Class Scope Testing

João Pereira ©

Class Scope Testing

- Intraclass visibility contributes to errors just as global storage does in procedural languages

- Purpose
  - Test the interactions of the methods in the class

- Testing question
  - **What sequences of methods to test**?

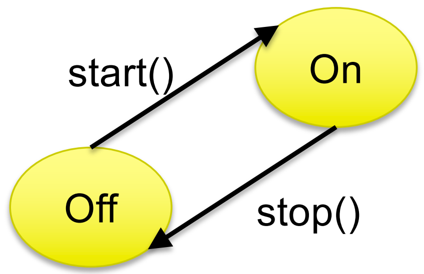
  - Answer depends on the method sequences that are allowed

Class Modalities

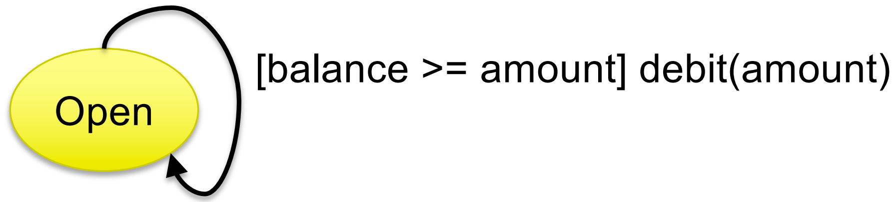
- Class modality
  - Classification of classes based on their reaction to method call sequences
- Characterization of the constraints on the method sequences
  - Domain constraints
  - Message sequence constraints
- Different kinds of faults may be introduced for different kinds of constraints
  - Require different kinds of testing strategies

Message sequence and domain constraints

- Message sequence constraints mean that the class will reject some sequences of messages based on finite states of object
  - e.g., a Timer cannot be stopped if it is not started
  - Represented by states in a finite state machine



- Domain constraints mean that the class will reject some messages depending on the current content of the object
  - e.g., a debit cannot be made on an Account with balance <= amount
  - Represented by constraints on state transitions



Types of class modality - 1

- Non-modal class
  - No constraint on the message sequences
    - An object of class *DateTime* accepts any interleaving of set/get messages.
- Uni-modal class
  - The constraints on the message sequences are independent of the content of the object
  - Constraints based solely on history
    - An object of class *TrafficSignal* accepts message *setRedLightOn* only after *setYellowLightOn, …*
- Quasi-modal class
  - The constraints on the message sequence are related to the contents of the object, but not necessarily history
    - An object of class *Stack* rejects a *push* message if the stack is full, but accept it otherwise
    - Many container and collection classes are quasi-modal

Types of Class Modality - 2

- Modal class
  - The constraints are related to both history and content of the object
    - An object of class *Account* does not accept a withdrawal message if balance is <= amount (domain constraint);
    - A message to freeze the object is accepted if account is not closed and not frozen (sequence constraint)
- Modal vs Quasi-modal
  - Depend on content vs depend on parameters indirectly related to content
- Classes that represent problem domain entities are often modal

State Constraint		**Class Modality**	
Domain	**Sequence**	**Type**	**Example**
No	No	Non-modal	DateTime
No	Yes	Uni-modal	TrafficLight
Yes	No	Quasi-modal	Stack
Yes	Yes	Modal	Account

Class Scope Test Patterns

- Invariant Boundaries
  - Identify test cases for complex domains
- Nonmodal Class Test
  - Design a test suite for a class without sequential constraints
- Modal Class Test
  - State-based testing
- Quasi-modal Class Test
  - State-based testing

Invariant Boundaries

- Intent
  - Select test-efficient test value combinations for classes, interfaces, and components composed of complex and primitive data types
- Context
  - How to model relationships among variables to support efficient and effective selection of test values?
    - The valid and invalid combinations of instance variable values may be specified by the class invariant
    - The class invariant typically refers to instance variables that are instances of primitive and complex data type
  - May be applied at any scope for which an invariant can be written
  - The Invariant Boundaries pattern does not consider input/output relationship or message sequence

Fault model

- Bugs in implementation of constraints needed to define and enforce a domain formed by several complex boundaries
- Domain testing will find such bugs **if**

  the correct boundary is known to the test designer

Strategy – Test model

1. Define the class invariant, **responsibility-based** assertions
2. Develop *on* points and *off* points for each condition in the invariant using the 1x1 selection criteria of domain model
3. Complete the test suite by developing *in* points for the variables not referenced in a condition
4. Represent the results in a domain matrix

Invariant Boundaries example - 1

```
class ClientProfile {
  Account account = new Account();
  Money creditLimit = new Money();
  short trCounter;
  ...
}
```

Money:
scalar type with two precision digits

Account abstract states:
- open: balance >= 0, inactive <= 499, !isClosed
- debit: balance < 0, inactive <= 499, !isClosed
- close: isClosed, balance = 0
- idle: inactive >= 500, !isClosed

Invariant Boundaries Example - 2

- Class Invariant
  - assert (trCounter >= 0) && (trCounter <= 500) &&
    (creditLimit <= trCounter * 10 + 10) && !account.isClosed());

Condition		On Point	Off Point
trCounter >= 0		0	-1
trCounter <= 500		500	501
creditLimit <= trCounter * 10 + 10		2510	2510.01
!account.isClosed()	isOpen()	close (0, i <= 499, closed)	open (0, i <= 499, open)
	isIdle()	close (0, 499, closed)	idle (0, 500, open)
	isDebit()	close (0, i <= 499, closed)	debit (-0.01, i <= 499, open)

- On and Off points for creditLimit
  - consider trCounter = 250 and a two-digit precision
- The minimal number of (On, Off) pair points for condition *account.isClosed()* is 1

Invariant Boundaries Example - 3

Constraint			Test Cases									
Variable	Condition		1	2	3	4	5	6	7	8	9	10
trCounter	>=0	On	0									
		Off		-1								
	<=500	On			500							
		Off				501						
	Typical	In					250	250	100	120	303	…
creditLimit	<=trCounter*10 + 10	On					2510					
		Off						2510.01				
	Typical	In	9	-2	570	600			204	806	390	…
account	!isclosed()	On							(0, 499, closed)			
		Off								(0, 2, open)		
	(not mandatory)	Off									(-0.01, 4, open)	
	(not mandatory)	Off										(0, 500, open)
	Typical	In	idle	open	open	open	open	debit				
Expected Result			✓	✗✗	✓	✗	✓	✗	✓	✗	✓	✓

✓ - Valid ✗ - Invalid **✗✗ - Impossible**

Test cases 9 and 10 are not mandatory

Automation

- May be automated, given sufficiently detailed variable domain definitions
- *In* points may be generated (or selected) by a random generation approach
- Several commercial tools are available

Entry and Exit Criteria

- Entry Criteria
  - A **validated invariant exists** or can be developed for the IUT
  - If this pattern is used with another pattern, the entry criteria for that pattern will also apply
- Exit Criteria
  - A complete set of domain tests has been developed
  - The coverage considerations of the other pattern will apply

Consequences

- All domain bugs should be revealed
- Does not provide extensive checking of input/output correctness, control logic and sequencial constraints
- Developping the invariant can be time consuming

Non-modal Class Test

- Intent
  - Develop a class scope test suite for a class that does not constrain message sequences.
- Context
  - Non-modal classes are often said to "*accept any message in any state*". Any operation may follow any other, excluding construction and destruction.
    - Does not constrain interleaving modifier and accessor messages.
  - Classes that implement basic data types are often non-modal
  - A non-modal class imposes few constraints on message sequence but usually has a complex state space and a complex interface.
    - **Few** or no message sequences are illegal.
  - May have a **class invariant:**
    - Defines all valid combinations of value of attributes

  - **How can we select message combinations that are likely to reveal faults in non-modal behaviour?**

Fault Model

- Many sequence-related bugs are still possible
  - A legal sequence is rejected
  - A legal sequence produces an incorrect value
  - An accessor method has an incorrect side effect that alters or corrupts object state
  - A legitimate modifier message is rejected
  - An illegal modifier message is accepted, resulting a corrupt state
  - An incorrect computation causes the class invariant to be violated

  - Example: a *DateTime* object might incorrectly throw an exception when the *setHour* message is sent twice in succession to the same value

Strategy: Test model

- Test model: **Class Invariant**
- Nonmodal bugs found with this test pattern:

Test	Behavior Tested	Pass	No Pass
Define-operation	Define operation accepts valid input	*On* point is accepted (assuming On satisfies invariant)	*On* point is rejected
Define-exception	Define operation rejects invalid input	*Off* point is rejected (assuming Off invalidates invariant)	*Off* point is accepted
Define-exception-corruption	Define exception changes state	No change in state after an exception	State is corrupted after an exception
Use-exception	Use operation does not throw an exception	Operation returns normally	An exception is thrown
Use-correct-return	Use operation returns same value as input to the define operation	Use and define values are the same	Use and define values are not the same
Use-corruption	Use operation does not corrupt the state of OUT	OUT’s state unchanged after a use operation	OUT’s state is changed after a use operation

Strategy: Test procedure

1. Find class invariant
2. Develop set of test cases using *Invariant Boundaries*
3. Select a message sequence strategy
   - E.g., define-use or suspect
4. Set the OUT to a test case from the domain matrix
   1. Use a modifier/define method
5. Send all accessor messages and verify that the returned values are consistent with the defining value
6. Repeat steps 3 and 4 until all cases of the domain matrix have been exercised

Strategy: Test Model - 3

- Sequences may be selected in several ways:
  - *Define-use sequence*: Consists of a *definition* method followed by all the *use* methods
  - *Random sequence*: The sequence of *use* and *modifier* calls is assigned randomly
  - *Suspect sequence*: If a sequence is suspect for any reason, try it.
    - We may suspect that *DataTime* operations involving a leap year of Feb 29 may fail
- Best: mix **define-use** with **random**
- Do not consider define-define sequences
  - **Why?**

Example: DateTime

```
class DateTime extends Object {
  DateTime(int sec, int min, int hour, int day,
               int month, int year, int zone);
  void setSec(int sec);
  void setMin(int min);
  void setHour(int hour);
  void setDay(int day);
  void setMonth(int month);
  void setYear(int year);
  void setZone(int zone);
  int getSec();
  int getMin();
  int getHour();
  int getDay();
  int getMonth();
  int getYear();
  int getZone();
  boolean equal(DateTime dt);
  DateTime add(DateTime dt);
  DateTime sub(DateTime dt);
}
```

Variable domains for DateTime

Variable	**Type**	**Minimum**	**Maximum**
sec	integer	0	59
min	integer	0	59
hour	integer	0	23
day	integer	1	31
month	integer	1	12
Year	integer	1900	32767
zone	integer	1	24

State invariant: 0 <= sec <= 59 && 0 <= min <= 59 && 0 <= hour <= 23 && 1 <= day<= 31 && …

Example – On and Off points

On and Off points for the DateTime invariant		
Boundary Condition	***On* point**	***Off* point**
sec,min >= 0	0	-1
sec,min <= 59	59	60
hour >= 0	0	-1
hour <= 23	23	24
day >= 1	1	0
day <= 31	31	32
month >= 1	1	0
month <= 31	31	32
year >= 1900	1900	1899
year <= 32767	32767	32768
zone >= 1	1	0
zone <= 24	24	25

Example 3: Matrix Domain

Boundary			Input Test Values													
variable	Condition	Type	1	2	3	4	5	6	7	8	9	10	11	12	…	24
sec	>= 0	On	0													
		Off		-1												
	<= 59	On			59											
		Off				60										
	Typical	In					24	48	52	6	24	32	8	19		55
min	>= 0	On					0									
		Off						-1								
	<= 59	On							59							
		Off								60						
	Typical	In	47	23	14	13					22	40	55	4		7
Hour	>= 0	On									0					
		Off										-1				
	<= 23	On											23			
		Off												24		
	Typical	In	16	16	6	10	18	3	15	9						18
....																
Expected Result			✓	✖	✓	✖	✓	✖	✓	✖	✓	✖	✓	✖		✖

Should also **have expected** output in domain matrix

Basic test strategy

1. Pick up a test vector
2. Set the OUT to the test vector value with a modifier or a constructor
3. Verify that modifier has produced a correct state/behaviour
   - If test vector is *off* point (closed boundary)/*on* point (open boundary)
     - Check exception thrown and state of OUT same
   - If *on* point (closed boundary)/*off* point (open boundary)
     - Check OUT is placed in the expected state
4. Try each accessor method to see that it reports the state correctly without a buggy side effect
   - Apply a different use method sequence for each test case
5. Can repeat 2-4 with all possible modifiers or just pick one

Selection of a modifier method

- Consider test vector 1
  - (sec=0, min = 47, hour = 16, …)
- This test value can be exercised with several *modifier* methods:
  1. dt = new DateTime(0, 47, 16, …);
  2. dt = new DateTime(4, 47, 16, …);
     dt.setSec(0);
  3. dt = new DateTime(3, 48, 17, …);
     dt2 = new DateTime(3, 1, 1, …);
     dt3 = dt.sub(dt2);

```
class DateTime extends Object {
  DateTime(int sec, int min, int hour,
    int dayMonth, int year, int zone);
  void setSec(int sec);
  void setMin(int min);
  void setHour(int hour);
  void setDay(int day);
  void setMonth(int month);
  void setYear(int year);
  void setZone(int zone);
  int getSec();
  int getMin();
  int getHour();
  int getDay();
  int getMonth();
  int getYear();
  int getZone();
  boolean equal(DateTime dt);
  DateTime add(DateTime dt);
  DateTime sub(DateTime dt);
}
```

Example 4

```
public void testTime1() {
   Time t = new Time(0, 47, 16, 104, 1864, 21);
   assertTrue(0, t.getSec());
   assertTrue(47, t.getMin());
   assertTrue(16, t.getHour());
   assertTrue(104, t.getDay());
    ….
}
```

```
public void testTime2A() {
 try {
    Time t = new Time(-1,23,16,70,21759,11);
     fail();
  } catch (SomeException se) { }
}
```

Or

```
public void testTime2B() {
  Time t = new Time(1,23,16,70,21759,11);
   try {
     t.setSec(-1);
     fail();
   } catch (SomeException se) {
      assertTrue(23, t.getMin());
      assertTrue(16, t.getHour());
      assertTrue(1, t.getSec());
      assertTrue(70, t.getDay());
      ….
   }
}
```

```
public void testTime3() {
  Time t = new Time(1,14,6,70,1769,9);
  Time t2 = new Time(58,0,0,1,1769,9);
   t = t.add(t2);
   t.setSec(59);
   assertTrue(59, t.getSec());
   assertTrue(14, t.getMin());
   assertTrue(6, t.getHour());
   assertTrue(70, t.getDay());
  assertTrue(1769, t.getYear());
      ….
}
```

Entry and Exit Criteria

- Entry Criteria
  - Alpha – omega cycle on the CUT
- Exit Criteria
  - All define-use method pairs have been exercised, and an object of CUT has taken on the values in each test case at least once
  - Achieve at least branch coverage on each method in the CUT

FSM based testing

- Behavior of SUT modeled by a finite state machine
- Example: Selling Machine
  - Insert two coins
  - Select coffee button for coffee or tea button for tea

Context - State machine

- A state machine is a system whose output is determined by both current and past input
- The effect of previous inputs is represented by a state
- A state machine can model the behavior of classes that are sensitive to the sequence of past messages
- A system has a state-based behavior when identical inputs are not always accepted and, when accepted, may produce different outputs

State machine

- State – an abstraction of past inputs
  - The **initial state** is the state in which the first event is accepted
  - A machine may be in only one state at a time
  - The current state refers to the active state
  - The **final state** is one in which the machine stops accepting events
- Transition – an allowable two-state sequence, an acceptance state and a resultant state, that is caused by an event and may result in an action
- Event
  - Can be the invocation of a method or a time interval
  - Can have a pre or post condition
  - If the event is not accepted in the current state, it is ignored
- Action – the result or output that follows an event

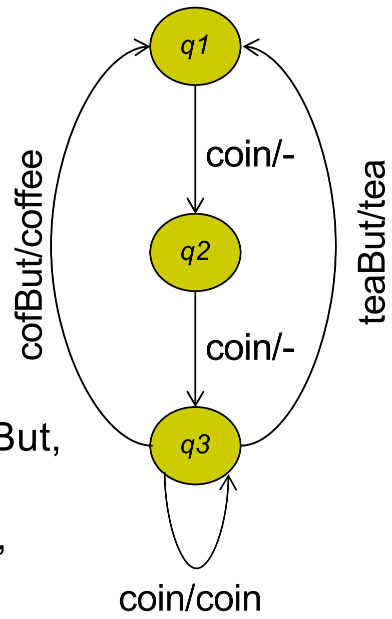
State machine: FSM model

- FSM - Finite State Machine - is 5-tuple
  - $M = (S, I, O, \delta, \lambda)$
- where
  - S is a finite set of states
  - I is a finite set of inputs
  - O is a finite set of outputs
  - $\delta : S \times I \rightarrow S$ (transfer function)
  - $\lambda : S \times I \rightarrow O$ (output function)

FSM example

Current state	Input	Output	Next state
q1	coin	-	q2
q2	coin	-	q3
q3	cofBut	coffee	q1
q3	teaBut	tea	q1
q3	coin	coin	q3

```
public class VendingMachine {
    //…
    public void cofBut() { ...}
    public void teaBut() {...}
    public void coin() { ...}
}
```



Described by:

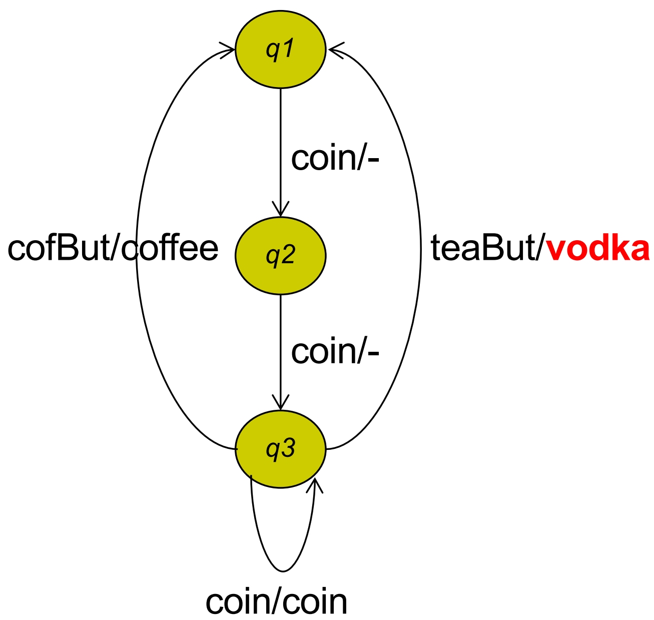
- Inputs: {coin, cofBut, teaBut}
- Outputs: {coffe, tea, coin}
- States: {q1, q2, q3} and Initial State: q1
- δ : {(q1, coin, q2), (q2, coin, q3), (q3, teBut, q1), (q3, cofBut, q1), (q3, coin, q3)}
- λ : {(q1, coin, -), (q2, coin, -), (q3, teBut, tea), (q3, cofBut, coffee), (q3, coin, coin)}

Conformance testing of an FSM

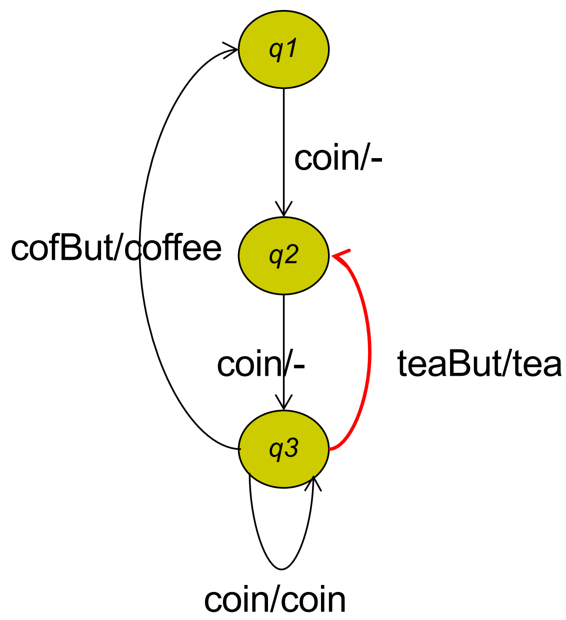
- Given
  - Specification FSM A
  - Implementation FSM B
- Conformance Testing Goal
  - Check that B conforms to A:
    - i.e., B behaves in accordance with A
    - i.e., outputs of B are the same of A

Fault model - 1

- Output fault
  - Wrong or missing

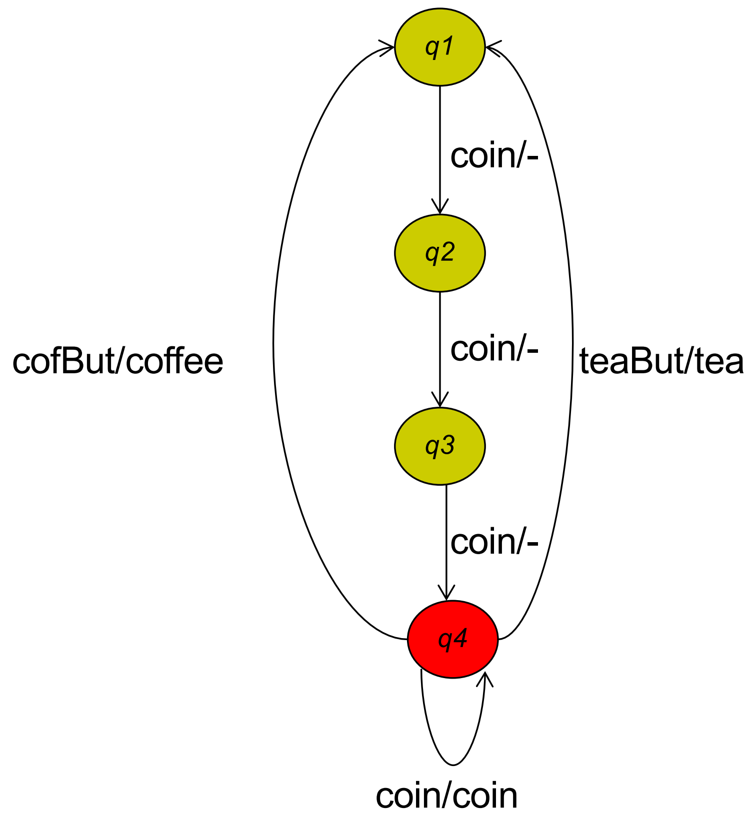


- Transition fault
  - Wrong or missing



Fault model - 2

- Extra or missing state

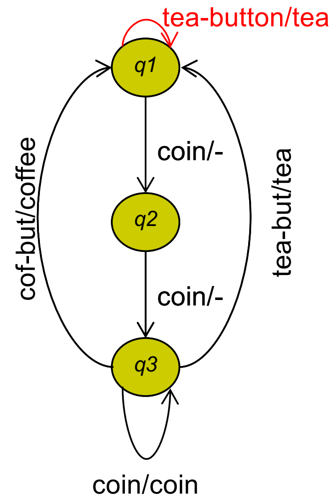


Conformance testing

- Checking these faults only shows that implementation agrees with the **explicit** behavior of the FSM
- Implicit behavior:
  - Transitions not declared are correctly handled
    - Do not affect state
- We also need to exercise excluded transitions
  - Represent an invalid sequence
  - Designated as sneak path

Fault model - 3

- Sneak path
  - Allows a message to be accepted when it should have been rejected
    - This results into an illegal transition



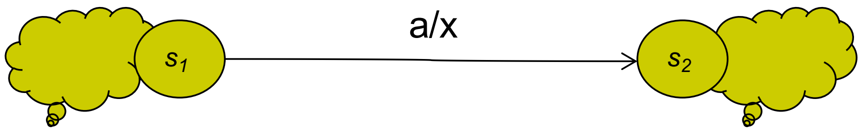
- Transitions may have a condition
  - Called guard
  - Also must consider sneak path for this case

Desired properties

- Make testing easier but not always available
  - **Status query**
    - Tester can query current state of SUT **without** changing state
  - Reset
    - Reliably bring SUT to initial state
  - Set state
    - Reliably bring SUT to any state

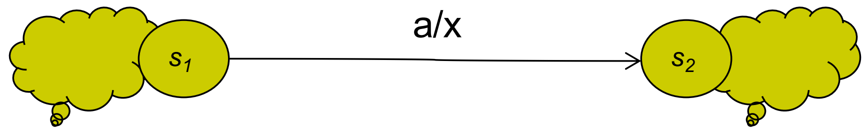
Conformance testing implementation

- One test case per transition in FSM specification
- Test case transition
  - Check that
    - When in state s_1
    - Submit input a
    - Then output is x and
    - Resulting state is s_2
- At most |S| * | I | + 1 test cases




```
@Test
public void checkTransition() {
  // Arrange
  sut.setState(s1);
  // Act
  x = sut.a();
  // Assert
  assertEquals(expectedOutput, x);
  assertEquals(sut.getCurrentState(), s2);
}
```

Transition testing – 1



- Task: *Set OUT in state* s_1
- How?
- Two approaches:

1. Use *set-state* property if available
2. No *set-state* property
   - Use reset property if available
   - Go from *initial state* to s_1
     - Always possible since FSM is deterministic
     - May have several paths
       - Choose one.

Transition testing – 2

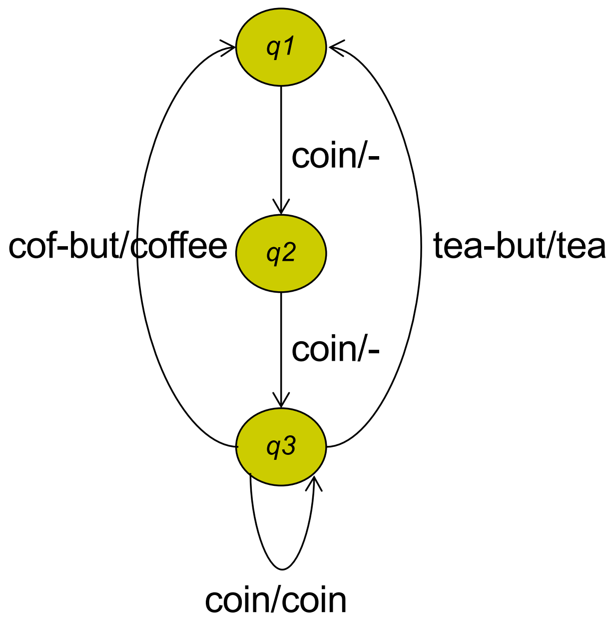
- No status message
- **How can a test case do state verification: Am I in state s?**
- Solution:
  - Apply sequence of inputs in the current state of the FSM such that from the outputs we can verify that we were in a particular start state

Transition testing - 3

- Different kinds of sequences

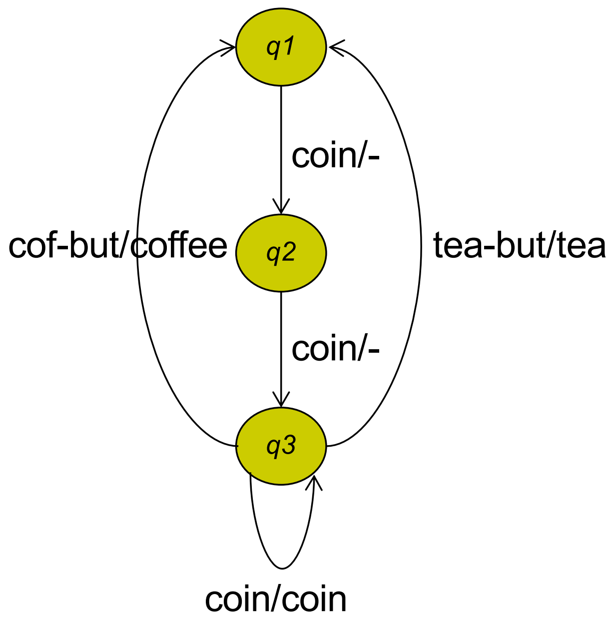
1. Distinguishing sequence (DS)
   - sequence x that produces different output for every state : $\forall$pairs (t,s) with $t \neq s$: $\lambda(s,x) \neq \lambda(t,x)$
   - a distinguishing sequence may not exist

- DS Sequence for *vending machine*
  - coin ; coin
  - Output state q1: - ; -
  - Output state q2: - ; coin
  - Output state q3: coin ; coin



Transition testing - 4

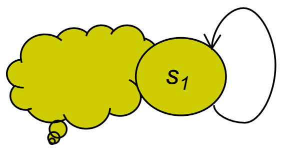
2. UIO (Unique Input Output) sequences
   - sequence x_s that distinguishes state s from all other states : $\forall$pairs (t, s) with $t \neq s$: $\lambda(s,x_s) \neq \lambda(t,x_s)$
   - each state has its own UIO sequence
   - UIO sequences may not exist



- UIO Sequences for *vending machine*
- State q1: coin/- ; coin/-
- State q2: coin/- ; tea-but/tea
- State q3: cof-but/coffee

Non-conformance test cases

- Check implicit behavior
- One test case per invalid transition in FSM specification
- Test case transition
  - Check that
    - When in state s_1
    - Submit ***invalid*** input a
    - Then output not changed
    - Resulting state is s_1




```
@Test
public void checkTransition() {
  // Arrange
  sut.setState(s1);
  // Act
  assertThrows(InvalidTransitionException.class,
                        () -> { sut.a();} );
  // Assert

  assertEquals(sut.getCurrentState(), s1);
}
```

Modal Class Test

- Intent
  - Develop a class scope test suite for a class that has fixed constraints on message sequence
- Context
  - A modal class has both message and domain constraints on the acceptable sequence of messages
    - An *Account* object will not accept a *debit* message to withdraw funds if balance <= 0
    - A *freeze* message is accepted if the account is not *closed* or *frozen*
  - Interactions between message sequences and state are often subtle and complex, therefore error prone

Fault model

- Missing transition
  - A message is rejected in a valid state
- Incorrect action
  - The wrong response is selected for accepting state and method
- Invalid resultant state
  - The method produces a wrong state for this transition
- A corrupt state is produced
  - i.e. violation of class invariant
- Sneak path
  - Allows a message to be accepted when it should have been rejected

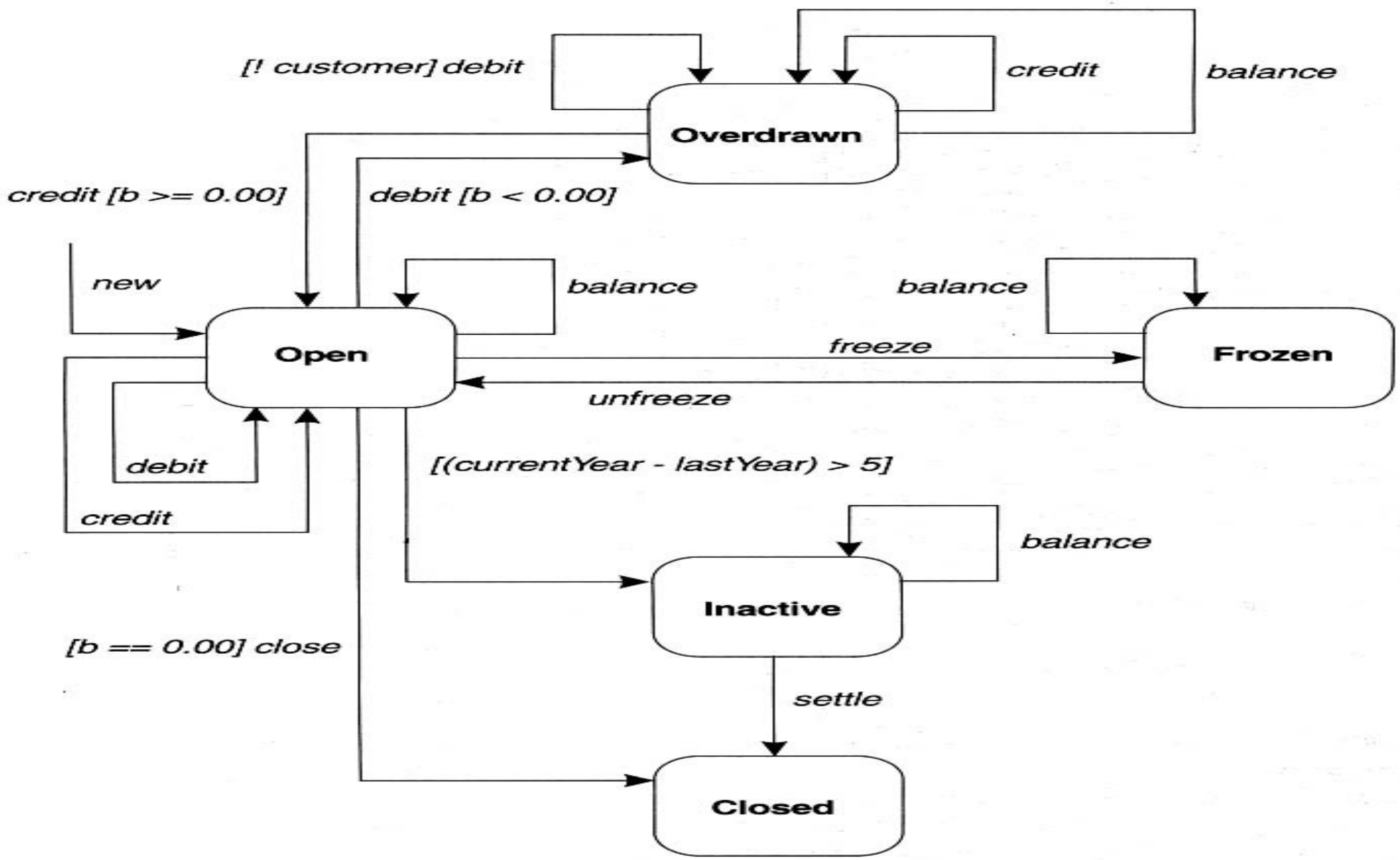
Strategy: Test model

1. Develop state model for CUT.
2. Elaborate the state model with a full expansion of conditional transition variants
3. Generate transition tree
4. Tabulate events and actions along each path to form message sequences
5. Develop test data for each path using **Invariant Boundaries** pattern for events, messages and actions
6. Execute conformance test suite until all tests pass
7. Develop a sneak path test suite. Add all forbidden transitions in all states and define the expected exception
8. Execute sneak path test suite until all tests pass

Example – *Account* class

- With the following operations:
  - open
  - balance
  - credit
  - debit
  - freeze
  - unfreeze
  - settle
  - close
- With the following states:
  - Open
  - Overdrawn
  - Closed
  - Frozen
  - Inactive

Step 1 - Generating the state model for CUT



Step 2 - Full expansion of conditional transition variants

- Some transitions are conditional
  - Conformance testing ensures we fire the transitions when the conditions are met
- What if the condition is not true?
- We must:
  - Develop a truth table for each conditional transition
  - Add additional transition for each truth table entry not covered in conformance tests
- *Guard* may be a pre-condition or a post-condition

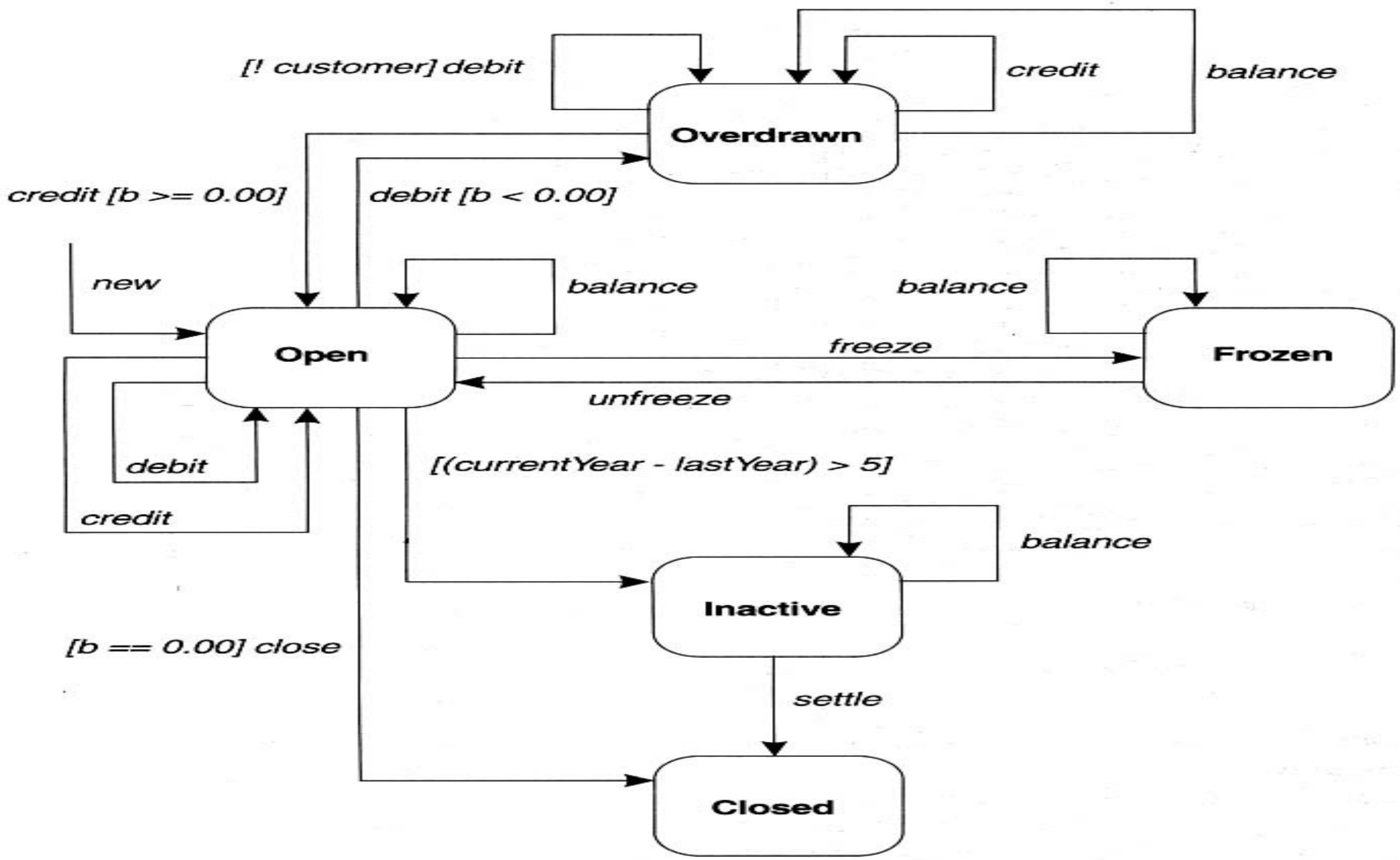
Step 2 - Full expansion of conditional transition variants

- Conditional transition variants

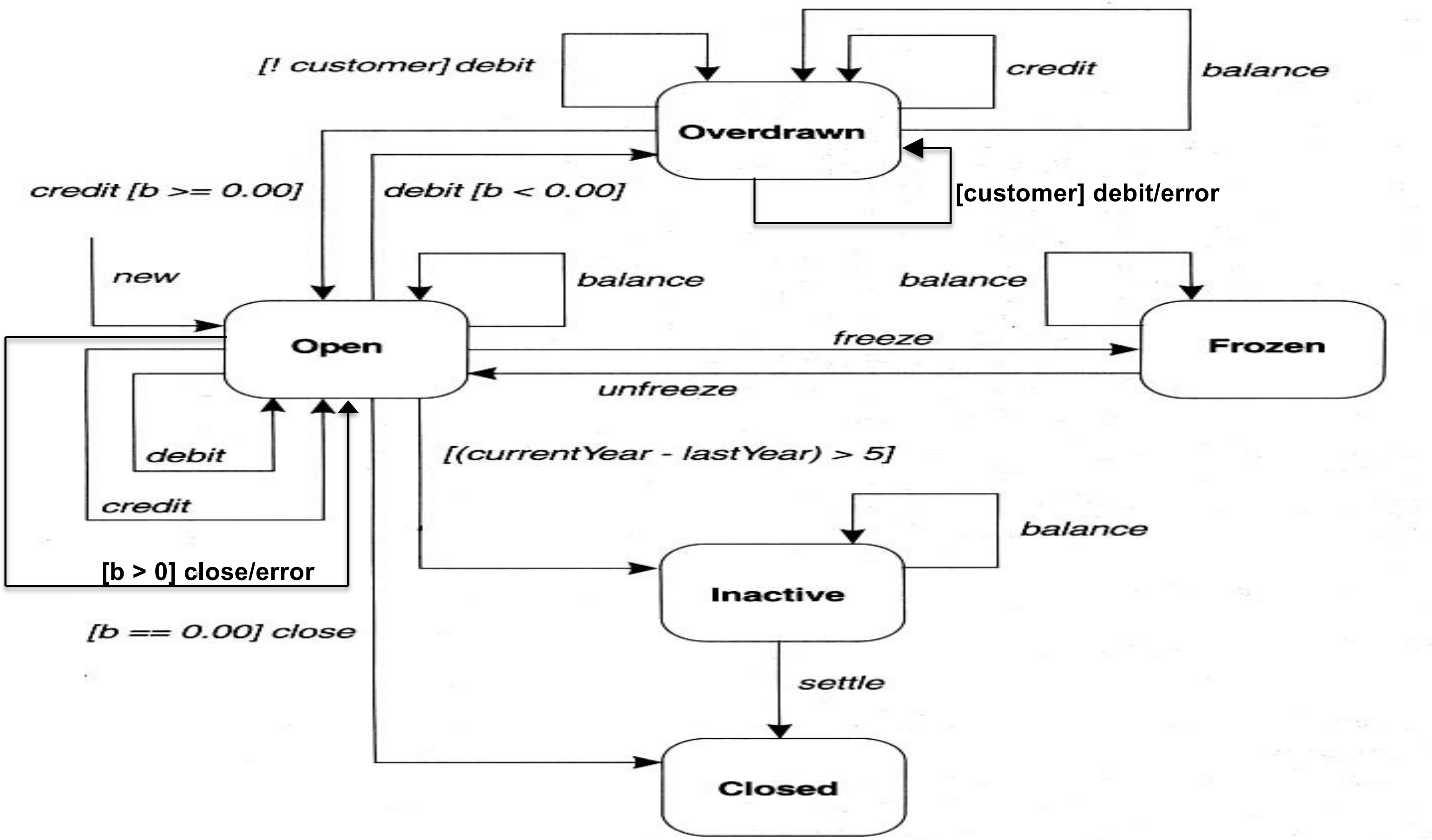
State	Message	Condition	Next State
Overdrawn	credit	Post: balance < 0.00	Overdrawn
Overdrawn	credit ✓	Post: balance >= 0.00	Open
Open	debit	Post: balance < 0.00	Overdrawn
Open	debit ✓	Post: balance >= 0.00	Open
Open	- ✓	Post: currentYear - lastYear > 5	Inactive
Open	close ✗	Pre: balance == 0.00	Closed
Overdrawn	debit ✗	Pre: !customer	Overdrawn

- *close()* in *Open* generates an additional transition when *balance > 0*
- *debit()* in Overdrawn generates an additional transition when *customer*
- **Update** state diagram with the additional transitions

Step 2b – Update the state model for CUT



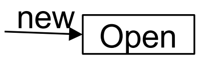
Step 2b – Update the state model for CUT



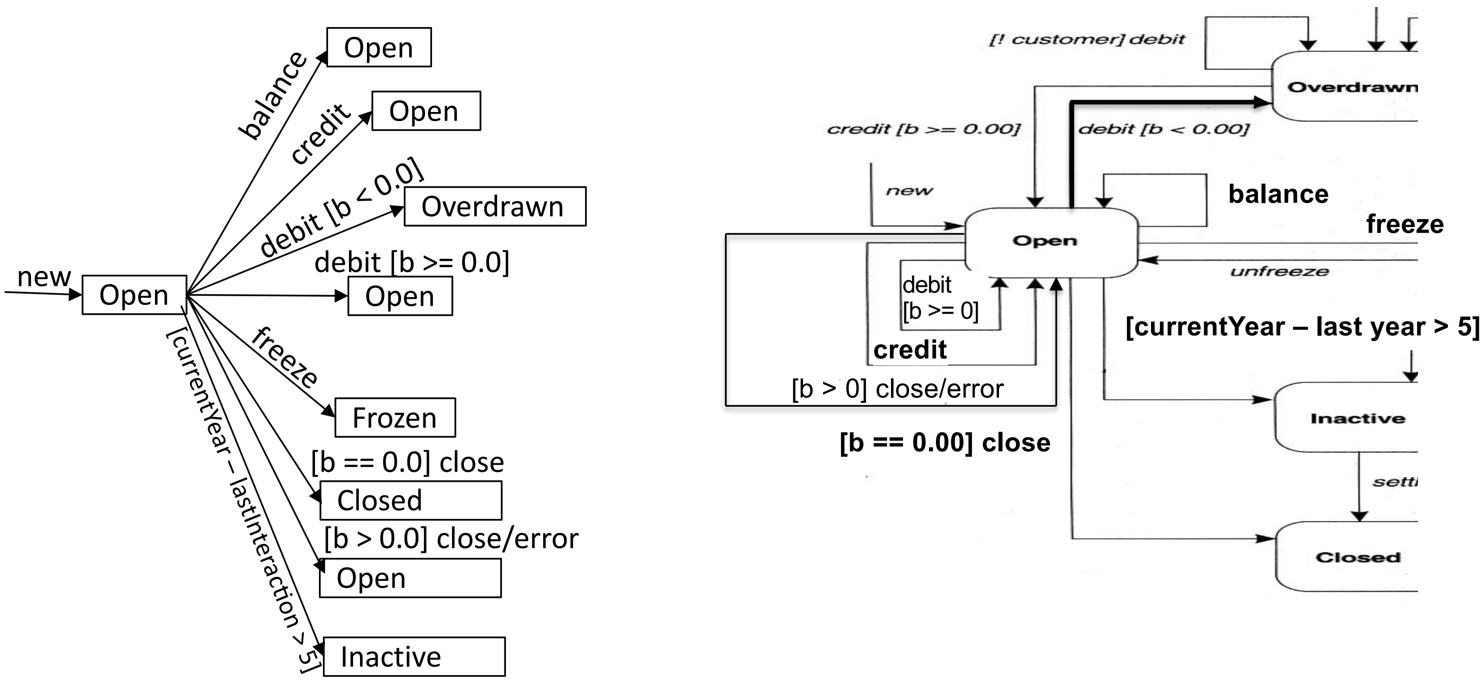
Step 3 - Generate transition tree

1. The initial state is the root node of the tree.
   - Use the alpha state if multiple constructors produce behaviorally different initial states
2. For each non-terminal leaf node in the tree:
   - Draw a new edge and node for each outbound transition
   - The new edge and node represent an event/resultant state reached by an outbound transition
3. For each edge and node drawn in step 2:
   - Copy the corresponding transition event/action to the new edge
   - If the state this node represents is already represented by another node (anywhere in the diagram) or is a final state, this node is terminal – no more transitions are drawn from this node.
4. Repeat steps 2 and 3 until all leaf nodes are terminal.

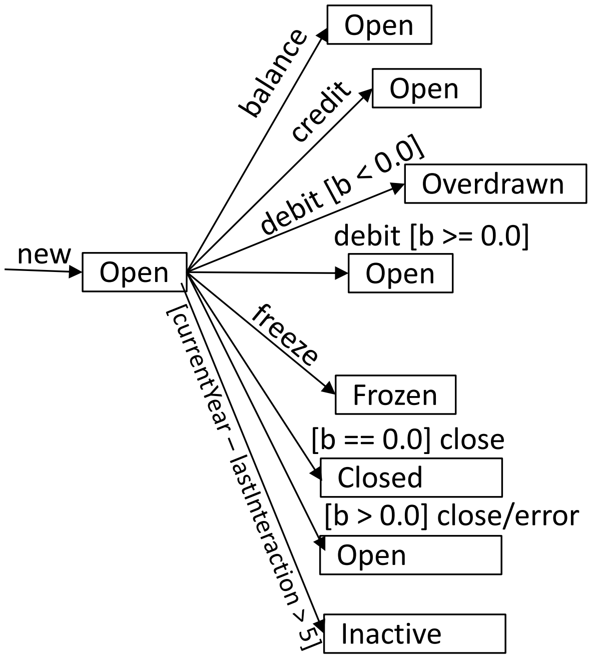
Step 3 - Generating the transition tree: Initial transition



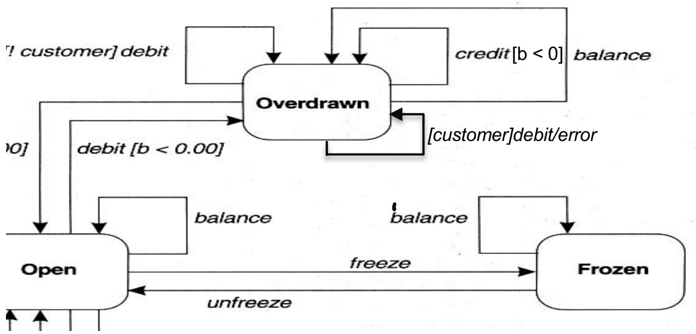
Step 3 - Generating the transition tree: Expand Open state



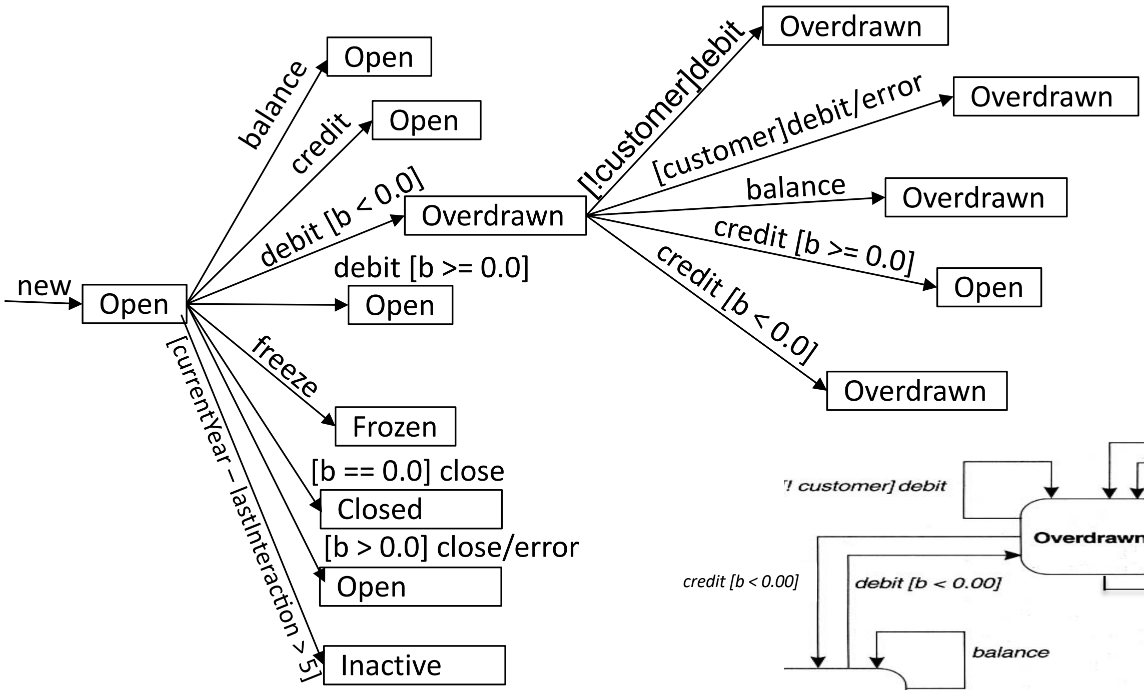
Generating the transition tree: Expand Overdrawn state

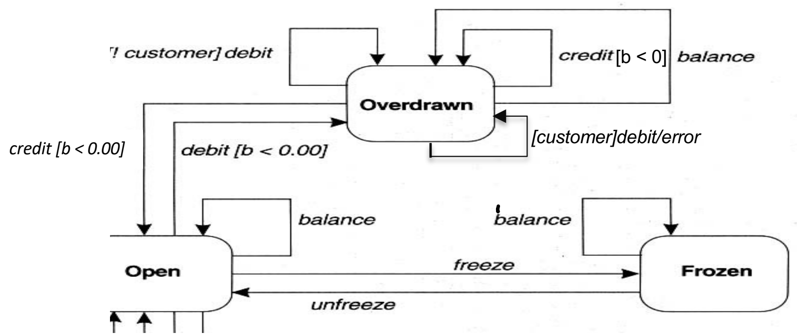


- Consider the Overdrawn state in the state diagram
- Represent all transitions in the transition tree

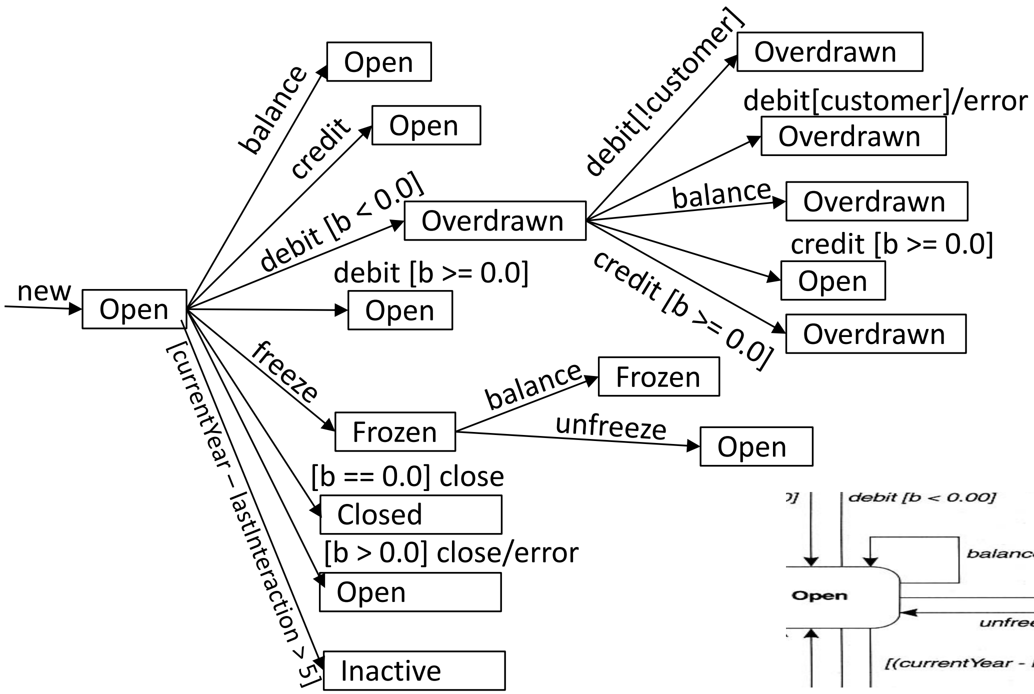


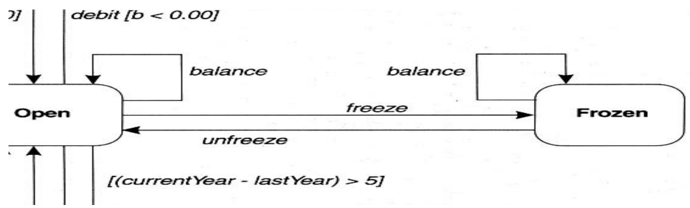
Generating the transition tree: Expand Overdrawn state



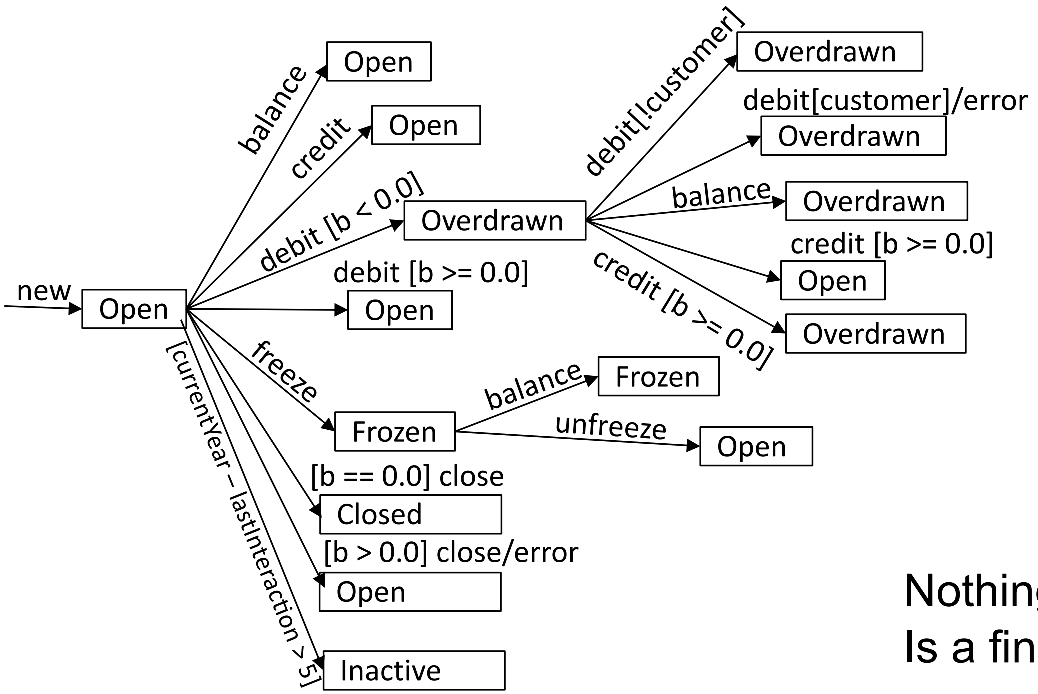


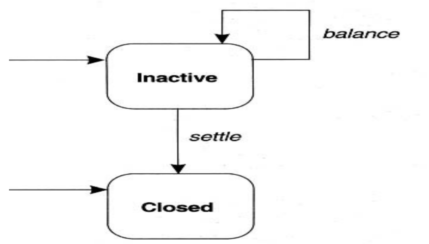
Step 3 - Generating the transition tree : Expand Frozen state





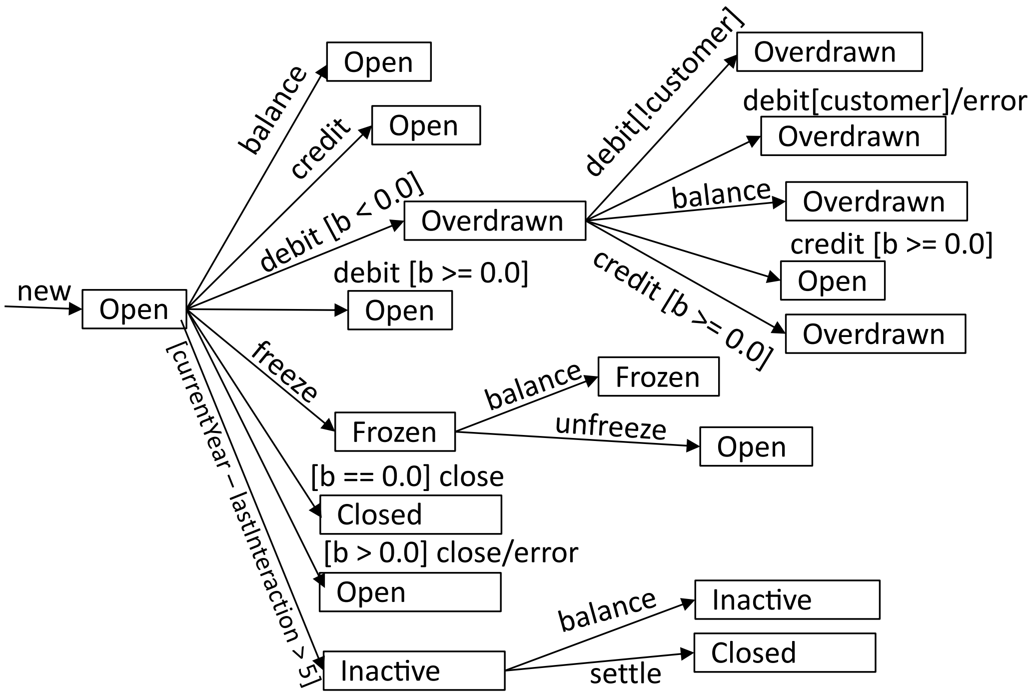
Step 3 - Generating the transition tree : Expand Closed state

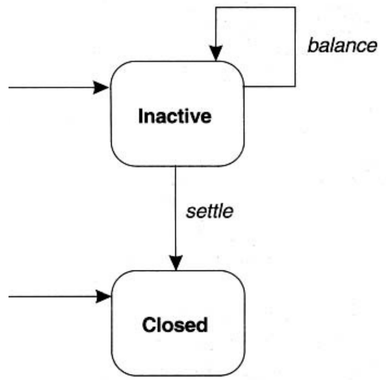




Nothing to expand in this case!
Is a final state.

Step 3 - Generating the transition tree – Expand Inactive state





- All states expanded
- Final version of transition tree

Step 4 - Generate Conformance Test Suite

Run	Test Run/Event Path			Expected Terminal State
	Level 1	Level 2	Level 3	
1	new			open
2	new	balance		open
3	new	credit		open
4	new	debit [b >= 0.0]		open
5	new	debit [b < 0.0]		overdrawn
6	new	debit [b < 0.0]	[customer]debit/error	overdrawn
7	new	debit [b < 0.0]	[!customer]/debit	overdrawn
8	new	debit [b < 0.0]	balance	overdrawn
9	new	debit [b < 0.0]	credit[b > 0.0]	overdrawn
10	new	debit [b < 0.0]	credit[b >= 0.0]	open
11	new	freeze		frozen

Step 4 - Generate Conformance Test Suite

Run	Test Run/Event Path			Expected Terminal State
	Level 1	Level 2	Level 3	
12	New	freeze	balance	frozen
13	New	freeze	Unfreeze	open
14	New	[cY –lY >5]		inactive
15	New	[cY –lY >5]	balance	Inactive
16	New	[cY –lY >5]	settle	closed
17	New	[b!=0.0]close/error		open
18	New	[b==0.0]close		closed

Step 5 - Develop test data for each path using Invariant Boundaries

debit in *Open*		
condition	On Point	Off Point
[b >= 0]	0 ✔	-0,01
[b < 0]	0	-0.01 ✔

close in *Open*		
condition	On Point	Off Point
b > 0	0	0,01 ✔
b == 0	0 ✔	0,01, -0,01

credit in *Overdrawn*		
condition	On Point	Off Point
[b >= 0]	0 ✔	-0,01
[b < 0]	0	-0.01 ✔

- Do not repeat test cases with same input values
- Add at least one test case for each identified cases
- The value -0.01 for *close()* is impossible

Step 7 – Develop Sneak Path Test Suite

- Build Transition Table

Events	States				
	Open	Overdrawn	Frozen	Inactive	Closed
credit	✓	?	PSP	PSP	PSP
debit	?	?	PSP	PSP	PSP
balance	✓	✓	✓	✓	PSP
freeze	✓	PSP	PSP	PSP	PSP
unfreeze	PSP	PSP	✓	PSP	PSP
settle	PSP	PSP	PSP	✓	PSP
5 years	✓	PSP	PSP	PSP	PSP
close	?	PSP	PSP	PSP	PSP

✓ = Valid Transition; PSP = Possible sneak path; ? = Conditional Transition

- Add PSP to the transition tree

Step 7b – Develop sneak path test suite

- Develop Sneak Path Test Suite
  - One test case per PSP
  - Should check *no change* on object state after PSP

Run	Test Run/Event Path			Expected State	Exception
	Level 1	Level 2	Level 3		
19	New	unfreeze		open	✓
20	New	settle		open	✓
21	New	debit [b < 0.0]	freeze	overdrawn	✓
22	New	debit [b < 0.0]	unfreeze	overdrawn	✓
….					

Entry and Exit Criteria

- Entry Criteria
  - Alpha – omega cycle
- Exit Criteria
  - Achieve branch coverage on each method in the Class Under Test
  - Provide N+ coverage
    - A test for each root-to-leaf path in the expanded transition tree and a full set of sneak path cases

Consequences

- This pattern has the following requirements
  - A testable behavior model is available or can be developed
  - The CUT is state-observable

Quasi-modal Class Test

- Intent
  - Develop a test suite for a class whose constraints on message sequence change with the state of the class.
- Context
  - A quasi-modal class has sequential constraints that reflect the organization of information used by the class.
  - Container and collection classes are often quasi-modal.
    - A Stack object has the same behavior for any content or order of items in the stack, but its behavior differs when the stack is empty, holding some items, or full.
      - A push message can be accepted an arbitrary number of times, but is rejected when a stack is full.
      - No combination of stacked values will result in different behavior unless one of these special cases obtains.
  - Effective testing must distinguish between content that affects behavior and content that does not affect behavior

Fault Model

- Those related with a wrong implementation of the state model of the CUT
- Those related with constraints on pairs of operations
  - Example: An item with the same value cannot be added twice to the a set

Strategy: Test Model

- Model the behavior of the class with a state machine
  - States are defined to represent the sequential constraints
    - Each state is defined with a state invariant
  - Methods are represented as events
  - Outbound messages and returned message values are represented as actions

Strategy: Test Model

1. Develop FREE state model for CUT. Characterize state by constraint parameters (not content). Treat each constraint parameters as state variable
2. Elaborate the transition tree with a full expasion of conditional transition variants
3. Generate transition tree. There is no need to indicate loops
4. Tabulate events and actions along each path to form test cases
5. Develop test data for each path using **Invariant Boundaries** pattern for events, messages and actions
6. Execute conformance test suite until all tests pass
7. Develop a sneak path test suite. Add all forbidden transitions in all states and define the expected exception
8. Execute sneak path test suite until all tests pass
9. Run class specific operation-pairs

Strategy: Test Model - 2

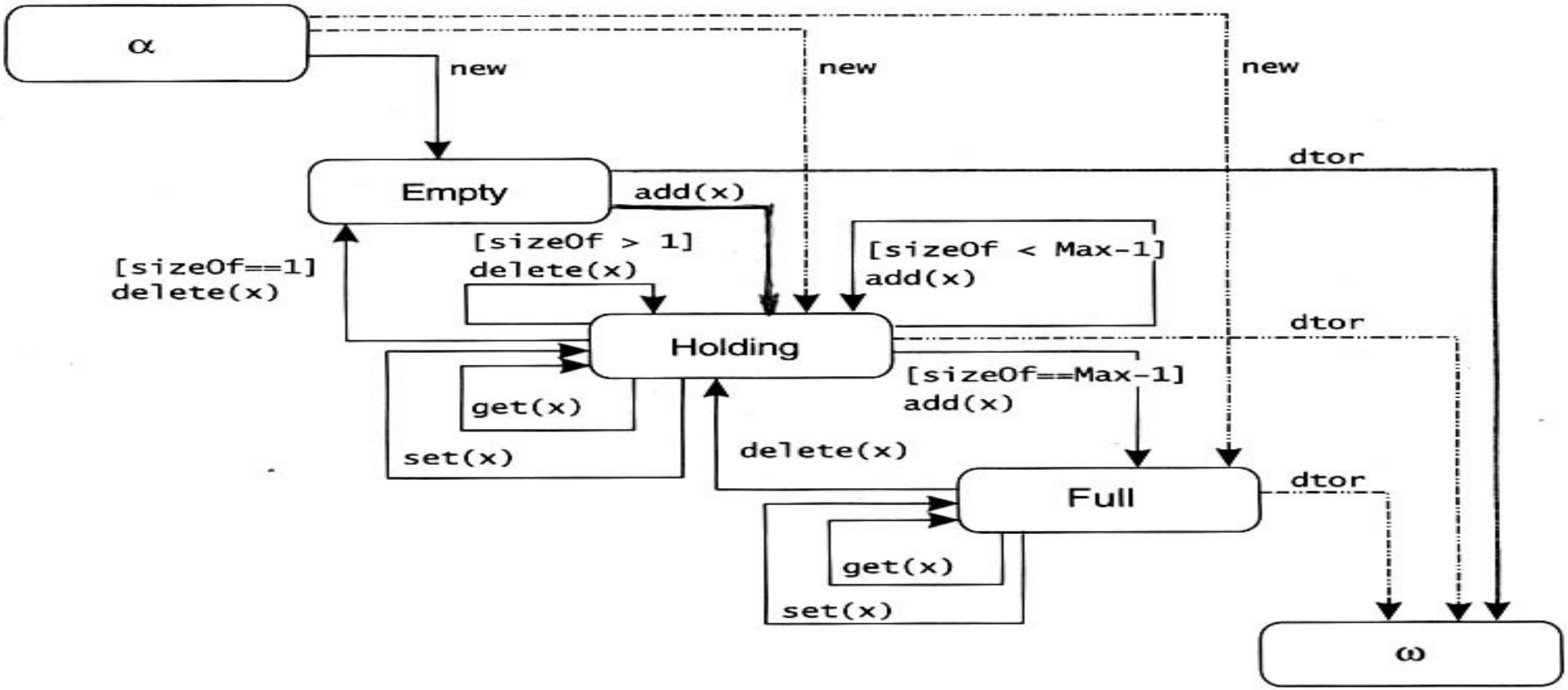
- The Quasi-modal test model has three parts:
  - A state model (states and events) of the generic CUT behavior
  - An *Invariant Boundaries* set for the parameters and parameter relationships that determine behavior
    - With *Stack*, the parameters of interest are the maximum size and the current number of stack elements
    - The parameters can typically be found by examining preconditions, postconditions and class invariants.
  - An operation-pair model of specific CUT behavior
    - Collections typically place additional constraints on pairs of operations.
      - An item with the same value cannot be added twice to a set
      - But can be added an arbitrary number of times to a stack.

Example: Generic collection behavior model

- The generic events:

Event	Definition
new	Create/initialize a new instance (a constructor)
add(x)	Add element x to the collection
set(x)	Change the value of existing element x without removing it from the collection. This operation is typically defined for keyed collections (e.g. dictionary or hash table), but may not be defined for other collections
get(x)	Return a copy of the reference to element x without changing the collection
delete(x)	Remove element x from the collection
dtor	Destroy the collection. This operation may be automatic or programmer-defined

Step 1 – Develop State Model of CUT



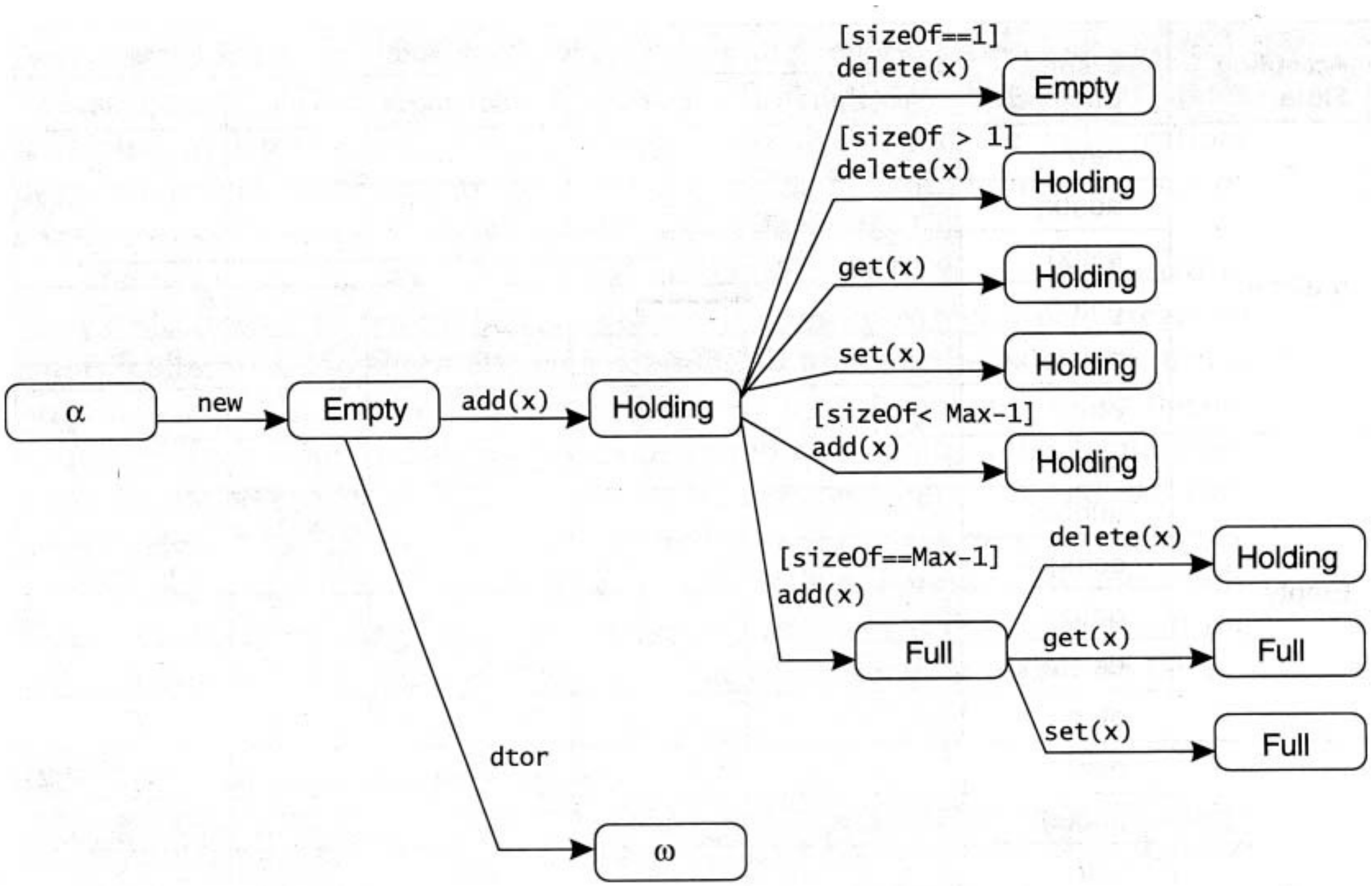
Step 2 - Expasion of conditional transition variants

- Conditional transitions on
  - Holding
    - *[sizeof > 1]remove()* and *[sizeof == 1]remove()*
    - *[sizeof == Max - 1]add()* and *[sizeof < Max - 1]add()*
  - Nothing to do in both cases
    - All possible situations already covered
- Thus, state model remains the same

Step 3a - Generate transition tree

1. The initial state is the root node of the tree.
   - Use the alpha state if multiple constructors produce behaviorally different initial states
2. For each non-terminal leaf node in the tree:
   - Draw a new edge and node for each outbound transition
   - The new edge and node represent an event/resultant state reached by an outbound transition
3. For each edge and node drawn in step 2:
   - Copy the corresponding transition event/action to the new edge
   - If the state this node represents is already represented by another node (anywhere in the diagram) or is a final state, this node is terminal – no more transitions are drawn from this node.
4. Repeat steps 2 and 3 until all leaf nodes are terminal.

Step 3b – Generate transition tree



Step 4 - Generate conformance test suite

Run	Test Run/Event Path				Expected Terminal State
	Level 1	Level 2	Level 3	Level 4	
1	new				empty
2	new	add			holding
3	new	add	[sizeof == 1]remove		empty
4	new	add	[sizeof >= 1]remove		holding
5	new	add	get		holding
6	new	add	set		holding
7	new	add	[sizeof <= Max - 1]add		holding
8	new	add	[sizeof == Max - 1]add		full
9	new	add	[sizeof == Max - 1]add	remove	holding
10	new	add	[sizeof == Max - 1]add	set	full
11	new	add	[sizeof == Max - 1]add	get	full
12	new	dtor			omega

Strategy: Test Procedure 1

- Tabulate events and actions along each path to form test cases
- Develop test data for each path by using *Invariant Boundaries* on message events and actions
- Run the conformance test suite until all tests pass

Step 5 - Develop test data for each path using
Invariant Boundaries

Invariant Boundaries for *add*		
	On Point	Off Point
size < MAX – 1	Max -1*	Max – 2*
size == MAX – 1	Max -1*	Max, Max – 2*

Invariant Boundaries for *remove*		
	On Point	Off Point
sizeof > 1	1*	2*
sizeof == 1	1*	0, 2*

- Create only one test case for this value instead of two
- Add extra test cases to Conformance test suite

Strategy: Test Procedure 2

- Develop a sneak path test suite
    - A sneak path is a bug that allows an illegal message to be accepted, resulting in an illegal transition
  - Add all illegal transitions for all states and define the expected exception
- Run the sneak path test suite until all tests pass.
- If any method scope tests have not yet been implemented, add them to the test suite and rerun until all tests pass.

Step 7 - Generate sneak path test suite

- Identify the possible sneak paths (PSP)
- Develop Sneak Path test suite by implementing one test case for each PSP

Run	Test Run/Event Path				Expected Terminal State	Exception
	Level 1	Level 2	Level 3	Level 4		
13	new	remove			empty	EmptyStack
14	new	get			empty	EmptyStack
15	new	set			empty	EmptyStack
16	new	add	dtor		holding	Invalid*
17	new	add	[sizeof == Max - 1]add	add	full	FullStack
18	new	add	[sizeof == Max - 1]add	dtor	full	Invalid*

Strategy: Test procedure 3

- After validating the behavior required by the generic state model, test class-specific behaviors:
  - ht.put("ex", obj1); ht.put("ex", obj2); is accepted and the value associated to key "ex" is obj2
- The focus of this pattern
- Design the interesting operation sequences

Step 9 - Interesting operation sequences

- Plan single and paired operations to test the class specific behavior
  - Example for a keyed collection (e.g. Dictionary in SmallTalk).
  - There are many more interesting operation sequences (check book).

Operation	Key value of x	Key value of y	y in Collection	Expected Result
add(x), add(y)	Any	Same	No	Second rejected
get(x), get(y)	Any	Same	Yes	Accepted
set(x), set(y)	Any	Same	Yes	Accepted
delete(x), delete(y)	Any	Same	Yes	Second rejected
add(x), add(y)	Any	Different	No	Accepted
get(x), get(y)	Any	Different	No	Rejected
set(x), set(y)	Any	Different	Yes	Accepted
delete(x), delete(y)	Any	Different	Yes	Accepted

Entry and Exit Criterias

- Entry Criteria
  - Alpha – omega cycle
- Exit Criteria
  - Achieve at least branch coverage on each method
  - Provide N+ coverage
    - A test for each root-to-leaf path in the expanded transition tree and a full set of sneak path pairs

Consequences

- Requirements
  - Testable behavior model is available or can be developed.
  - The CUT is state observable
  - A suitable test driver is available
  - Some servers of the CUT may require stubs to provide sufficient control of CUT state
- The test generation procedure is able to reveal:
  - missing transitions
  - missing or incorrect actions, and
  - incorrect or invalid resultant states
- It does not explicitly focus on incorrect output or incorrect values that are within the bounds of valid states