State-Based testing

Adapted from many
variants of Somé & Williams
original presentation of Binder's work

What is Grey Box testing

Grey Box testing is testing done with ***limited*** knowledge of the internal of the system.

Grey Box testers have access to detailed design documents with information beyond requirements documents.

Grey Box tests are generated based on information such as state-based models or architecture diagrams of the target system.

State based testing

The tests are derived from a state model of the system. We can derive the state model in several way, e.g. from

- Expected system behavior
- State part of a UML design or requirements specification.
- Other state diagrams

Most system will, however, have a ***large*** number of states

Potential Faults (1)

- This slide and the next represent the error model for state-based models. [Adapted from Binder, *Testing of OO systems*.]
- Action fault
  - The actions on a transition are incorrect, or missing.
- Guard condition fault
  - The guard condition on a transition is not correct.
- Next state fault
  - Transition is to a legal, but incorrect, state.
- State fault
  - There are extra states, or missing states.

Potential Faults (2)

- Unspecified event / missing transition
  - There is no transition specified for a legal event at a particular state.
- Extra transition (sneak path)
  - A legal event is accepted in a particular state, but it should not be.
- Illegal event failure
  - Unexpected event causes a failure
- Trap door
  - Implementation accepts events that are not intended to be accepted at any time.

Potential Faults (3)

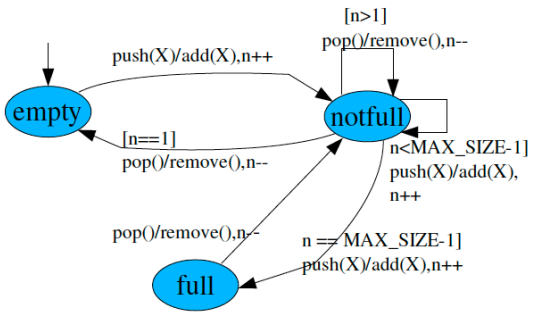
Event	Action	Result State	Description
OK	OK	OK	Normal transition
	wrong	OK	Incorrect action
	undefined	Wrong	Incorrect action, wrong state
		Corrupt	Incorrect action, corrupt state
	Missing	OK	Missing action
		Wrong	Missing action, incorrect state
		Corrupt	Missing action, corrupt state
Reject Legal	DC	Same	Missing transition, no side effect
		Defined	Missing transition, side effect
		Corrupt	Missing transition, corrupt state
Accept Illegal	Defined	Same	Sneak path, no side effect
		Defined	Sneak path, side effect
		Corrupt	Sneak path to corrupt state
	Undefined	Same	Sneak path, no side effect, incorrect output
		Defined	Sneak path, side effect, incorrect output
		Corrupt	Sneak path to corrupt state, incorrect output
Accept Undefined	Defined	Same	Trap door to action
		Defined	Trap door to action, side effect
		Corrupt	Trap door to action, corrupt state
	Undefined	Same	Trap door, incorrect output
		Defined	Trap door, incorrect output, side effect
		Corrupt	Trap door, incorrect output to corrupt state

State test criteria

We can choose one or more of the following test selection criteria:

- All states – testing passes through all states
- All events – testing forces all events to occur at least once
- All actions – testing forces all actions to be produced at least once

State test criteria

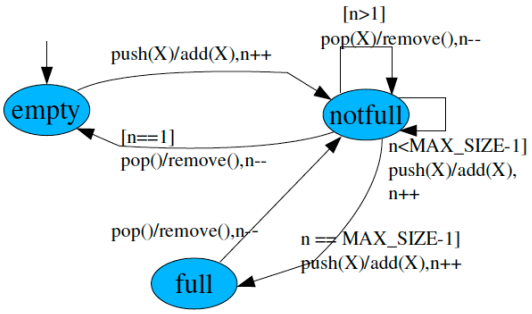


All States – testing passes through all states

- **empty – notfull - full**

ID	Start State	Event	Condition	Reaction	New state
1.1	-	constructor	-	-	empty
1.2	empty	push(X)	-	add(X),n++	notfull
1.3*	notfull	push(X)	n<MAX_SIZE-1	add(X),n++	notfull
1.4	notfull	push(X)	n==MAX_SIZE-1	add(X),n++	full

State test criteria

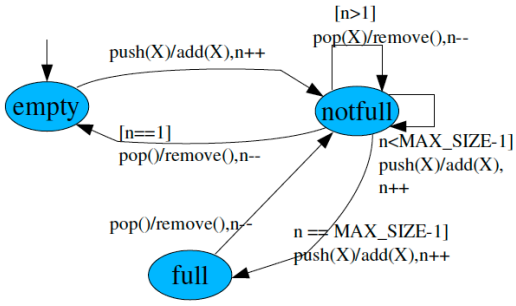


All Events – all events consumed at least once

- **push – pop**

ID	Start State	Event	Condition	Reaction	New state
1.1	-	constructor	-	-	empty
1.2	empty	push(X)	-	add(X),n++	notfull
1.3	notfull	pop()	n==1	remove(),n--	empty

State test criteria

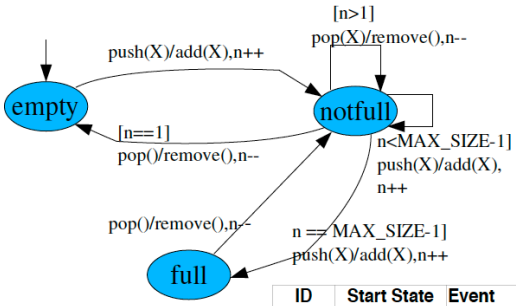


All Actions – all actions produced at least once

- **add, n++ – remove, n--**

ID	Start State	Event	Condition	Reaction	New state
1.1	-	constructor	-	-	empty
1.2	empty	push(X)	-	add(X),n++	notfull
1.3	notfull	pop()	n==1	remove(),n--	empty

State test criteria



All Transitions – all transition taken at least once

Includes:
- All-States
- All-Events
- All-Actions

ID	Start State	Event	Condition	Reaction	New state
1.1	-	constructor	-	-	empty
1.2	empty	push(X)	-	add(X),n++	notfull
1.3*	notfull	push(X)	n<MAX_SIZE-1	add(X),n++	notfull
1.4	notfull	push(X)	n==MAX_SIZE-1	add(X),n++	full
1.5	full	pop()	-	remove(), n--	notfull
1.6*	notfull	pop()	n>1	remove(), n--	notfull
1.7	notfull	pop()	n==1	remove(), n--	empty

All n-Transitions sequences: sequences of transition of length *n*

Testing strategies

- Coverage criteria: during the execution of the test suite ...
  - All transitions taken
    - Transition coverage is generally considered as the minimum acceptable coverage of an FSM-based model.
  - All state-event combinations are covered (next slide).
    - This can detect problems when an FSM is not completely specified, or there are extra transitions.
    - Includes transition coverage
  - All single transition paths
    - All paths starting from any state, where no transition is covered twice.
    - Can find "history errors", where a path passes through
  - Loop coverage

12

State – event table

Event / State	`push`	`pop`
Empty	`add, size++`	`error`
Partly full	[size < max – 1] `add, size++`	[size > 1] `remove, size--`
	[size = max – 1] `add, size++`	[size = 1] `remove, size--`
Full	`error`	`remove, size--`

13

State-event coverage

- Assumption : `max = 3`.
- Sequence of events:

[initial state]	`-- / size = 0`
[empty]	`push / add, size++`
[partly full] [size < max – 1]	`push / add, size++`
[partly full] [size = max – 1]	`push / add, size++`
[full]	`push / error`
[full]	`pop / remove, size–`
[partly full] [size > 1]	`pop / remove, size–`
[partly full] [size = 1]	`pop / remove, size–`
[empty]	`pop / error`

- The result is the same as for a transition tour (next slide) because the FSM was completely specified relative to events `push` and `pop`.

14

Transition tour

- A transition tour is typically a single path or a small set of paths through the FSM during which all transitions are taken at some point.
- What can be detected: incorrect responses on a transition.
- What may not be detected: transitions that have a correct response, but go to an incorrect state.

15

A transition tour of the stack

- Assumption for the tour: **`max = 3`**.
- The following is a sequence of events and expected responses that comprise a transition tour:

```
-- / size = 0
push / add, size++
push / add, size++
push / add, size++
push / error
pop / remove, size--
pop / remove, size--
pop / remove, size--
pop / error
```

- Each of the transitions has been covered. If we can execute the entire sequence, and the expected responses are observed, the transition tour passes.

16

Round-trip path tree – 1

A round-trip path tree

- Is built form a state transition diagram
- Includes all round-trip paths
  - Transition sequences beginning and ending in the same state
  - Simple paths for initial to final state. If a loop is present, we use only one iteration
- Is used to
  - Check conformance to explicit behavioral models
  - Find sneak paths

Round-trip path tree – 2

A test strategy based on round-trip path trees (such as Binder's N+) will reveal:

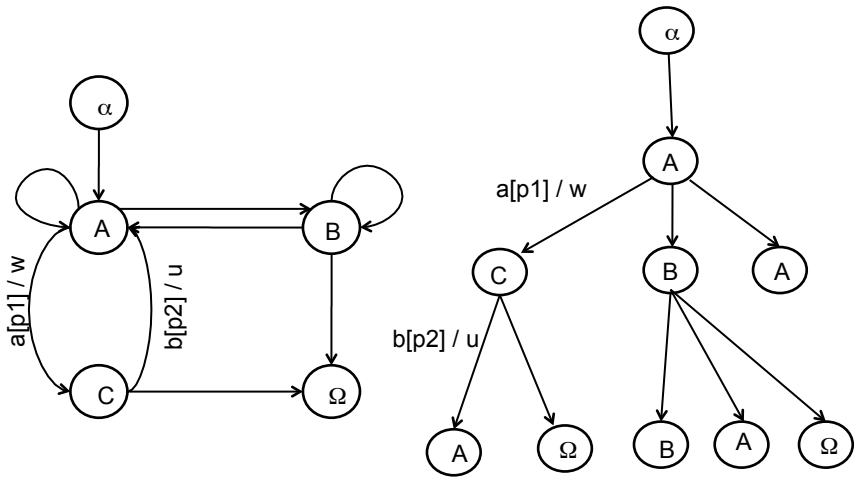
- All state control faults
- All sneak paths – messages are accepted when they should not
- Many corrupt states - unpredictable behavior

Challenge for round-trip path testing

In order to test a system based on state transitions via triggers, predicates (guards) and activities, we need to be able to observe and log these entities.

Thus, we may need to include "points of observations" in the code that gives us access to the necessary information.

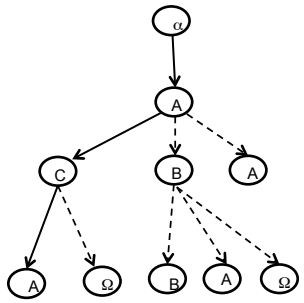
Round-trip tree – small example



Test description – 1

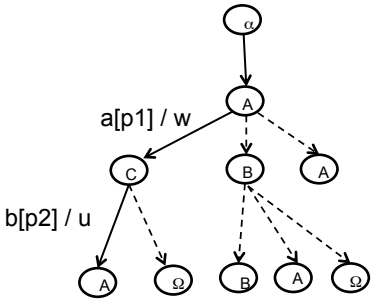
Each test completes one branch of the round-trip tree. The necessary transitions describes the test case.

The table on the next slide shows the test case for
α -> A -> C -> A



Test description – 2

ID	Start state	Event	Condition	Reaction	New state
1	α	constructor	-	-	A
2	A	a	p1	w	C
3	C	b	p2	u	A



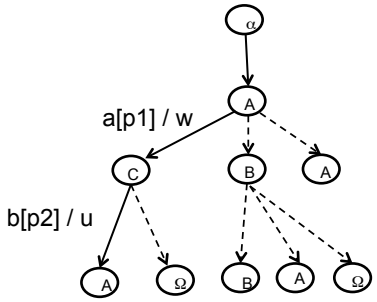
Sneak path test cases

A sneak path – message accepted when it should not be accepted – can occur if

- There is an unspecified transition
- The transition occur even if the guard predicate is false

Sneak path test description

ID	Start state	Event	Condition	Reaction	New state
1	α	constructor	-	-	A
2	A	c	p1	Error message	A
3	A	a	p1 - false	Error message	A



State Verification

- How can we check that a transition ends in a specific state?
- If we have an implementation that has a `queryState()` method, we can call this after a transition has been executed, and see if we are in the correct state.
  - Requires an implementation that has been designed with testing in mind.
- Otherwise, we have to use indirect evidence
  - (see appendix to see how complex this is!)

25

Bottom Line (1)

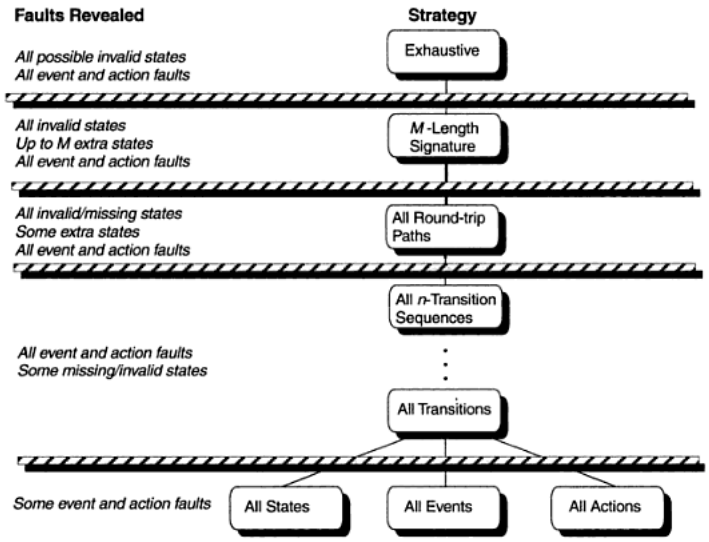


FIGURE 7.39 State-based test power hierarchy.

Bottom Line (2)

TABLE 7.12 Size of State-based Test Suite by Strategy and Size of IUT

Representative IUT				Test Strategy					
				All Transitions		N+		W-Method	
	n	k		(1)	(2)	(1)	(2)	(3)	(4)
3Player	7	9	Test Cases	21	32	63	63	63	63
			Messages	21	32	147	284	441	3087
Small	10	15	Test Cases	50	75	150	150	150	150
			Messages	50	75	500	1125	1500	15000
Med	15	30	Test Cases	150	225	450	450	450	450
			Messages	150	225	2250	6750	6750	101250
Large	30	100	Test Cases	1000	1500	3000	3000	3000	3000
			Messages	1000	1500	30000	150000	90000	2700000

Key: n = number of states, k = number of events
(1) Assumes average number of events per test case = $k/2$.
(2) Assumes average number of events per test case = $k/3$.
(3) Worst case minimum, approximately $n^2 \times k$ [Chow 78].
(4) Worst case maximum, approximately $n^3 \times k$ [Chow 78].

Appendix
More on State Machine Coverage

28

ISO 9646 Conformance Tests

- Developed as a standard for conformance testing of communications protocols that used an FSM model to describe protocol behaviour.
- For a particular protocol, this is a standard methodology for deriving a set of tests that must pass in order for an implementation to be certified for use.
- Two objectives:
  - Check that the implementation has states corresponding to the model.
  - Check that the implementation has transitions corresponding to the model.

29

ISO 9646 Conformance Tests

- A test consists of:
  - Preamble: Series of events that take the system from a known / initial state to a targeted state, or the state at the head of the transition being targeted.
  - Event: Cause the system to take the targeted transition.
  - Response: Check that the appropriate actions for the transition are taken, based on observed response.
  - Next state verification: Collect evidence that the implementation is now in the correct state.
    - How to do this?
  - Postamble: Put the implementation into a known state (to set up the next test).

30

Indirect state verification

- If all we can observe are responses from the implementation, we can try to observe event-response combinations that could only have happened if we were in the desired state.
- Example: Suppose we are in an unknown state, and the following event / response sequence is performed.
  - pop / error
    - This could only have happened if the state we were in initially was Empty.
- Suppose we have a transition x / y that should lead to the state Empty.
  - The sequence x / y ; pop / error will check the transition's action (y) in response to event x, and that after the transition is taken, the transition entered the correct state.
  - If we see a response other than error to the pop event, then the transition x / y must have gone to an incorrect state.

31

Indirect state verification

- Distinguishing sequence (DS): a sequence of inputs that produces a different sequence of responses for every state in the implementation.
  - This is the most useful property if there is such a sequence: apply the DS and identify the state from the response.
  - No guarantee that a DS exists
- Unique input/output sequence (UIO): for a given state, there is a sequence of inputs that produces a response sequence different from any other state.
  - This can only provide a yes/no answer to the question, "am I in the given state?"
  - No guarantee that a UIO sequence exists – but it is more likely to exist than a DS.

32

Preambles and Postambles

- Preambles:
  - Objective: take the system from the designated start state to any targeted state in the system, in order to set up a test.
  - Requirements: a sequence of events that will move the system to a specified state.
  - The start point of a preamble is fixed; the state where a preamble ends will vary depending on the test purpose.
- Postamble:
  - Objective: take the system from the state in which it is believed the system is in, to a final state or the common start/end state.
  - Requirements: a sequence of events that will move the system to a designated final state, in order to leave the system in a known state.
  - The start point of a postamble will depend on which state the system is in after state verification – which may not necessarily be the end state of a targeted transition.

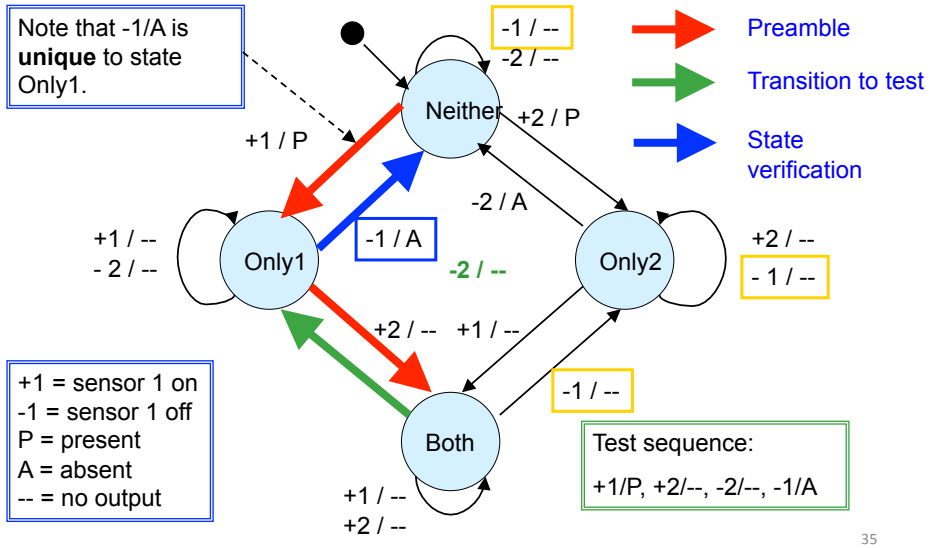
33

Sensor example

- Two motion sensors, one at door 1 and the other at door 2. When someone enters the detection range, a sensor reports (+). When someone leaves the detection range, a sensor reports (-).
- Integrate the two sensor inputs to produce:
  - P (present) event when someone enters the area
  - A (absent) event when the area becomes clear.

34

Sensor example



35

Developing Conformance Tests

- Step 1: Determine if a state verification sequence exists.
  - A DS to identify all states
  - A UIO sequence for each state.
- Step 2: Determine the start state for testing.
  - All test cases will start in this state.
  - All test cases will end in this state, if the system is intended to run continuously.
  - Choose this to be a state which is easily reachable from any other state, as well as on system start-up.
    - Usually, this is the state after the initial transition, and represents a "ready" or "idle" state.

36

Developing Conformance Tests (2)

- Step 3: Determine a path from the start state to each of the other states.
  - Path includes set of events and expected observable responses.
  - This will allow for preambles to put the system in a known state at the start of any transition (except the initial transition).
- Step 4: Determine a path from each state to ...
  - ... the start state, for a continuously running system.
  - ... OR a final state, if such states are present.
  - Choose a path that is as short as possible.
  - This will allow for postambles, to return the system to a known state, or to terminate the system gracefully.
  - Objective: set system up for the following test.

37

Developing Conformance Tests (3)

- Step 5: Determine if we can observe a distinct set of states.
  - This will be done by checking that the distinguishing sequence OR unique input-output sequences derived from the state model actually work in the implementation
  - See next slide for how to do this.
  - This will allow us to discover if the implementation has "at least" as many states as the state model.
    - Does not detect the presence of extra states in the implementation.
- If these test paths do not result in test cases that pass, the remainder of the test cases that will be created in step 6 are not ready to execute.

38

Developing Conformance Tests (4)

- Step 5 [continued]
- Objective: verify that we can observe distinct states.
- To check that a distinguishing sequence really distinguishes the states, use {preamble, DS, postamble} for each of the states where DS is the distinguishing sequence input and the expected output corresponds to the state.
- To check that the unique-input-output (UIO) sequences distinguish the states:
  - for each state S do:
    - for each state T do:
      - {preamble(T), UIO(S,T), postamble}
  - In the above, UIO(S,T) represents applying the UIO input for state S, but with expected results for state T.

39

Developing Conformance Tests (5)

- Step 6: Create a test case path for each transition, consisting of:
  A. The preamble for the state from which the transition exits.
  B. The event and expected response for the transition.
  C. The state verification sequence for the "next state" after the transition.
  D. The postamble for the state expected to be reached at the end of the state verification sequence.

40

Error model: What is detected

- Using the ISO 9646 conformance test approach will detect:
  - All implementations with fewer states than the model.
  - All implementations with an output error
    - An incorrect observable response on a transition.
  - All implementations with a next state error
    - A transition that has the correct response, but goes to an incorrect state.
  - All implementations with a missing transition.

41

Error model: What may not be detected

- What may not be detected:
  - Incorrect non-observable actions on transitions (e.g. problems with state variables)
  - Implementations with more states than the model.
  - Extra transitions, if the model is not completely specified
    - Completely-specified: transitions are present for all legal events in every state.
  - Illegal transitions: acceptance of non-legal events.

42

Sensor Example

- For each state, we need:
  - A preamble leading to the state
  - A sequence that will verify that
    - Also record the state at the end of the sequence, to select the state postamble
  - A postamble taking us to a known state (in this case, "Neither").

State	Preamble	State verification		Postamble
		Sequence	End state	
Neither	[none]	+1 / P	Only1	[none]
Only1	+1 / P	-1 / A	Neither	-1 / A
Only2	+2 / P	-2 / A	Neither	-2 / A
Both	+1 / P,+2 / --	-2 / --, -1 / A -1 / --, -2 / A see next slide	Neither	-2 / --, -1 / A

43

State verification for "both"

- It turns out that there is not a single sequence that we can use to verify the state "both".
- However:
  - -2 / --, -1 / A reduces the possibilities to "both" or "only1"
  - -1 / --, -2 / A reduces the possibilities to "both" or "only2"
- We can use the combination of the two sequences to reduce the possible states to just "both". If the two sequences pass, we must have been in our desired state.
  - However, we need to be at the end of the transition being checked before executing each of the state verification sequences.
  - This means that we have to use two test cases for the transition:
    - Preamble, transition, state verification 1, postamble
    - Preamble, transition, state verification 2, postamble
  - Both test cases have to pass to verify the transition.

44

Check the states

- This set of tests will identify the existence of these states in the actual implementation
- Each row in the table represents a test case.
- Since we are using UIO sequences for state verification, we will have to apply each UIO sequence to each state.
  - Results in a minimum of n^2 tests for n states in the model.
  - State "both" has 2 partial UIO sequences, so we have to verify the combination is distinctive.
    - This means applying both UIO sequences to every state.
    - Result: more than n^2 test cases to check states for the sensor example.

45

Check the states (1/3)

Test ID	UIO for (S)	UIO applied to (T)	Preamble (T)	State Verification (S)	Postamble
S1-1	Neither	Neither	[none]	+1 / P	-1 / A
S1-2	Neither	Only1	+1 / P	+1 / --	-1 / A
S1-3	Neither	Only2	+2 / P	+1 / --	-1 / -- -2 / A
S1-4	Neither	Both	+1 / P +2 / --	+1 / --	-1 / -- -2 / A
S2-1	Only1	Neither	[none]	-1 / --	[none]
S2-2	Only1	Only1	+1 / P	-1 / A	[none]
S2-3	Only1	Only2	+2 / P	-1 / --	-2 / A
S2-4	Only1	Both	+1 / P +2 / --	-1 / --	-2 / A

46

Check the states (2/3)

Test ID	UIO for (S)	UIO applied to (T)	Preamble (T)	State Verification (S)	Postamble
S3-1	Only2	Neither	[none]	-2 / --	[none]
S3-2	Only2	Only1	+1 / P	-2 / --	-1 / A
S3-3	Only2	Only2	+2 / P	-2 / A	[none]
S3-4	Only2	Both	+1 / P +2 / --	-2 / --	-1 / A

47

Check the states (3/3)

Test ID	UIO for (S)	UIO applied to (T)	Preamble (T)	State Verification (S)	Postamble
S4-1A	Both	Neither	[none]	-2 / --, -1 / --	[none]
S4-1B			[none]	-1 / --, -2 / --	[none]
S4-2A	Both	Only1	+1 / P	-2 / --, -1 / A	[none]
S4-2B			+1 / P	-1 /A, -2 / --	[none]
S4-3A	Both	Only2	+2 / P	-2 /A, -1 / --	[none]
S4-3B			+2 / P	-1 / --, -2 / A	[none]
S4-4A	Both	Both	+1 / P +2 / --	-2 / --, -1 / A	[none]
S4-4B			+1 / P +2 / --	-1 / --, -2 / A	[none]

48

Check the transitions

- This set of tests will check the response on each transition, and also verify that the transition goes to the correct next state.
- Each row in the table represents a test case.
- For each transition, there will be at least one test case.
  - Results in a minimum of ***m*** tests for ***m*** transitions in the model.
- For any transition that is expected to end in the "both" state, two test cases will be needed as there are two partial state verification sequences for this state.

49

Check the transitions (1/2)

Test ID	State-event combination	Preamble	Transition	State verification	Postamble
T1	initial, --			+1 / P	-1 / A
T2	neither, +1		+1 / P	-1 / A	
T3	neither, +2		+2 / P	-2 / A	
T4	neither, -1		-1 / --	+1 / P	-1 / A
T5	neither, -2		-2 / --	+1 / P	-1 / A
T6	only1, +1	+1 / P	+1 / --	-1 / A	
T7A	only1, +2	+1 / P	+2 / --	-1 / --, -2 / A	
T7B		+1 / P	+2 / --	-2 / --, -1 / A	
T8	only1, -1	+1 / P	-1 / A	+1 / P	-1 / A
T9	only1, -2	+1 / P	-2 / --	-1 / A	

50

Check the transitions (2/2)

Test ID	State-event combination	Preamble	Transition	State verification	Postamble
T10A	only2, +1	+2 / P	+1 / --	-1 / --, -2 / A	
T10B		+2 / P	+1 / --	-2 / --, -1 / A	
T11	only2, +2	+2 / P	+2 / --	-2 / A	
T12	only2, -1	+2 / P	-1 / --	-2 / A	
T13	only2, -2	+2 / P	-2 / A	+1 / P	-1 / A
T14A	both, +1	+1 / P, +2 / --	+1 / --	-1 / --, -2 / A	
T14B		+1 / P, +2 / --	+1 / --	-2 / --, -1 / A	
T15A	both, +2	+1 / P, +2 / --	+2 / --	-1 / --, -2 / A	
T15B		+1 / P, +2 / --	+2 / --	-2 / --, -1 / A	
T16	both, -1	+1 / P, +2 / --	-1 / --	-2 / A	
T17	both, -2	+1 / P, +2 / --	-2 / --	-1 / A	

The conformance test suite

- 41 test cases: 20 to check states, and 21 to check transitions.
- If an incorrect response is observed during the execution of the preamble of a test case, the verdict should be declared as "inconclusive"
  - Some other test will target the transition on the preamble that didn't work. This test case would fail.
  - Doing this will better identify the location of problems.
- What about an incorrect response during the postamble?
- Something to think about…
  - If the last few steps of a test are the same as the first few steps of some other test, could you overlap the tests? What are the implications?

52

Invalid event testing

- When not all events are specified at every state, or there is a guard condition case that is not covered, invalid event testing is appropriate.
- Guard condition example:
  - Suppose that at some state there is a transition specified for `[x = true]`, but there is no transition specified at that state for `[x = false]`
- Objective: try the unspecified events to see what happens.

53

Creating invalid event tests

- Unspecified events:
  - Use the appropriate state preamble to put the system into the state with the unspecified event.
  - Apply the unspecified event.
  - Check that the implementation reacts "appropriately". Depending on the situation, this could mean that:
    - The event is ignored with no action.
    - The event is rejected with an error action.
  - Check the implementation's state after the invalid event. Depending on the situation, this could be:
    - The same state as when the event was applied (especially, if the event is ignored).
    - An error state.

54

Creating invalid event tests

- Missing guard conditions.
  - Create a path that will do the following.
    1. Set up the guard condition that was not specified.
    2. Put the implementation in the state with the missing guard condition.
    3. Apply the appropriate event associated with the missing guard condition (if required).
  - Check that the implementation reacts "appropriately". Depending on the situation, this could mean that:
    - The event is ignored with no action.
    - The event is rejected with an error action.
  - Check the implementation's state after the invalid event. Depending on the situation, this could be:
    - The same state as when the event was applied (especially, if the event is ignored).
    - An error state.

55