

# Test models: State machines

Błażej Pietrzak

[blazej.pietrzak@cs.put.poznan.pl](mailto:blazej.pietrzak@cs.put.poznan.pl)

# Basic State Machine Model

## What is a State Machine?

A system model composed of events (inputs), states and actions (outputs):


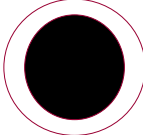
- Output is determined by both current input and past inputs
- An information concerning past inputs is represented by states

*Robert V. Binder: Testing Object-Oriented Systems. Models, Patterns and Tools, Addison-Wesley 2000*

# Basic State Machine Model

## State

The Information concerning past inputs

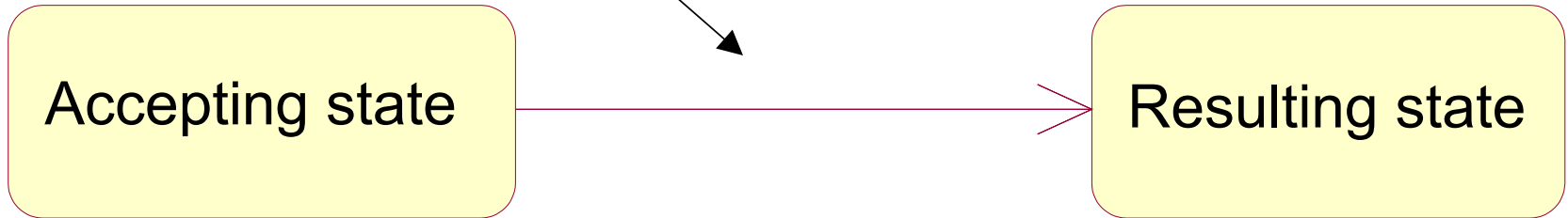
-  Initial State – first event is accepted, not need to be entered by a transition
-  Final State – stop accepting events

Machine may be in only one state at a time

# Basic State Machine Model

## Transition

An allowable two-state sequence



# Basic State Machine Model

## Event

An input or interval of time

## Action

The result or output that follows an event

# A Mealy machine

- A transition may have an output action
- Any output action may be used in more than one transition
- No output action is associated with state, state is passive

# Properties of Finite State Automata

- Incomplete Specification
- Equivalent (A minimal state machine)
- Reachability analysis
- Nonreachable states
  - Dead state – no paths lead out of a dead state
  - Dead loop – no states outside of the loop may be reached
  - Magic state – has no inbound transition but provides transitions to other states

## Guarded Transition

A guarded transition cannot fire unless

- The system is in accepting state for this transition
- The guarded event occurs
- The guarded predicate evaluates to true

A transition without an explicit guard: [TRUE]

An event not accepted in a certain state: [FALSE]

*Event-name arg-list '[' guard-condition ']' '/'  
action-expression*



# Limitations of the Basic Model

- Not specific for object oriented systems
  - An OO interpretation of generic events, actions and states is needed
- Limited scalability
  - Up to 20 states on diagram is readable
  - Basic state models do not scale well  
*No Partitioning, hierarchic abstraction*
- Concurrency cannot be modeled
  - The basic model cannot accomodate two or more simultaneous transition concurrency

# State Machines and OO Development

- ✓ If .. else, switch statements
- ✓ Many design patterns in GOF collection:  
State, Chain of Responsibility, Interpreter,  
Mediator
- ✓ Other patterns

# Testable model

- Complete and accurate reflection of implementations to be tested
- Preserve details essential to fault detection and conformance demonstration
- Represents all events to the IUT must respond
- Represents all actions
- Unambiguous and testable definition of state in a way that checking the resulting state can be automated

# The FREE State Model

- Flattened Regular Expression (FREE) state model
  - The class under test is flattened to represent the behavior of inherited features
  - Behavior using definitions of state, event, and action that support the development of effective test suites
- Can be expressed using the UML
- Object state at method activation and deactivation is the focus of the FREE state model

# FREE state model

## State

- Object state - Current contents of an object
- State invariant reports the state
- Aggregate state – subset of combinational value set
- Abstract state – state is visible to clients by its interface not implementation

```
class TwoBits {  
    private boolean bit1;  
    private boolean bit2;  
    boolean isTrue() { return bit1 && bit2; }  
}
```

How many states does TwoBits have?

# Checking the resultant state

## State Reporter methods

Evaluates the state invariant and returns a Boolean variable

```
boolean isGameStarted() { return ... }
```

```
public void testIsGameStarted() {  
    app.startGame();  
    assertTrue(app.isGameStarted());  
}
```

# Checking the resultant state

## Test Repetition

State Reported methods cannot be implemented.

Some corrupted states can be detected.

Save output action, repeat the test sequence and compare the actions.

*Bezier B.: Black box testing. Wiley & Sons, 1995*

```
public void testPlay() {
    app.player1Hits();
    int firstWinner = app.getWinner();
    assertEquals(PLAYER_1, firstWinner);
    app.player2Hits();
    int secondWinner = app.getWinner();
    assertEquals(PLAYER_2, secondWinner);

    app.player1Hits();
    assertEquals(firstWinner, app.getWinner());
    app.player2Hits();
    assertEquals(secondWinner, app.getWinner());
}
```

# FREE state model – cont.

Hybrids are not welcome!

Use only one representation  
(e.g. Mealy machine)



# FREE state model Transitions

- Transition is a unique combination of
  - State invariants
  - An associated event
  - Optional guard expressions
- Event is either
  - A message sent to the class under test
  - A response received from a server of the class under test
  - An interrupt or similar external control action that must be accepted by the CUT

# FREE state model

## Transitions – cont.

- Guard is
  - A predicate expression associated with an event
- Action is either
  - A message sent to a server object
  - The response provided by an object of the class under test to its client

## FREE state model

### Alpha and Omega states

- The  $\alpha$  (alpha) state
  - A null state representing the declaration for an object before its construction
  - Differs from an initial state
  - An alpha transition
- The  $\omega$  (omega) state
  - It is reached after an object has been destructed or deleted, or has gone out of scope
  - It may differ from an explicitly modeled final state
  - It is reached by explicit or implicit destructors

# FREE state model

## Inheritance and Class Flattening

- To obtain the model for subclasses, the class hierarchy is flattened
- Statecharts are well suited to representing a flattened view of behavior

## FREE state model

### Inheritance and Class Flattening – cont.

The type substitutability principles

- A superclass state can be partitioned into substates
- A subclass may define new states, but these must be orthogonal to the superclass states
- The effect of superclass private variables on the superclass state space must be additive

# FREE state model

## Inheritance and Class Flattening – cont.

### Substitutability Rule

- Orthogonal composition
- Concatenation
- State partitioning and substate addition
- Transition retargeting
- Transition splitting

# FREE state model

## Inheritance and Class Flattening – cont.

The flattened transition diagram shows the complete behavior of the class under test.

Class flattening is not always necessary:

- When only test within subclass boundaries without checking inherited properties
- Not enough information about superclass
- Lack of time

## Free state model

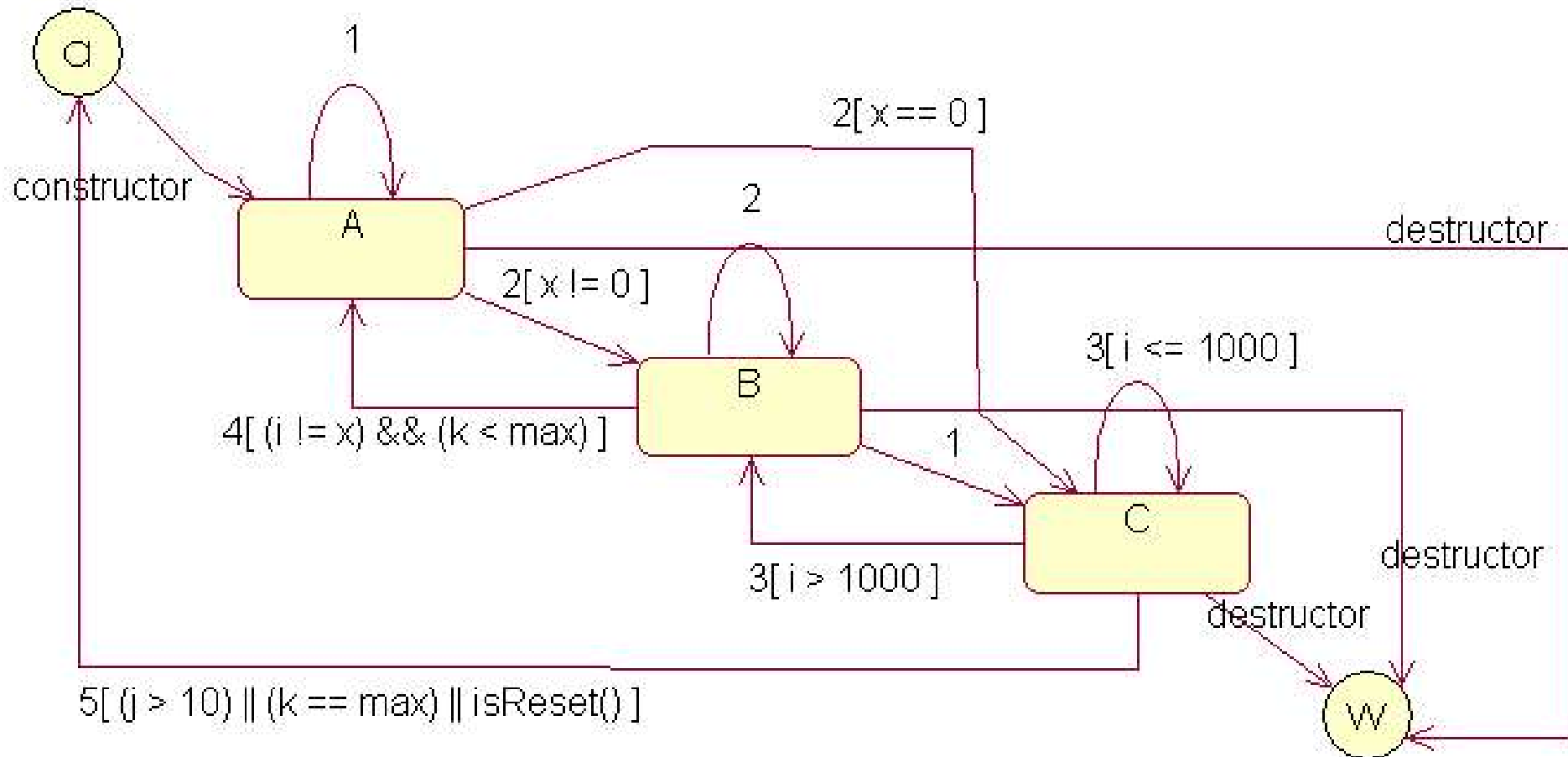
### Unspecified Event/State pairs

- May mean that implicit transition is illegal, ignored, or „impossible”
- May also be a specification omission
- An illegal event is an otherwise valid event (message) that should not be accepted given the current state of the UIT
- A [sneak path](#) is the bug that allows an illegal transition or eludes a guard
- Complete behavior testing should exercise both explicit and implicit behavior



# FREE state model

## Response matrix



# FREE state model

## Response matrix – cont.

Events and guards				Accepting state – expected output					
				a	A	B	...	C	$\omega$
constructor				✓	6	6	6	6	6
Event 1				x	✓	✓	...	2	6
Event 2	x = 0								
	DC			x	x	✓	...	2	6
	False			x	✓	x	...	x	x
	True			x	✓	x	...	x	x
Event 3	i <= 1000								
	NI			x	2	2	...	x	6
	False			x	2	2	...	✓	x
	True			x	2	2	...	✓	x

# FREE state model

## Response matrix – cont.

Events and guards				Accepting state – expected output					
				a	A	B	...	C	$\omega$
Event 4	i != x	k < max							
	DC	DC		X	2	X	...	2	6
	False	False		X	X	1	...	X	X
	False	Ture		X	X	2	...	X	X
	True	False		X	X	2	...	X	X
	True	True		X	X	✓	...	X	X

# FREE state model

## Response matrix – cont.

Events and guards				Accepting state – expected output					
				a	A	B	...	C	$\omega$
Event 5	$i > 10$	$k = \max$	isReset()						
	DC	DC	DC	X	2	5	...	X	6
	False	False	False	X	X	X	...	5	X
	False	False	True	X	X	X	...	✓	X
	False	True	False	X	X	X	...	✓	X
	False	True	True	X	X	X	...	✓	X
	True	False	False	X	X	X	...	✓	X
	True	False	True	X	X	X	...	✓	X
	True	True	False	X	X	X	...	✓	X
	True	True	True	X	X	X	...	✓	X

# FREE state model

## Response matrix – cont.

Events and guards				Accepting state – expected output					
				a	A	B	...	C	$\omega$
destructor				X	✓	✓	...	✓	2

DC – don't care

X – excluded

✓ - explicit transition

# FREE state model

## Response matrix – cont.

Response codes for illegal states

Resp. code	Name	Response
0	Acceptance	Execute explicit transition
1	Queue	Place illegal event in queue for calculation and omitting
2	Ignored	Do nothing
3	Marker	Return non-zero error code
4	Rejection	Throw IllegalEventException
5	Silence	Turn off the source of event
6	Damage stopping	Go to damage stopping routine (e.g. Memory dump) and stop the process

# Responses to Illegal Events

- An appropriate error message or exception should be produced
- The abstract state of the object should remain unchanged after rejecting the illegal event
- Prefer defensive school – easier to test

# Fault Model

- Control Faults
  - Missing or incorrect transition (resulting state is bad, but not corrupted)
  - Missing or incorrect event (permitted message is neglected)
  - Missing or incorrect action (upon transition bad things happen)
  - Extra, missing, or corrupt state (behavior is unpredictable)



# Fault Model – cont.

- Control Faults – cont.
  - A sneak path (a message is accepted when it should not)
  - Illegal message failure (unexpected message cause damage)
  - Trap door (implementation accepts undefined messages)

## Fault Model – cont.

- Incorrect composite behavior
  - Missing or incorrect overriding method in subclass
  - Subclass state extension conflicts with superclass state
  - Subclass fails to retarget superclass transition
  - Guard evaluation in subclass produces a new side effect
  - Guard parameters are bound to the wrong subclass or superclass methods

# Developing a Test Model

1. Create a testable state model (e.g. FREE state model)
2. Validate state model
  - Structure
  - State names
  - Guarded transitions
  - Well-formed subclass behavior
  - Robustness

## Test generation strategies

### Strategies:

- Exhaustive – preferred
- N+ strategy – preferred
  - Number of tests about  $k^2n / 2$ , where  $k$  is states no.,  $n$  is events no.
- All transitions – minimum
  - at least once all states, all events, and all actions
  - Although detects bad or omitted event-action pairs, it cannot be proved that bad state occurred.
  - If state machine is incomplete cannot detect sneak paths
  - Number of tests =  $k \times n$ , where  $k$  is states no.,  $n$  is events no.
- Piecewise (*pol. sztukowanie*) – not recommended

# N+ strategy

- Advanced state-based testing
- UML state models
- Testing considerations unique to OO implementations
- It uses a flattened state model
- All implicit transitions are exercised to reveal sneak paths
- The implementation must have a trusted ability to report the resultant state

# Steps for N+ Test Suite

1. Develop a testable (e.g. FREE) model of the implementation under test
  - Validate the model using the checklists
  - Expand the statechart
  - Develop a response matrix
2. Generate the round-trip path test cases
3. Generate the sneak path test cases
4. Sensitize the transitions in each test case

# Steps for N+ Test Suite

## An example

FREE model of the implementation  
under test

- Flattened transition diagram
- Response matrix

# Steps for N+ Test Suite

## An example

FREE state model  $\rightarrow$  Transition tree

- Initial state or alpha state (if exists) is a root of the tree.
- Check the state of every not-final node – leaf in the tree and every transition going from that state. For each transition create at least one edge. Each new edge and node represent an event and resulting state reached by outgoing transition:



# Steps for N+ Test Suite

## An example

1. ...

- a) If not guarded condition create one new branch.
- b) If guard is a simple predicate or composed of only AND operators create one new branch.
- c) If guard is composed predicate or with OR operators then create new branch for each combination of values that make guard = true.

## Steps for N+ Test Suite

### An example

1. ...
  - a) If guard defines dependency that is reached after repetition of some event (e.g. [counter  $\geq$  10]) then the test sequence demands at least that number of repetitions. The transition is marked with \*
2. For each edge and node from step 2:
  1. Note event, guard and action
  2. If the state represented by new node is already represented in another node (anywhere on diagram) or is ending state then mark this node as ending node – don't make new transitions from it.

# Steps for N+ Test Suite

## An example

Transition tree -> Conformance test suite

Response matrix -> Sneak path test suite

# Literature

- Robert V. Binder: „Testowanie systemów obiektowych. Modele wzorce i narzędzia.” WNT 2003
- Bezier B.: „Black box testing.” Wiley & Sons, 1995
- E. Gamma, R. Helm, R. Johnson, J. Vlissides: „Design Patterns - Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1995

# Quality Assessment

Thank You for your attention 😊

- What is your general impression (1-6)
- Was it too slow or too fast?
- What important did you learn during the lecture?
- What to improve and how?

