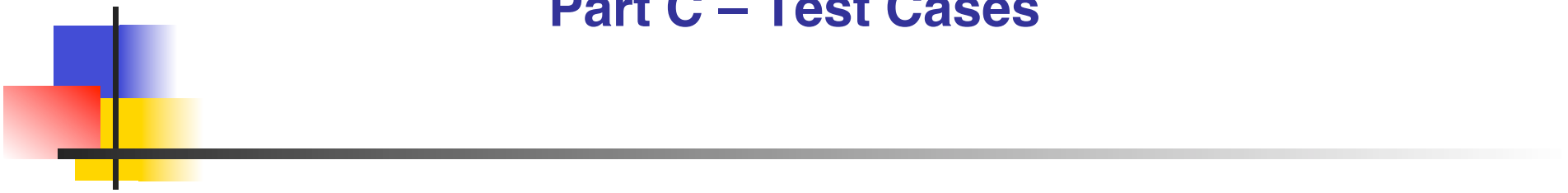
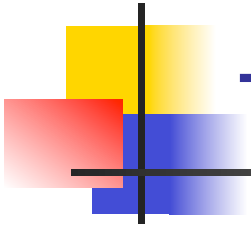


State-Based Testing
Part C – Test Cases

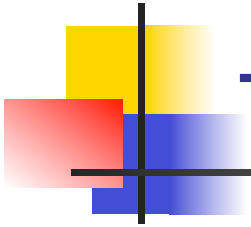
Generating test cases for complex behaviour

Reference: Robert V. Binder
Testing Object-Oriented Systems: Models, Patterns, and Tools
Addison-Wesley, 2000, Chapter 7



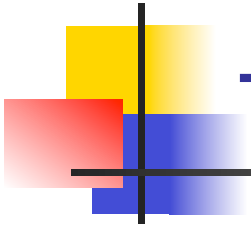
Test strategies

- Exhaustive
- All Transitions
  - **Every transition executed at least once**
  - **Exercises all transitions, states and actions**
  - **Cannot show incorrect state is a result**
  - **Difficult to find sneak paths**



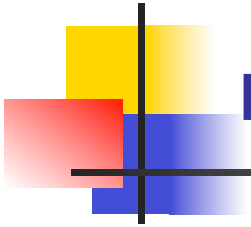
Test strategies – 2

- All n-transition sequences
  - **Can find some incorrect and corrupt states**
- All round trip paths
  - **Generated by N+ test strategy**
    - **What is a round trip path?**



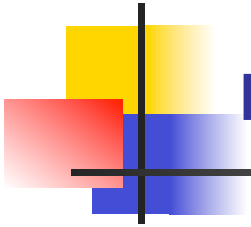
Test Strategies – 3

- All n-transition sequences
  - **Can find some incorrect and corrupt states**
- All round trip paths
  - **Generated by N+ test strategy**
  - **A prime path of nonzero length that starts and ends at the same node**
  - **N+ coverage**



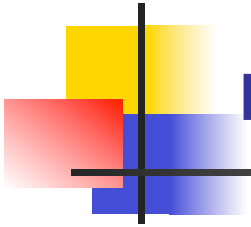
N+ test strategy overview

- Encompasses UML state models
- Testing considerations unique to OO implementations
- It uses a flattened model
- All implicit transitions are exercised to reveal sneak paths



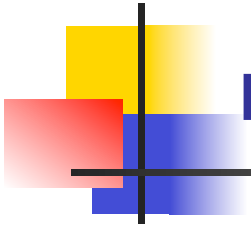
N+ test strategy overview – 2

- Relies on an the implementation to properly report resultant state
- More powerful than simpler state-based strategies
  - **Requires more analysis**
  - **Has larger test suites**
  - **Look at cost/benefit tradeoff**



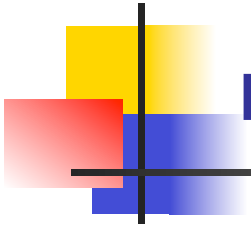
N+ coverage reveals

- All state control faults
- All sneak paths
- Many corrupt state bugs
- Because it exercises at flattened scope
  - **Many super-class / sub-class integration bugs**
  - **Subcontracting bugs**



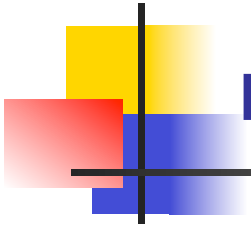
N+ coverage reveals – 2

- If more than one α transition exists, faults on each one
- All transitions to the ω states
- Can suggest presence of trap doors when used with program text coverage analyzer



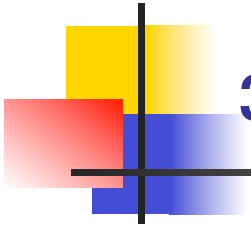
N+ test strategy development

- Develop a state-based model of the system
  - **Validate the model using the checklists**
  - **Flatten the model – Expand the statechart**
  - **Develop the response matrix**
- Generate the round-trip path tree
- Generate the round-trip path test cases



N+ test strategy development – 2

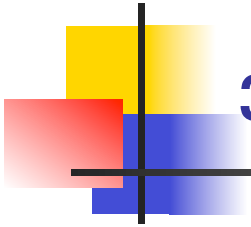
- Generate the sneak path test cases
- Sensitize the transitions in each test case
  - **Find input values to satisfy guards for the transitions in the event path**
    - **Similar to finding path conditions in path testing**



3-player game example

- We will use an extension of the 2-player game as an example

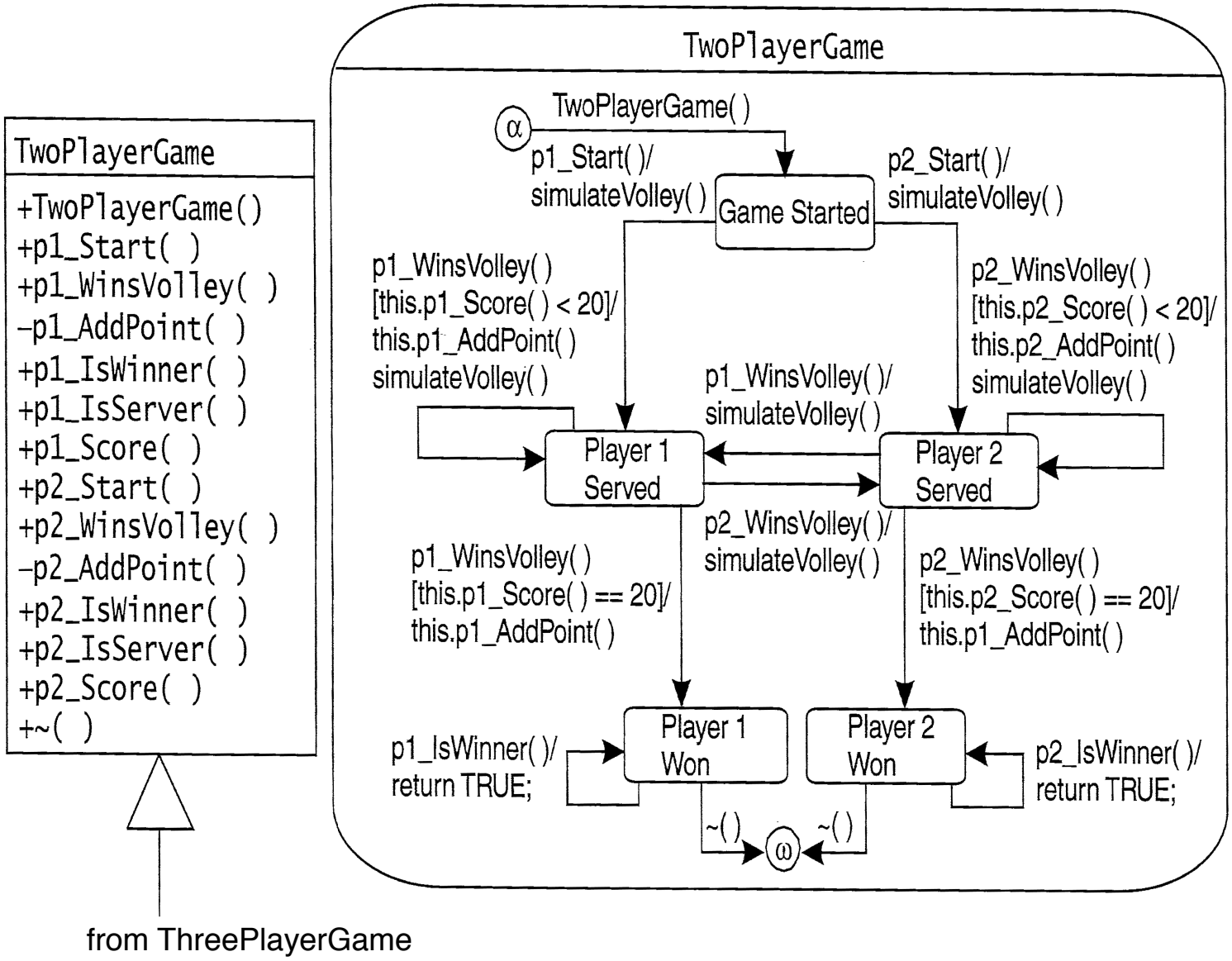
- There is now a third player that may win any of the volleys



3-player game Java interface

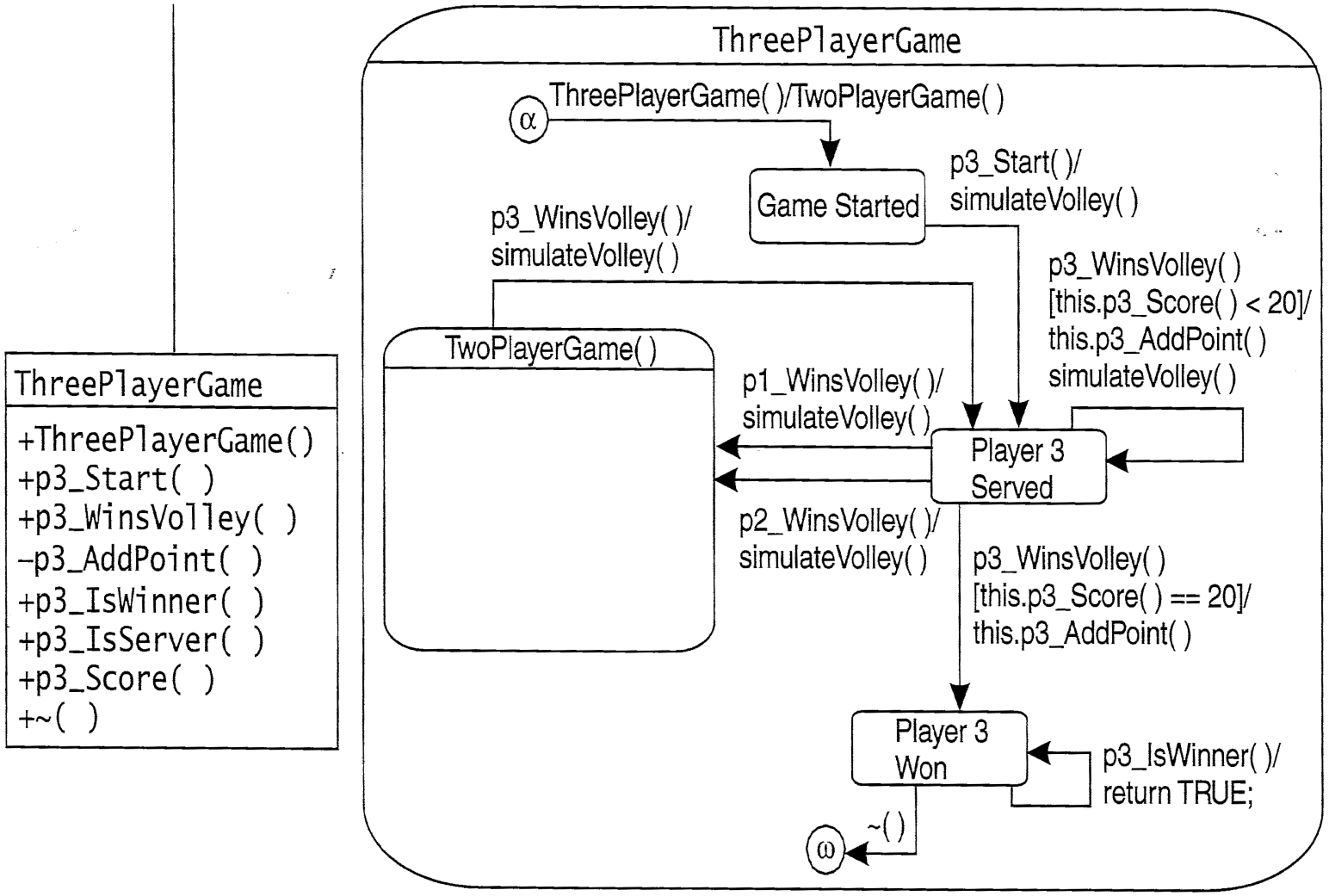
```
class ThreePlayerGame extends TwoPlayerGame {

  private int   p3_points;
  public        ThreePlayerGame()     // Constructor
  public void p3_start()              // P3 serves first
  public void p3_WinsVolley()         // P3 ends the volley
  public void p3_AddPoint()           // Add 1 to P3's score
  public boolean p3_isWinner()        // True if P3's score is 21
  public boolean p3_isServer()        // True if P3 is server
  public int   p3_score()             // Returns p3's score

}
```

TwoPlayerGame statechart

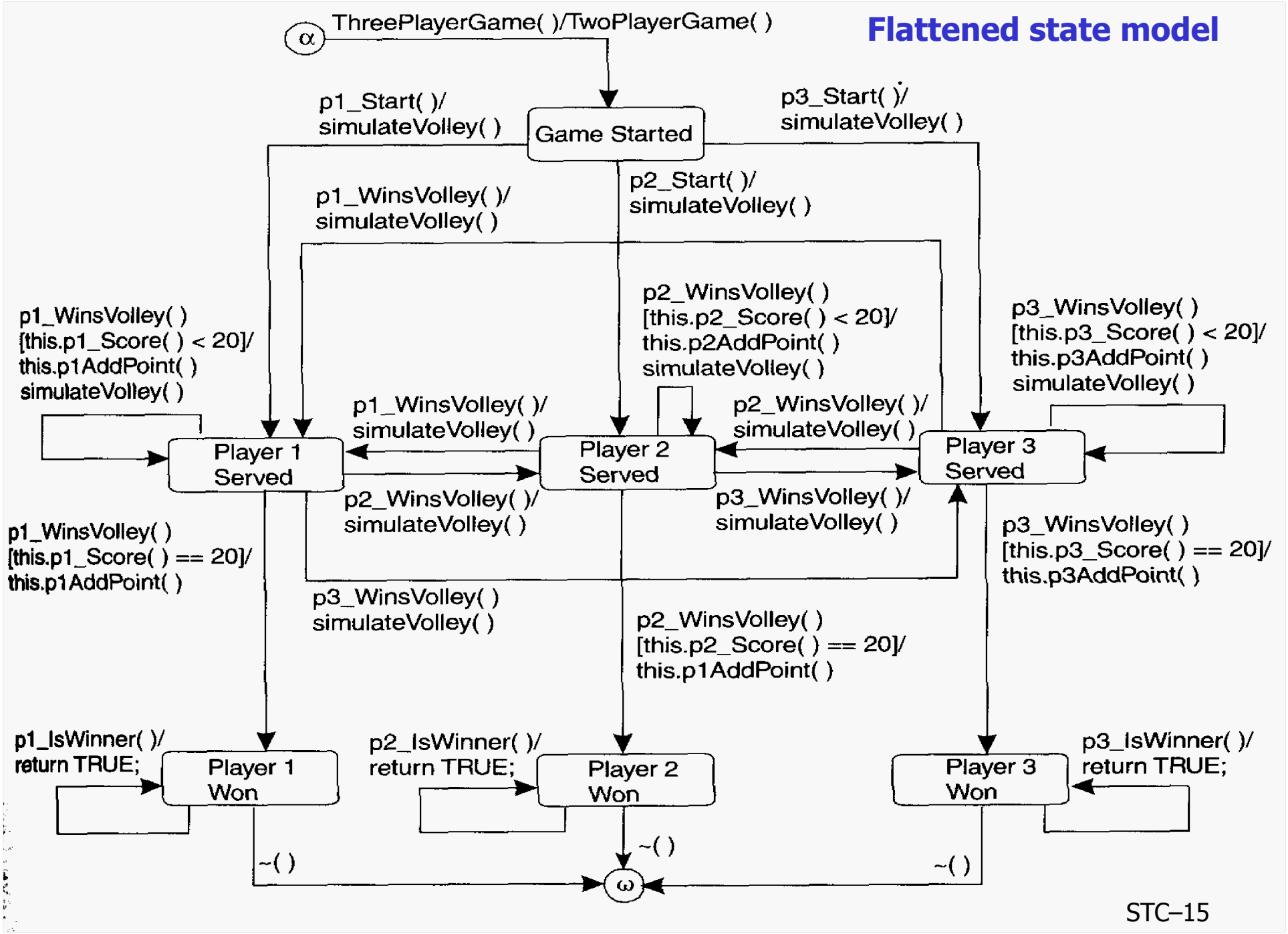


ThreePlayerGame statechart

to TwoPlayerGame



Flattened state model



Response matrix

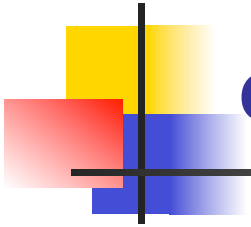
Events and Guards			Accepting State/Expected Response								
			α	Game Started	Player 1 Served	Player 2 Served	Player 3 Served	Player 1 Won	Player 2 Won	Player 3 Won	ω
ctor			✓	6	6	6	6	6	6	6	6
p1_Start				✓	4	4	4	4	4	4	6
p2_Start				✓	4	4	4	4	4	4	6
p3_Start				✓	4	4	4	4	4	4	6
p1_WinsVolley	p1_score < 20	p1_Score == 20									
	DC	DC		4		✓	✓	4	4	4	6
	F	F			6						
	F	T			✓						
	T	F			✓						
	T	T									
p2_WinsVolley	p2_score < 20	p2_Score == 20									
	DC	DC		4	✓		✓	4	4	4	6
	F	F				6					
	F	T				✓					
	T	F				✓					
	T	T									
p3_WinsVolley	p3_score < 20	p3_Score == 20									
	DC	DC		4	✓	✓		4	4	4	6
	F	F					6				
	F	T					✓				
	T	F					✓				
	T	T									
p1_isWinner				✓	✓	✓	✓	✓	✓	✓	6
p2_isWinner				✓	✓	✓	✓	✓	✓	✓	6
p3_isWinner				✓	✓	✓	✓	✓	✓	✓	6
Other Public Accessors				✓	✓	✓	✓	✓	✓	✓	6
dtor				✓	✓	✓	✓	✓	✓	✓	6

See key in slide SEI-11

Possible responses to illegal events

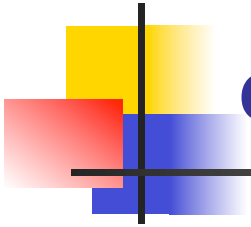
TABLE 7.3 Response Codes for Illegal Events

Response Code	Name	Response
0	Accept	Perform the explicitly specified transition
1	Queue	Place the illegal event in a queue for subsequent evaluation and ignore
2	Ignore	No action or state change is to be produced, no error is returned, no exception raised
3	Flag	Return a nonzero error code
4	Reject	Raise an `IllegalEventException`
5	Mute	Disable the source of the event and ignore
6	Abend	Invoke abnormal termination services (e.g., core dump) and halt the process



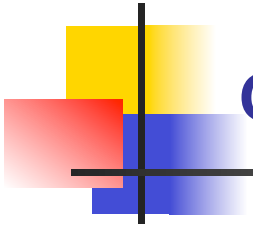
Generate Round-Trip Path Tree (GRTPT)

- Root
  - **Initial state – use α state with multiple constructors**
- First edges
  - **Draw for each transition out of initial state and add node for resultant state**



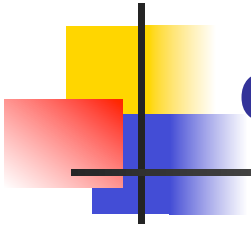
GRTPT – 2

- Remaining edges
  - **Draw for each transition out of a leaf node and add node for resultant state**
  - **Mark new leaf nodes as terminal nodes, if new leaf is**
    - **Already in the tree**
    - **A final state**
    - **An ω state**



GRTPT– Traversing the FSM

- **How can one traverse a FSM?**

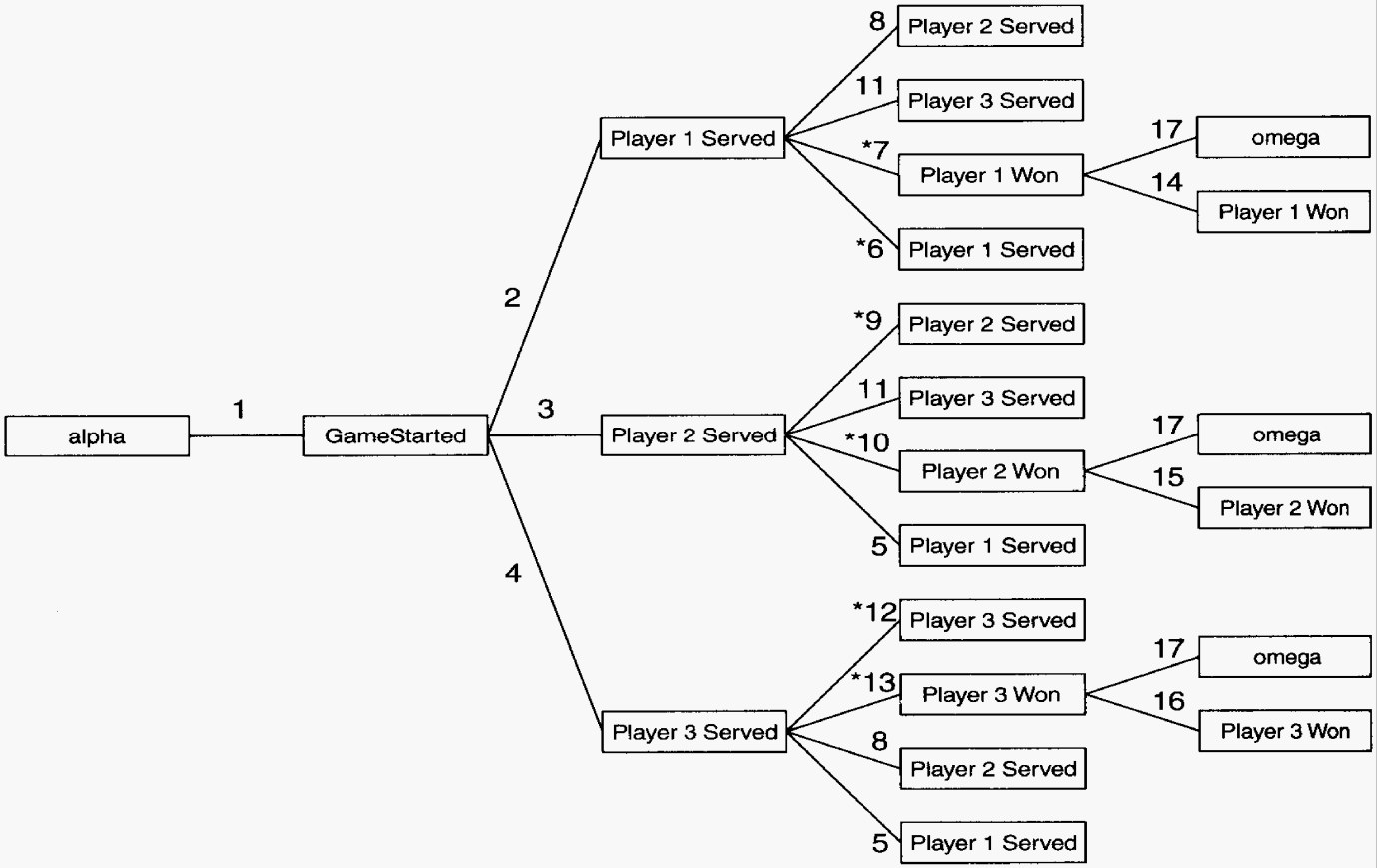


GRTPT– Traversing the FSM

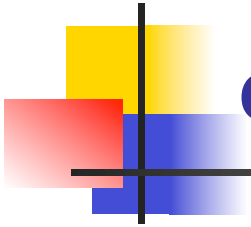
- Breadth-first
  - **Many short test sequences**
- Depth-first
  - **Fewer long test sequences**

Transition tree for the 3-player game

1 ThreePlayerGame( )
2 p1_Start( )
3 p2_Start( )
4 p3_Start( )
5 p1_WinsVolley( )
6 p1_WinsVolley( )[this.p1_Score( ) < 20]
7 p1_WinsVolley( ) [this.p1_Score( ) == 20]
8 p2_WinsVolley( )
9 p2_WinsVolley( ) [this.p2_Score( ) < 20]
10 p2_WinsVolley( ) [this.p2_Score( ) == 20]

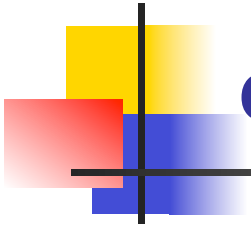


11 p3_WinsVolley( )
12 p3_WinsVolley( ) [this.p3_Score( ) < 20]
13 p3_WinsVolley( ) [this.p3_Score( ) == 20]
14 p1_IsWinner( )
15 p2_IsWinner( )
16 p3_IsWinner( )
17 ~( )



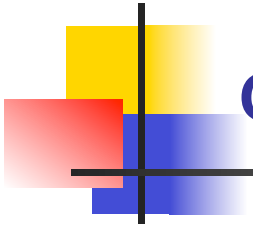
Guarded transitions – model true conditions

- If several conditional variants can make a guard true, transcribe one transition for each variant
  - **Add new transition to the tree**
  - **Guard is a simple Boolean expression, or contains only logical "and"**
    - **Then only one transition is needed**
      - **[x = 0]**
      - **[(x = 0) and (z != 42)]**



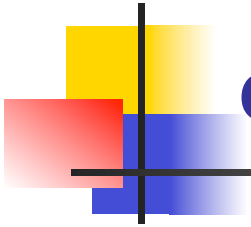
Guarded transitions – model true conditions – 2

- **Guard is compound Boolean expression with at least one logical "or" operator**
  - **Then one transition is required for each predicate combination that yields a true result**
  - **[x = 0] or [z != 42]**
    - Need true / false and false / true



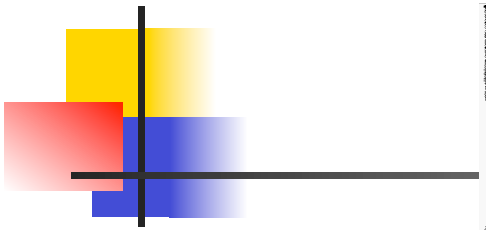
Guarded transitions – model true conditions – 3

- Guard specifies a relationship that occurs only after repeating some event such as **[counter ≥ 10]**
  - Test sequence requires at least the number of iterations to satisfy the condition.
  - The transition is graphed with a single arc annotated with an asterisk.



Guarded transitions – model false conditions

- Model at least one false combination
- Models to cover each guard's false variants are developed for the sneak attack tests
  - **Recall variant testing for decision tables**
    - **There are other variations**

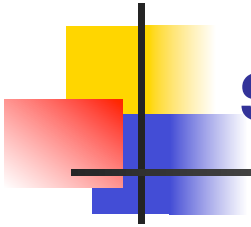


Generated test cases part 1

	Test Case Input		Expected Result	
TCID	Event	Test Condition	Action	State
1.1	ThreePlayerGame			GameStarted
1.2	p1_start		simulateVolley	Player 1 Served
1.3	p2_WinsVolley		simulateVolley	Player 2 Served
2.1	ThreePlayerGame			GameStarted
2.2	p1_start		simulateVolley	Player 1 Served
2.3	p3_WinsVolley		simulateVolley	Player 3 Served
3.1	ThreePlayerGame			GameStarted
3.2	p1_start		simulateVolley	Player 1 Served
3.3	*		*	Player 1 Served
3.4	p1_WinsVolley	p1_Score == 20		Player 1 Won
3.5	dtor			omega
4.1	ThreePlayerGame			GameStarted
4.2	p1_start		simulateVolley	Player 1 Served
4.3	*		*	Player 1 Served
4.4	p1_WinsVolley	p1_Score == 20		Player 1 Won
4.5	p1_IsWinner		return TRUE	Player 1 Won
5.1	ThreePlayerGame			GameStarted
5.2	p1_start		simulateVolley	Player 1 Served
5.3	*		*	Player 1 Served
5.4	p1_WinsVolley	p1_Score == 19	simulateVolley	Player 1 Served
6.1	ThreePlayerGame			GameStarted
6.2	p2_start		simulateVolley	Player 2 Served
6.3	*		*	Player 2 Served
6.4	p2_WinsVolley	p2_Score == 19	simulateVolley	Player 2 Served
7.1	ThreePlayerGame			GameStarted
7.2	p2_start		simulateVolley	Player 2 Served
7.3	p3_WinsVolley		simulateVolley	Player 3 Served
8.1	ThreePlayerGame			GameStarted
8.2	p2_start		simulateVolley	Player 2 Served
8.3	*		*	Player 2 Served
8.4	p2_WinsVolley	p2_Score == 20		Player 2 Won
8.5	dtor			omega

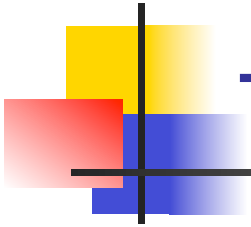
Generated test cases part 2

	Test Case Input		Expected Result	
TCID	Event	Test Condition	Action	State
9.1	ThreePlayerGame			GameStarted
9.2	p2_start		simulateVolley	Player 2 Served
9.3	*		*	Player 2 Served
9.4	p2_WinsVolley	p2_Score == 20		Player 2 Won
9.5	p2_IsWinner		return TRUE	Player 2 Won
10.1	ThreePlayerGame			GameStarted
10.2	p2_start		simulateVolley	Player 2 Served
10.3	p2_WinsVolley		simulateVolley	Player 2 Served
11.1	ThreePlayerGame			GameStarted
11.2	p3_start		simulateVolley	Player 3 Served
11.3	*		*	Player 3 Served
11.4	p3_WinsVolley	p3_Score == 19	simulateVolley	Player 3 Served
12.1	ThreePlayerGame			GameStarted
12.2	p3_start		simulateVolley	Player 3 Served
12.3	*		*	Player 3 Served
12.4	p3_WinsVolley	p3_Score == 20		Player 3 Won
12.5	dtor			omega
13.1	ThreePlayerGame			GameStarted
13.2	p3_start		simulateVolley	Player 3 Served
13.3	*		*	Player 3 Served
13.4	p3_WinsVolley	p3_Score == 20		Player 3 Won
13.5	p3_IsWinner		return TRUE	Player 3 Won
14.1	ThreePlayerGame			GameStarted
14.2	p3_start		simulateVolley	Player 3 Served
14.3	p2_WinsVolley		simulateVolley	Player 2 Served
15.1	ThreePlayerGame			GameStarted
15.2	p3_start		simulateVolley	Player 3 Served
15.3	p1_WinsVolley		simulateVolley	Player 1 Served



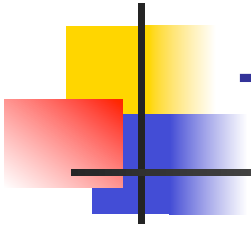
Sneak path testing

- Look for Illegal transitions and evading guards
- Transition tree tests explicit behaviour
- We need to test each state's illegal events
- A test case for each non-checked, non-excluded transition cell in the response matrix
- Confirm that the actual response matches the specified response



Testing one sneak path

- Put IUT (Implementation Under Test) into the corresponding state
  - **May need to have a special built-in test method, as getting there may take too long or be unstable**
  - **Can use any debugged test sequences that reach the state**
    - **Be careful if there are changes in the test suite**

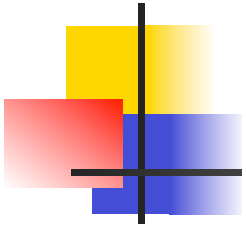


Testing one sneak path – 2

- Apply the illegal event by sending a message or forcing the virtual machine to generate the desired event
- Check that the actual response matches the specified response
- Check that the resultant state is unchanged
  - **Sometimes a new concrete state is acceptable**
- Test passes if response and resultant state are as expected

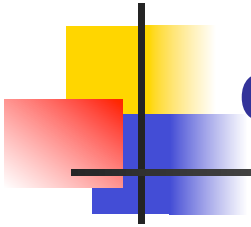
Sneak Path Test Suite Part 1

Test Case				Expected Result	
TCID	Setup Sequence	Test State	Test Event	Code	Action
16.0	ThreePlayerGame	Game Started	ThreePlayerGame	6	Abend
17.0	ThreePlayerGame	Game Started	p1_WinsVolley	4	IllegalEventException
18.0	ThreePlayerGame	Game Started	p2_WinsVolley	4	IllegalEventException
19.0	ThreePlayerGame	Game Started	p3_WinsVolley	4	IllegalEventException
20.0	10.0	Player 1 Served	ThreePlayerGame	6	Abend
21.0	5.0	Player 1 Served	p1_start	4	IllegalEventException
22.0	10.0	Player 1 Served	p2_start	4	IllegalEventException
23.0	5.0	Player 1 Served	p3_start	4	IllegalEventException
24.0	1.0	Player 2 Served	ThreePlayerGame	6	Abend
25.0	6.0	Player 2 Served	p1_start	4	IllegalEventException
26.0	1.0	Player 2 Served	p2_start	4	IllegalEventException
27.0	6.0	Player 2 Served	p3_start	4	IllegalEventException
28.0	7.0	Player 3 Served	ThreePlayerGame	6	Abend
29.0	2.0	Player 3 Served	p1_start	4	IllegalEventException



Sneak Path Test Suite Part 2

Test Case				Expected Result	
TCID	Setup Sequence	Test State	Test Event	Code	Action
30.0	7.0	Player 3 Served	p2_start	4	IllegalEventException
31.0	2.0	Player 3 Served	p3_start	4	IllegalEventException
32.0	4.0	Player 1 Won	ThreePlayerGame	6	Abend
33.0	4.0	Player 1 Won	p1_start	4	IllegalEventException
34.0	4.0	Player 1 Won	p2_start	4	IllegalEventException
35.0	4.0	Player 1 Won	p3_start	4	IllegalEventException
36.0	4.0	Player 1 Won	p1_WinsVolley	4	IllegalEventException
37.0	4.0	Player 1 Won	p2_WinsVolley	4	IllegalEventException
38.0	4.0	Player 1 Won	p3_WinsVolley	4	IllegalEventException
39.0	9.0	Player 2 Won	ThreePlayerGame	6	Abend
40.0	9.0	Player 2 Won	p1_start	4	IllegalEventException
41.0	9.0	Player 2 Won	p2_start	4	IllegalEventException
42.0	9.0	Player 2 Won	p3_start	4	IllegalEventException
43.0	9.0	Player 2 Won	p1_WinsVolley	4	IllegalEventException
44.0	9.0	Player 2 Won	p2_WinsVolley	4	IllegalEventException
45.0	9.0	Player 2 Won	p3_WinsVolley	4	IllegalEventException
46.0	13.0	Player 3 Won	ThreePlayerGame	6	Abend
47.0	13.0	Player 3 Won	p1_start	4	IllegalEventException
48.0	13.0	Player 3 Won	p2_start	4	IllegalEventException
49.0	13.0	Player 3 Won	p3_start	4	IllegalEventException
50.0	13.0	Player 3 Won	p1_WinsVolley	4	IllegalEventException
51.0	13.0	Player 3 Won	p2_WinsVolley	4	IllegalEventException
52.0	13.0	Player 3 Won	p3_WinsVolley	4	IllegalEventException
53.0	12.0	omega	any	6	Abend

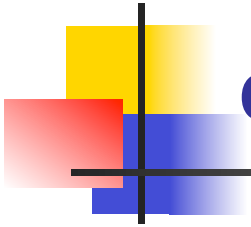


Checking Resultant state

- State reporter
  - **Can evaluate state invariant to determine state of object**
  - **Implement assertion functions**

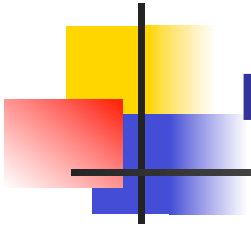
    bool isGameStarted() { … }
    - **After each event appropriate state reporter is asserted**

- Test repetition – good for corrupt states
  - **Repeat test and compare results**
  - **Corrupt states may not give the same result**
  - **Not as reliable as state reporter method**



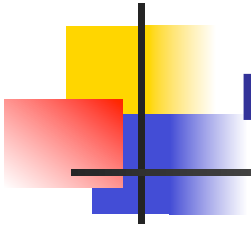
Checking Resultant state – 2

- State revealing signatures
  - **Identify and determine a signature sequence**
    - **A sequence of output events that are unique for the state**
    - **Analyze specification**
  - **Expensive and difficult**



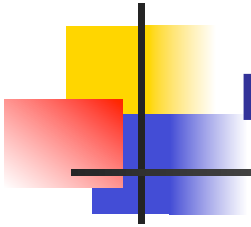
Major test strategies in increasing power

- Piecewise
  - **Every state, every event, every action at least once**
  - **Does not correspond to state model**
  - **Inadequate for testing**



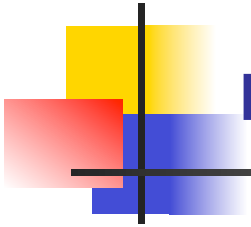
Major test strategies in increasing power – 2

- All transitions – minimum acceptable
  - **Every transition is exercised at least once**
  - **Implies all states, all events, all actions**
  - **Incorrect / Missing event / action pairs are guaranteed**
  - **Does not show incorrect state is a result**
  - **Unless completely specified, sneak paths are not found**



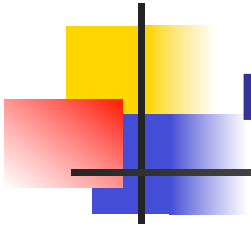
Major test strategies in increasing power – 3

- All transition k-tuples
  - **Exercise every transition sequence of k events at least once**
    - **1-tuple is equivalent to all transitions**
  - **Not necessarily all incorrect or corrupt states are found**



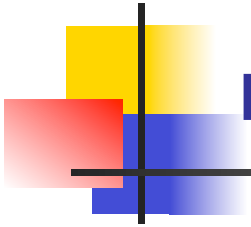
Major test strategies in increasing power – 4

- All round-trip paths
  - **Called N+ coverage**
  - **Shortest trip is to loop back once to the same state**
  - **The longest trip depends upon the structure of the FSM**
  - **Any sequence that goes beyond a round trip must be part of a sequence that belongs to another round trip**



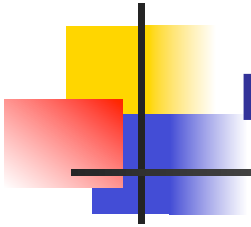
Major test strategies in increasing power – 5

- All round-trip paths – cont'd
  - **Finds all incorrect or missing event/action pairs**
  - **Can find some incorrect or invalid states**
    - **E.g. enter state that mimics correct behaviour for 10 events but becomes corrupt on the 11'th**
  - **N+ strategy relies on state inspector**



Major test strategies in increasing power – 6

- M-length signature
  - **Used for opaque systems – cannot determine current state**
  - **A state signature is used to determine the current state of the IUT**
    - **A sequence of output actions unique for the state**
    - **If the actual state signature is the expected one, then in the correct state**
  - **To find corrupt states, need to try sequences long enough to get beyond any possible number of corrupt states, which is guessed as being M**



Major test strategies in increasing power – 7

- Exhaustive

Test Suite Size

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].

Power comparison state-based testing strategies

Fault Class	Fault Revealing Power of Some State-based Testing Strategies						
	Observable					Opaque	
	Each Event Once	Each State Once	Each Action Once	All Transitions	N+ Cover	All Transitions	W-Method [Chow 78]
Guard fault	1			1	✓	1	1
Missing transition	3			✓	✓	✓	✓
Sneak path (Extra transition)				2	✓	2	✓
Incorrect action (wrong or missing)				✓	✓	✓	✓
Incorrect resultant state				✓	✓		✓
Missing state		✓		✓	✓		✓
Corrupt resultant state ("extra state")				✓	✓		5
Trap door (extra event)	4			4	4	6	6

Key: Blank = This fault class is not targeted, ✓ = All faults of this class will be revealed.

1. Other published strategies (see Bibliographic Notes) do not consider guards.
2. Guaranteed only if model is completely specified.
3. Guaranteed only if events and transitions are one-to-one.
4. Not certain to reveal. Requires post-test, call-path coverage analysis.
5. Will reveal up to *m* corrupt (extra) states.
6. Opaque assumption precludes coverage analysis.