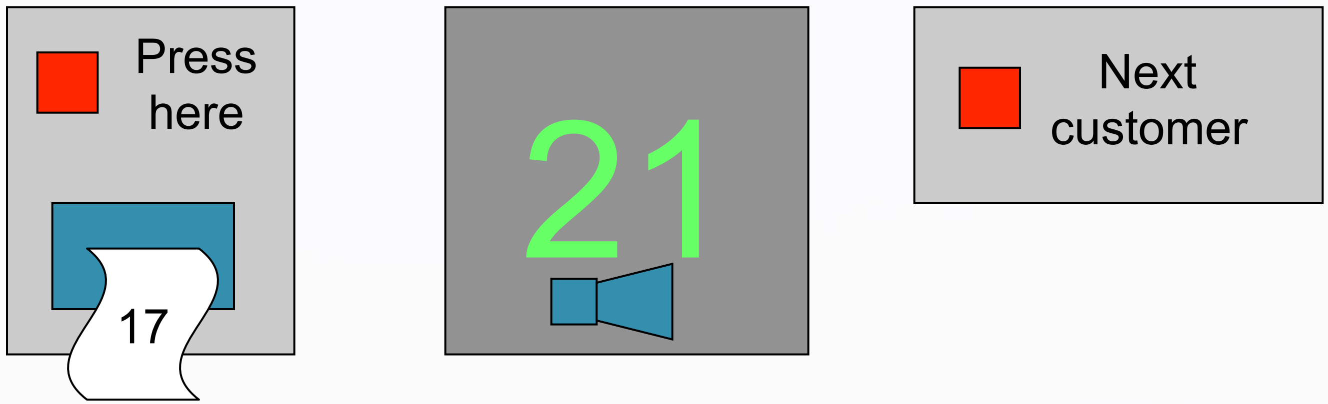


Turn ticket system



- Three units:
  - Customer ticket terminal
    - Button B1 (”press for ticket”)
    - Ticket printer
  - Number display
    - Number display D
    - Speaker S
  - Attendant terminal
    - Button B2 (”next customer”)

Turn ticket system



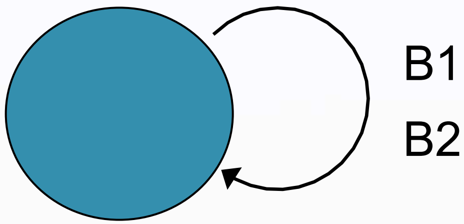
Informal specification

- At entrance, customer presses B1, which causes a ticket to be printed. The first ticket has number 1, the number is increased by 1 for successive tickets. Ticket 999 is followed by 0.
- When attendant presses B2
  - If receipts have been printed with higher number than the currently displayed, the display number is increased.
  - If no such receipts have been printed, nothing happens.
- D initially displays the number 0. Display increments adds 1 to the current number, unless the current number is 999 in which case the new number is 0. Each increase is accompanied by a sound from S.

Group exercise

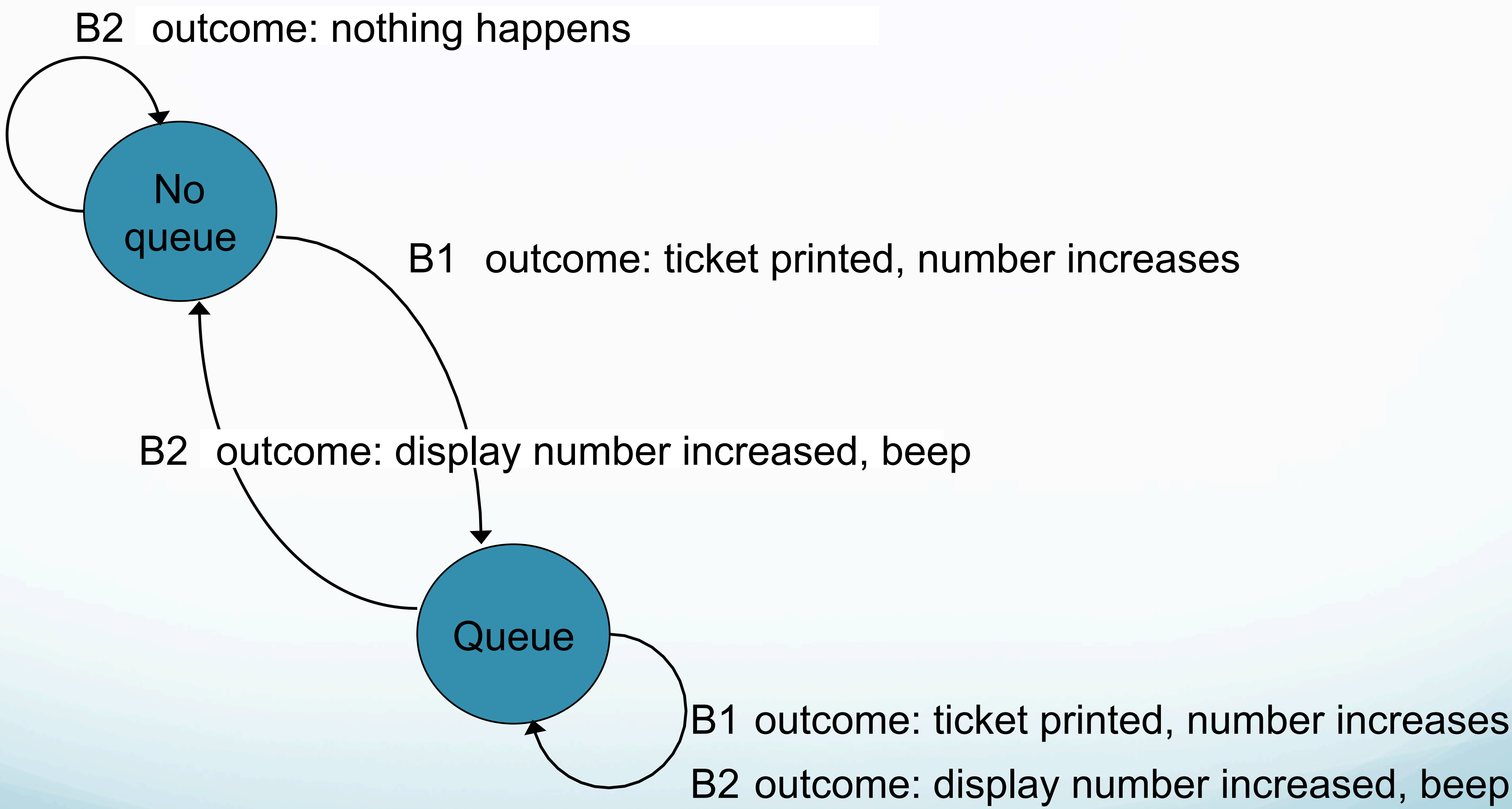
- Come up with a **finite-state machine** that models this system
  - Many different variants exist with different levels of complexity and accuracy
- What are the limitations of your model?
- Come up with an **extended finite state machine** that models the system more accurately
  - Perhaps not based on the FSM, as the CD player was

The simplest FSM representation of all



- This FSM models <u>all</u> systems with two events
- Hence, it is useless!

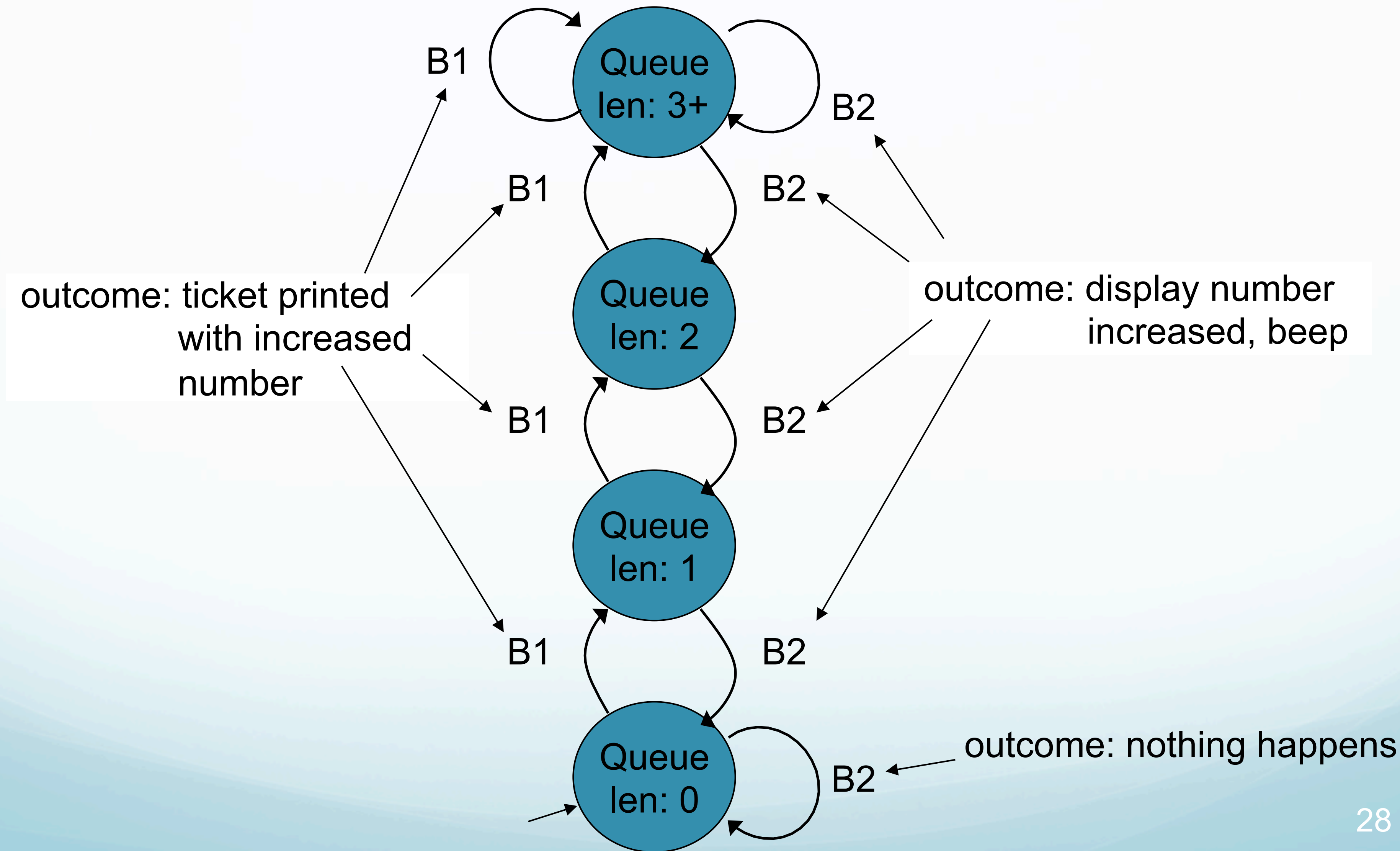
FSM representation
(very simple)



Properties of this FSM

- Simple! (easy to understand, easy to come up with)
- Nondeterministic
  - (Why?)
- Can determine:
  - Correct behaviour of one B2 press after a B1
- Cannot determine:
  - Correct behaviour of multiple B2 presses
  - Exact numbers on tickets and display
- Useful for:
  - Understanding
  - Writing some testcases
  - Automatically prove some properties

FSM representation
(bounded queue)

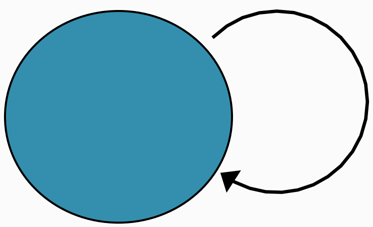


Properties of this FSM

- Still rather simple
- Still nondeterministic (in top state)
- Can determine
  - Correct behaviour as long as not more than 2 people in queue
- Cannot determine
  - Correct behaviour with more than 2 people in queue
  - Exact numbers on tickets and display
- Useful for
  - Understanding
  - Writing more testcases? (probably not more than first)
  - More accurate automated testing

EFSM representation 1

- State:
  - Number of people in queue (n)



```
B1  n := n+1
B2  if n >= 1 then
        n := n-1;
    else
        null;
    fi
```

> Expected outcome: ticked printed with increased number (B1)

> Expected outcome: Display number increased, people in the queue, beep heard (n := n-1;)

> Expected outcome: nothing happens (null;)

Properties of this EFSM

- Simple?
- Deterministic!
- Can determine
  - How to act (whether to increase display number)
- Cannot determine
  - Exact numbers on tickets and display
- Useful for
  - Understanding?
  - Even more accurate automated testing

EFSM representation 2

- State:
  - Last printed number (P), currently displayed number (Q)
  - Initial state: P=0, Q=0

> We'll say that variables in expected outcome refer to new value

- Next state function:
  - B1: P := P+1 mod 1000
    - Expected outcome: ticket printed with number P
  - B2: if P=Q then

```
          null;   // expected outcome: nothing happens
       else
          Q=Q+1 mod 1000;
               // expected outcome: D displays Q, beep
```

- Precondition function:
  - Anything is possible

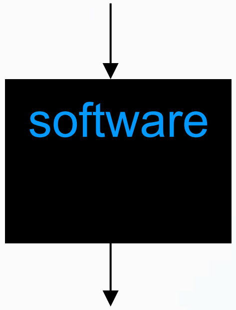
Properties of this EFSM

- Somewhat harder to comprehend
- Deterministic
- Can determine
  - Full behaviour of system
- Cannot determine
  - Nothing
- Useful for
  - Complete verification of system
  - Starting point of implementation

Black box testing

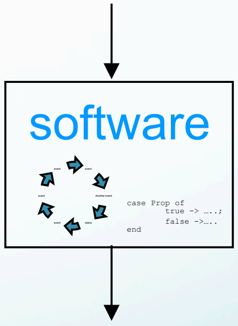
Black box and white box testing

Black box testing: Test tactic in which the test object is addressed as a box one **cannot** open.



A test is performed by sending an **input value** and observing the **output** *without using any knowledge about the test object internals.*

White box testing: Test tactic in which the test object is addressed as a box one **can** open.



A test is performed by sending an **input value** and observing the **output** and internals while explicitly *using knowledge about the test object internals.*

Black Box testing

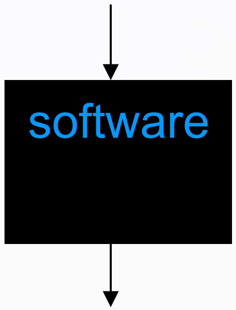
Techniques

- Random Testing
- Equivalence Class Partitioning
- Boundary Value Analysis
- Cause and Effect Graphing
- State Transition Testing
- Error Guessing
- Use case testing
- … …

> Techniques tell you how to select the inputs:
> How to create a test case

Misleading…

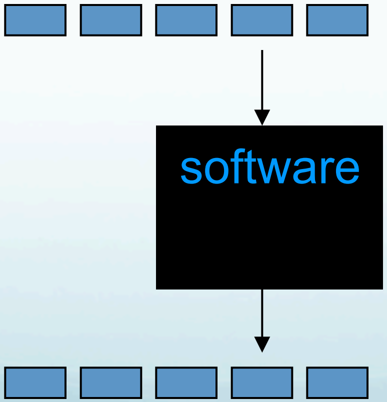
Black box testing is often depicted as:



which might be misleading….

It suggests that given an input, the output can be checked against an expected output.

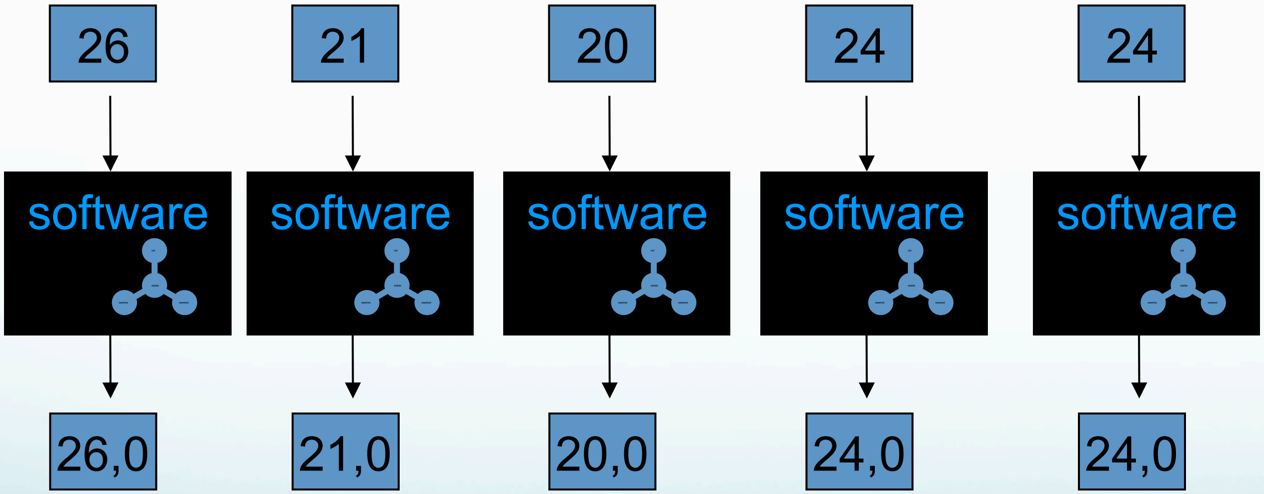
In case the test object has memory, the expected output **depends on the history**!



“Box” has state

Testcase: Start the test object, send 1 value;

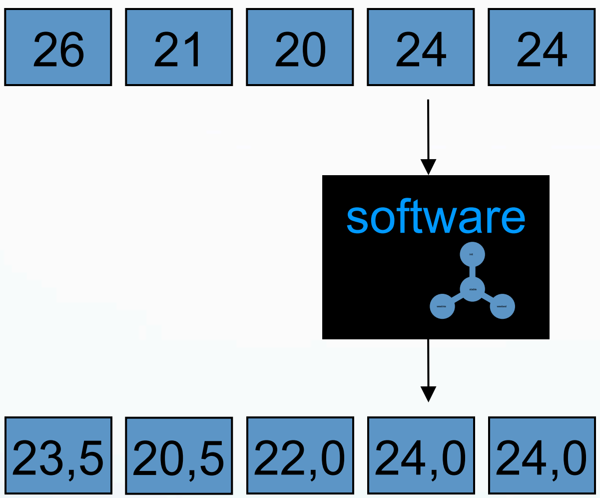
Execute 5 test cases



"Box" has state

Testcase: Start the test object, send 5 values;

Execute 1 test case

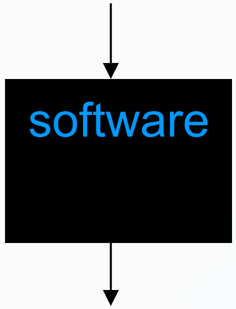


Specification:

Return average temperature over today and yesterday

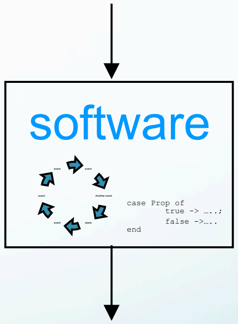
Before execution of a test case, the test object has to be brought into a known state. From there, as many as possible other states should be reached by different test cases.

Black box and white box testing



Black box testing: Test tactic in which the test object is addressed as a box one **cannot** open.

A test is performed by sending ***a sequence of input values*** and observing the **output** without using any knowledge about the test object internals.



White box testing: Test tactic in which the test object is addressed as a box one **can** open.

A test is performed by sending a ***sequence of input values*** and observing the **output** and internals while explicitly using knowledge about the test object internals.

Creating test cases

- Look at the specification, find different cases
  - Enumerate statements in specification, make sure that each aspect is covered by at least one test case
  - Different representations like FSMs can help
  - Go through use cases, refine them into test cases
    - Use case: idealized idea about interaction with system
    - Test case: detailed description using the system interface

- Equivalence class partitioning
  - In order to reduce number of test cases
  - Find classes of situations where behaviour should be similar
  - FSM specifications really useful

Creating test cases

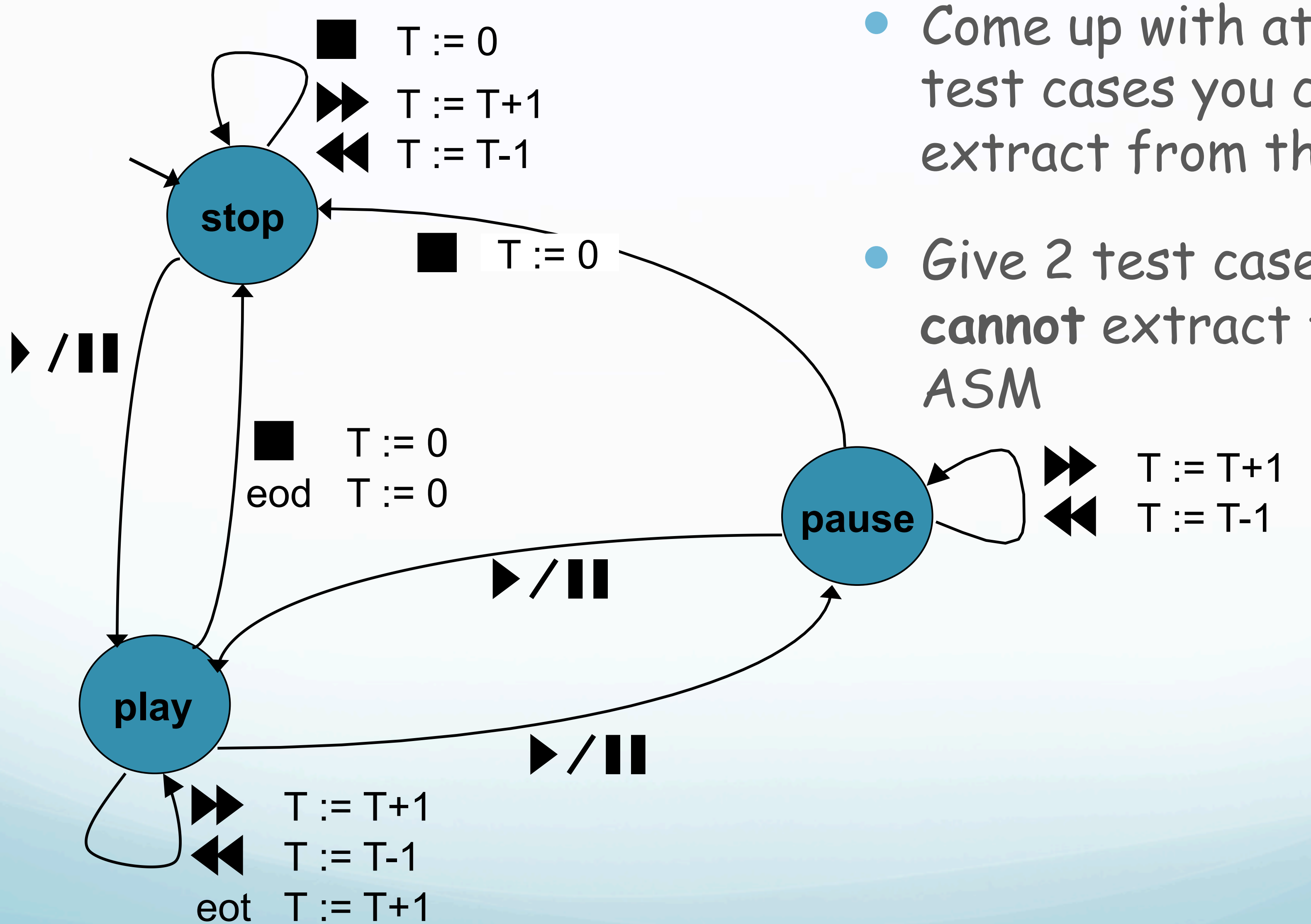
- Boundary values
  - Malfunctions often occur around boundaries
  - Are there boundaries? Try out values close to limits
    - Maxint? (should system work correctly for maxint?)
    - ATM: What if user wants to withdraw exactly all money from account?
    - CD: Make sure that last track on CD can be played
  - Disadvantage: identifying boundaries may require knowledge about code
    - … but once one knows about the boundaries, the test case can be written without reference to internals
- Error guessing
  - Similar to boundary values: are there values that seem especially dangerous?
    - How about if the century actually ends, so year = 00 ?

Creating test cases

- Random testing
  - Write general test cases
    - Precondition: Account contains m SEK, n <= m
    - Action: User withdraws n SEK from account
    - Expected outcome: New balance is m-n SEK.
  - Generate random sequences of test cases
  - …but try to create sequences that make sense (not much worth if 99.9% of your tests end up in expected failures)

- Code coverage analysis (next lecture)

Group exercise



- Come up with at least 2 test cases you can extract from the EFSM
- Give 2 test cases you **cannot** extract from the ASM

Group exercise

- Examples of test cases you **can** get from the EFSM
  - When the CD player is playing, after pressing ”stop” the player stops
  - When the CD player reaches an ”eot” it changes track
  - When pressing ”pause” and then ”play” (without pressing anything else in between), the track doesn’t change

- Examples of test cases you **cannot** get from the EFSM
  - Cannot test what happens when pressing ”pause/play” and ”stop” at the same time
  - While in ”pause” we cannot test what happens when we press the ”forward” button till the ”eod”

Assignment 1

- You are given a library implementing a simple calculator, and an informal specification
- Come up with an FSM or EFSM that models the calculator (to some interesting level of detail)
- Come up with black box test cases that verifies the library against the spec, using JUnit.
- You don't need to fix any bugs
  - Though good exercise for you if you do fix the bug