

### 7.3.5 Inheritance and Class Flattening

#### Flattening the Class Hierarchy

Inheritance must be considered to decide the prudent extent of (re)testing for a subclass. The state model of a root superclass need not consider inherited features because there are none. Subclass behavior, however, is determined by both inherited and local features. If sequential constraints exist on two or more levels, the behavior (i.e., the state machine) of the lower levels is a composite of these levels. Our testing model would be incomplete without a representation of inherited behavior. To obtain this model for subclasses, the class hierarchy is flattened.

Statecharts are well suited to representing a flattened view of behavior. A subclass state space results from two factors. The superclass state space is inherited to the extent that superclass instance variables are visible. The subclass adds state space dimensions to the extent that it defines or overrides instance variables. An orthogonal extension adds new states while preserving superclass states. A well-formed<sup>23</sup> and testable modal class hierarchy should conform to the following type substitutability principles:

- A subclass may accept a superclass event in more states than the superclass. That is, a superclass state can be partitioned into substates.
- A subclass may define new states, but these must be orthogonal to the superclass states. The inherited superclass state space must not be spindled, folded, or mutilated by the extensions.
- The effect of superclass private variables must be orthogonal for states formed by private and protected variables. In other words, the effect of superclass private variables on the superclass state space must be additive.

The slogan “Require no more, ensure no less” summarizes these principles. That is, compared with the superclass, a subclass’s method preconditions

---

23. This discussion draws on the type-conformance rules used in Syntropy [Cook+94] and Liskov and Wing’s principles of subtyping [Liskov+94]. The requirements of a well-formed class hierarchy and relative strength and weakness of invariants are further discussed in Chapter 17. Briefly, subclass state invariants should be the same or more restrictive (stronger) with respect to inherited instance variables. The preconditions of overriding methods should be the same or less restrictive (weaker) than the overridden preconditions, and the postconditions of overriding methods should be the same or stronger than the overridden postconditions. A subclass can have states that do not exist in any superclass because it can declare its own variables, and therefore may add new states.

should not be more restrictive and its postconditions should not be less restrictive. The Venn diagram in Figure 7.18 suggests state value sets that conform and do not conform to these rules.

Inheritance may be used in many ways to produce subclass behavior. The following cases illustrate how inheritance can be used to implement subclass behavior. They show that although the statechart for a given subclass may be simple, its actual behavior can be complex and explained only by a flattened model. Clearly, inheritance may be used in many other conforming and non-conforming ways. These examples follow the type substitutability rules.

- Orthogonal composition. Figure 7.19 shows an example of subclass behavior resulting from orthogonal composition. Each subclass inherits all of the nonprivate superclass features without redefinition and defines some new local features.

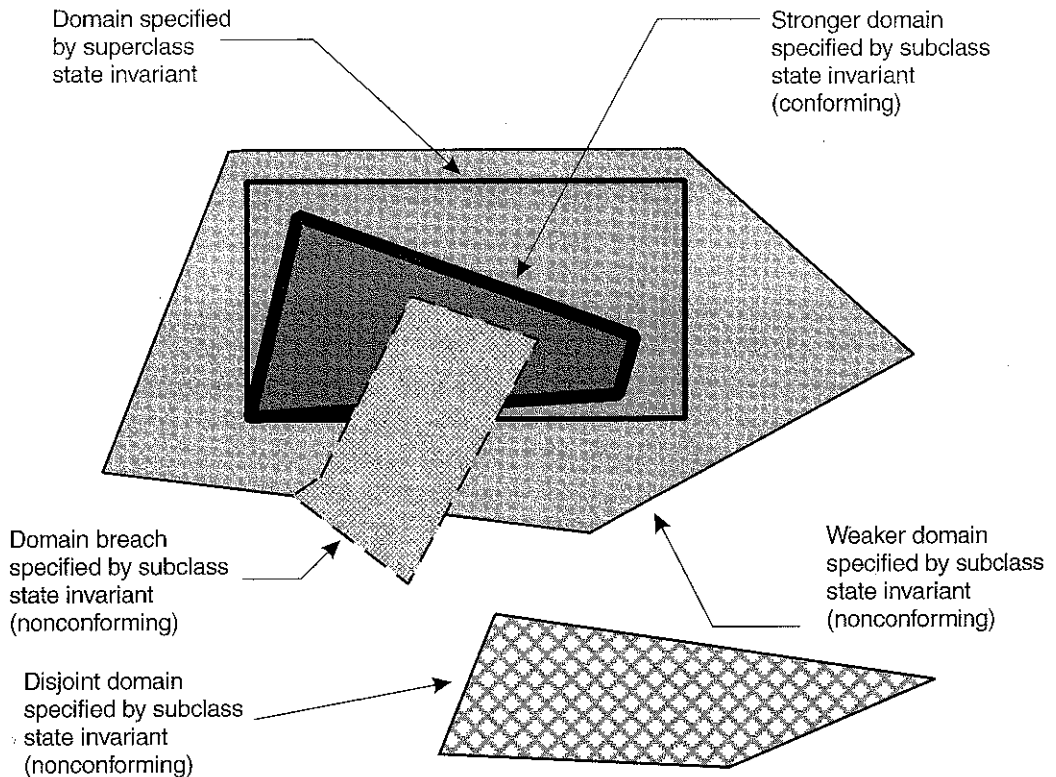


FIGURE 7.18 Conformance relationships between superstate and substate invariants.

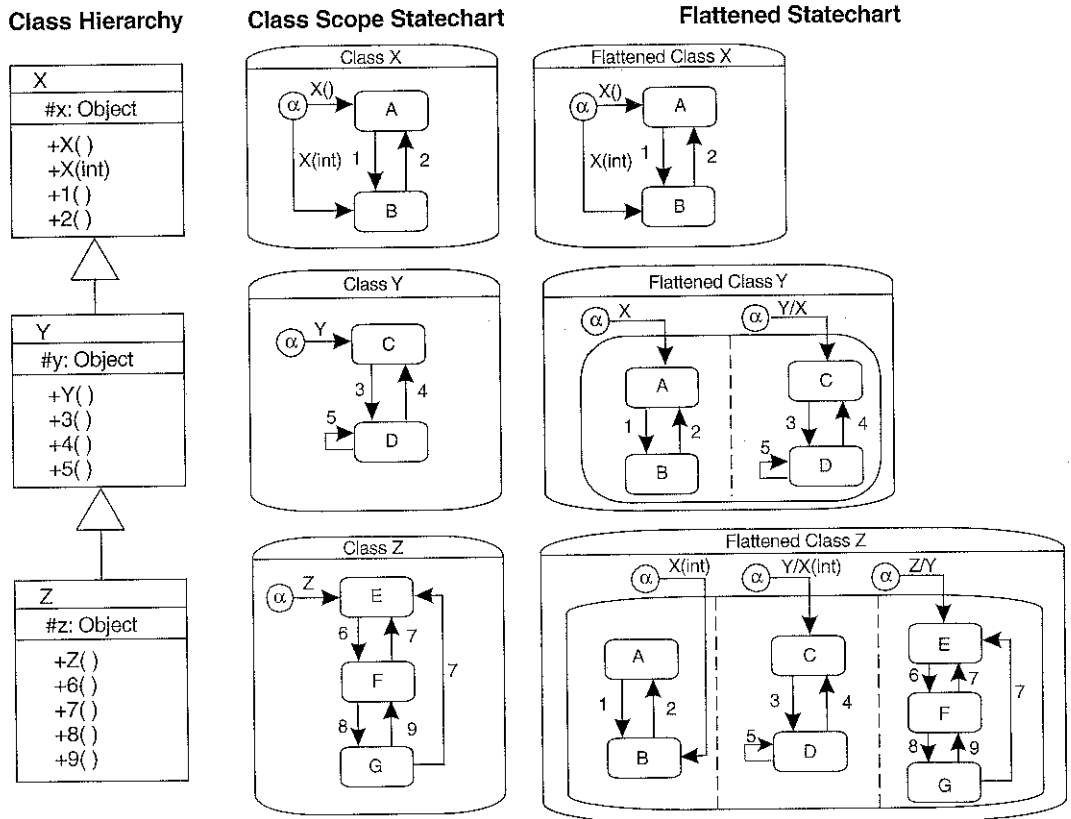
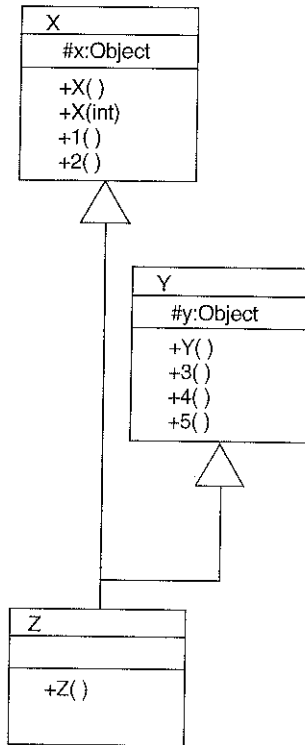


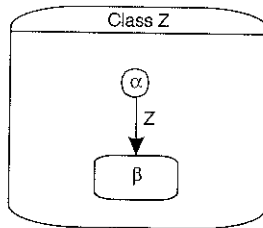
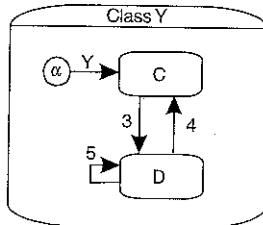
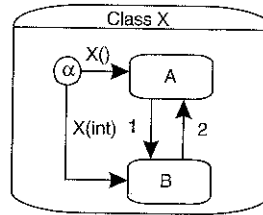
FIGURE 7.19 Flattened view of orthogonal composition.

- **Concatenation.** Figure 7.20 shows an example of subclass behavior resulting from concatenation. Concatenation involves the formation of a subclass that has no locally defined features other than the minimum requirements of a class definition. Mixin classes are an example of concatenation.
- **State partitioning and substate addition.** Figure 7.21 on page 219 provides an example of subclass behavior resulting from partitioning a superclass state. All superclass behavior for the state is preserved, and some new behavior is implemented. The new behavior is a result of partitioning the superclass state C and adding a new state J. This requires overriding some superclass methods.
- **Transition retargeting.** Figure 7.22 on page 220 shows an example of subclass behavior that results from partitioning a superclass state and changing

## Class Hierarchy



## Class Scope Statechart



## Flattened Statechart

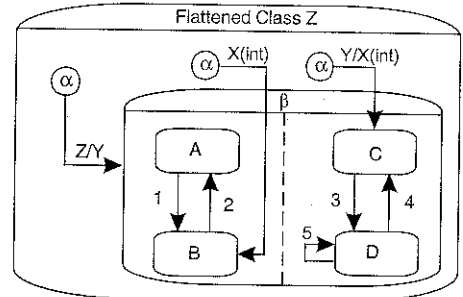
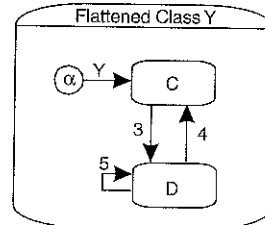
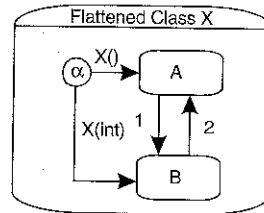


FIGURE 7.20 Flattened view of type concatenation.

the resultant state of some events. An object of class Z will reach substate *H* on event 4. *H* is a substate of *C*, which is reached when event 4 is accepted by an object of class Y. This requires overriding some superclass methods. Note that the postcondition and state invariant for the overriding method should be the same or more restrictive than those of the superclass.

- *Transition splitting.* Figure 7.23 on page 221 gives an example of superclass transitions redefined by guards. An object of class Z will reach substate *H* on event 4. *H* is a substate of *C*, which is reached when event 4 is accepted by an object of class Y. This requires overriding some superclass methods.

Multiple alpha transitions are used to model constructor behavior. Subclass constructors typically percolate up the hierarchy (see *Percolation*) so a complete

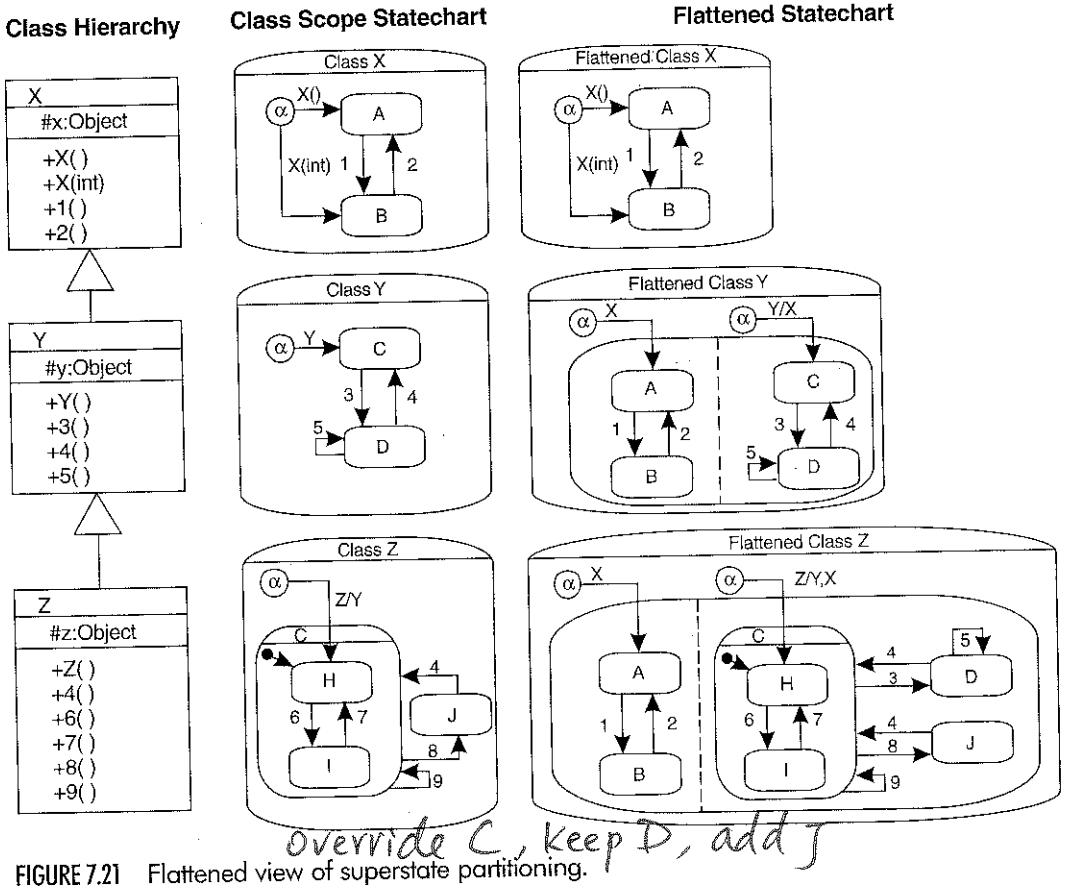
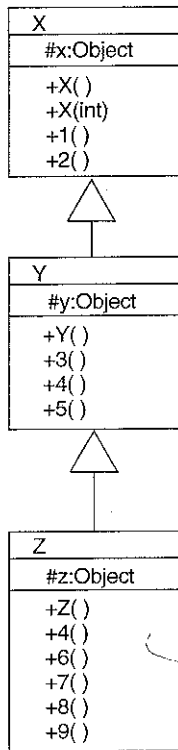


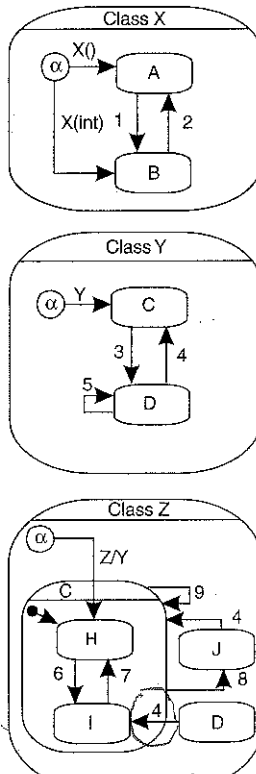
FIGURE 7.21 Flattened view of superstate partitioning.

flattened model must show the chain of events that happens upon construction of a subclass. Each class in the hierarchy has an alpha state. The transition from the subclass's alpha state to its initial state has the subclass constructor as its event. The typical percolation of constructors is represented by the action of sending the constructor message to its immediate superclass, for example, super new. When multiple alpha states appear, the first transition to fire is the one whose event must arrive from outside the scope of the statechart. This transition should involve the constructor message for the lowest-level subclass. The action on this transition becomes the event on the alpha transition for the next class up, and so on, until the root-level class is reached.

## Class Hierarchy



## Class Scope Statechart



## Flattened Statechart

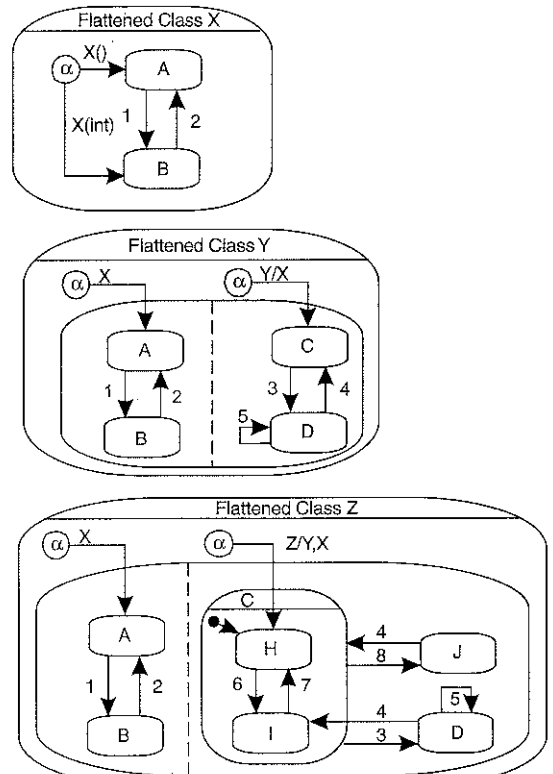


FIGURE 7.22 Flattened view of transition retargeting.

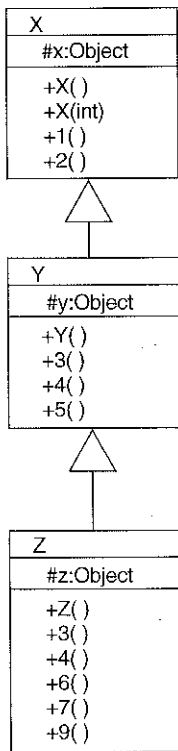
*not inside*  
*overriding method*

## Expanding the Statechart

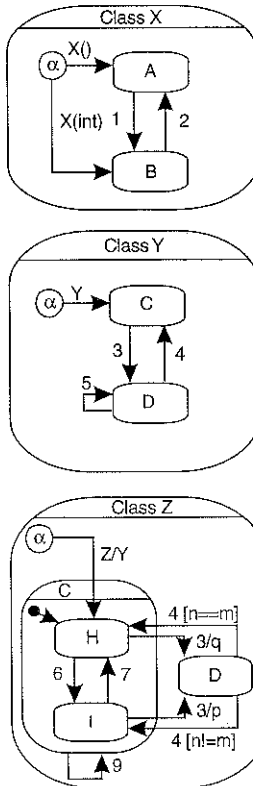
A statechart is an effective graphic technique to maintain intellectual control over a complex system. Although a statechart corrals the complexity beast, it cannot alter its basic nature. It is a model—not magic. Complex behavior cannot be reduced or eliminated by representing it as a statechart. Instead, we must pay for the reduction in graphic complexity by relying on the implicit semantics of the statechart. This may obscure subtle errors.

For example, suppose that the implementation incorrectly enters the last active substate upon accepting a transition that, on the statechart, enters a superstate. The implicit specification is that the initial state should obtain after accepting this transition. Testing the transition without checking the resultant

## Class Hierarchy



## Class Scope Statechart



## Flattened Statechart

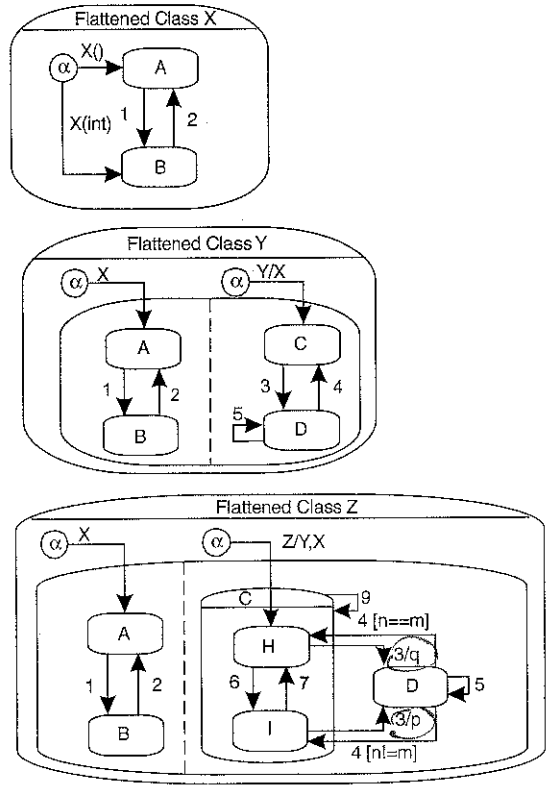


FIGURE 7.23 Flattened view of transition splitting.

state will not reveal this bug. It is possible to exercise the transition and still miss the bug if you conclude that the transition “works” after the event is accepted when the initial state just happens to be the current state. Testing from an expanded model will reveal these kinds of bugs, just as it will reveal garden-variety bugs.

Even if we manage to avoid bugs due to misinterpretation, we must expand the statechart’s implicit state machine to generate a complete test suite. Implicit behavior is no less a requirement than explicit behavior. If we test only the transitions and states that have an explicit arrow or box in the statechart, we will not exercise any of the implied behavior. Clearly, such a test suite could not be considered adequate.

row for each truth value combination of its guard and an additional column for each predicate within its guard. The truth table of the guard expression is enumerated in these predicate columns. In contrast, unguarded events do not have any predicate columns. Each state has a column, including alpha and omega. The cell located at the intersection of an event/guard row and state column designates whether the event is legal. A check mark indicates an explicitly specified transition. If necessary, the check mark can be replaced with a pointer to the desired action or other useful notation.

If an event has guards, then there is a row for each unique event/guard combination. If an event is guarded in some transitions but not in others, then include an additional don't care row.

Cells that do not correspond to explicit legal transitions represent illegal transitions. Each illegal cell holds a number that designates the response to be produced when an illegal event occurs. Table 7.3 lists several possible responses. Allen describes a similar, more elaborate approach to illegal events [Allen+95]. Cells with an "X" are mutually exclusive. For example, if an event is simply not accepted in a state, then any guard rows that apply to other states are excluded. The "X" is also applied to guard cells that are ignored with an unguarded event.

The response matrix corresponds to the flattened view of the class under test. Attempting to produce a response matrix without a flattened model may be difficult and confusing.

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 <code>IllegalEventException</code>
5	Mute	Disable the source of the event and ignore
6	Abend	Invoke abnormal termination services (e.g., core dump) and halt the process

Exception

Error



## 7.4 State-based Test Design

### 7.4.1 How State Machines Fail

It is not enough to have correct input/output mapping for a particular input/output pair when sequential control requirements exist as well. An implementation must also be faithful to the constraints of the specified state machine. For example, if a Traffic Light is *Green*, switching to *Red* is incorrect. Even if the red light becomes lit in response to the *RedOn* message, the system is buggy. That is, the light must always switch to *Yellow* before *Red* in a correct system. Understanding how an implementation can fail to observe these constraints provides a focus for test design and a benchmark against which to evaluate test design strategies.

#### Control Faults

A control fault allows an incorrect sequence of events to be accepted or produces an incorrect sequence of output actions. Although an infinite number of faulty implementations exist for any given specification, there are relatively few general kinds of control faults.<sup>26</sup> Compared with the specification, a buggy implementation will have one or more incorrect responses:

- A missing or incorrect transition (the resultant state is incorrect, but not corrupt)
- A missing or incorrect event (a valid message is ignored)
- A missing or incorrect action (the wrong thing happens as a result of a transition)
- An extra, missing, or corrupt state (behavior becomes unpredictable)
- A sneak path (a message is accepted when it should not be)
- An illegal message failure (an unexpected message causes a failure)
- A trap door (the implementation accepts undefined messages)

These faults may occur individually or in any nightmare combination. Table 7.4 shows relationships among these faults. Not all combinations are possible.

26. There are  $sa^{es}$  of possible implementations for a machine with  $s$  states,  $e$  events, and  $a$  actions [Sidhu+89]. For example, with only 5 states, 2 events, and 2 actions, there are 10 billion possible implementations. Fortunately, compact testing strategies are very effective at revealing incorrect implementations.

TABLE 7.4 State Control Faults

Event	Action	Resultant State	Error Description
OK	OK	OK Wrong Corrupt	Normal transition Incorrect state Corrupted state
	Wrong/ undefined	OK Wrong Corrupt	Incorrect action Incorrect action, wrong state Incorrect action, corrupt state
	Missing	OK Wrong Corrupt	Missing action Missing action, incorrect state Missing action, corrupt state
Reject Legal	DC	Same Defined Corrupt	Missing transition, no side effect Missing transition, side effect Missing transition, corrupt state
Accept Illegal	Defined for this machine	Same Defined Corrupt	Sneak path, no side effect Sneak path with side effect Sneak path to corrupt state
	Undefined	Same	Sneak path, no side effect, incorrect output
		Defined Corrupt	Sneak path with side effect, incorrect output Sneak path to corrupt state, incorrect output
Accept Undefined	Defined for this machine	Same Defined Corrupt	Trap door to action Trap door to action, side effect Trap door to action, corrupt state
	Undefined	Same Defined	Trap door with incorrect output Trap door with incorrect output and side effect
		Corrupt	Trap door with incorrect output to corrupt state

For example, omitting an event means that no corresponding incorrect action will exist. Figures 7.27 through 7.34 illustrate several of these faults.

- *Missing transition.* The implementation does not respond to a valid event/state pair. For example, player 2 loses the volley, but continues as server (Figure 7.27).
- *Incorrect transition.* The implementation behaves as if an incorrect resultant state has been reached. For example, after player 2 misses, the game resets (Figure 7.28).

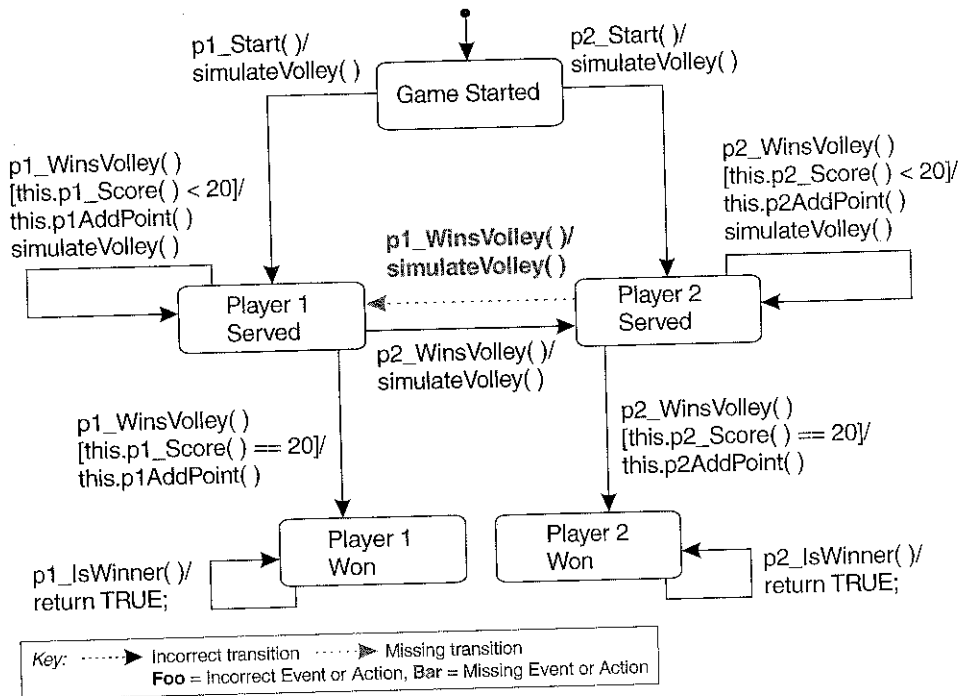


FIGURE 7.27 Missing transition.

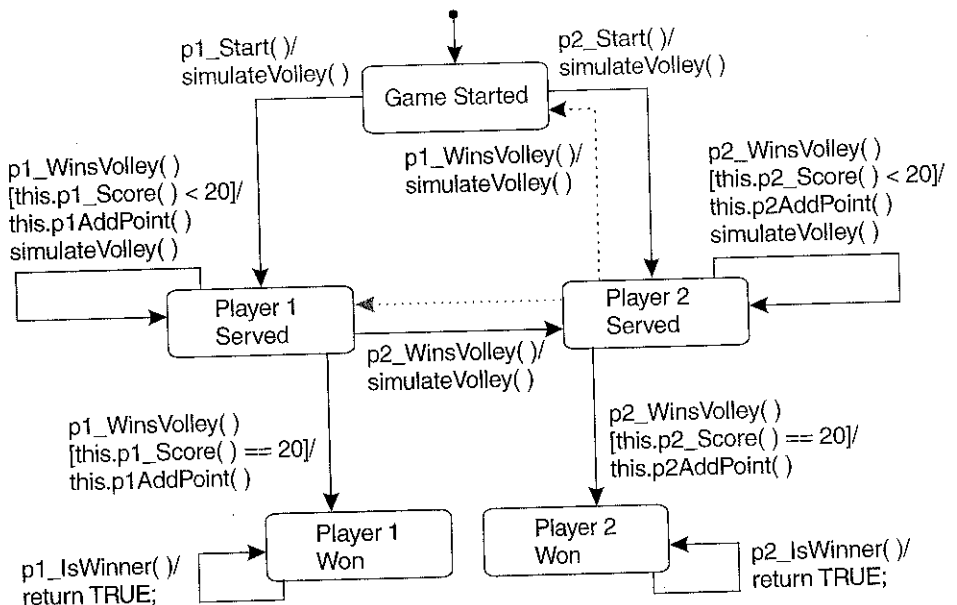


FIGURE 7.28 Incorrect transition/resultant state.

- *Missing action.* The implementation does not produce any action for a transition. For example, suppose the output action after player 1 starts is omitted. No volley is generated and the system will wait forever (Figure 7.29).
- *Incorrect action.* The implementation produces the wrong action for a transition (this case is different from incorrect output from the action). For example, after player 2 misses, player 2's score is incremented (Figure 7.30).
- *Sneak path.* The implementation accepts an event that is illegal or unspecified for a state. For example, when player 2 is serving, player 2 can win if his or her start button is pressed (Figure 7.31).
- *Corrupt state.* The implementation computes a state that is not valid. Either the class or state invariant is violated. This state can result from a variety of coding and design errors or bugs in the virtual machine. For example, if player 1 is serving at game point and player 2 misses, the game crashes and can't be restarted (Figure 7.32, page 234).
- *Illegal message failure.* The implementation fails to handle an illegal message correctly. Incorrect output is produced, the state is corrupted, or both.

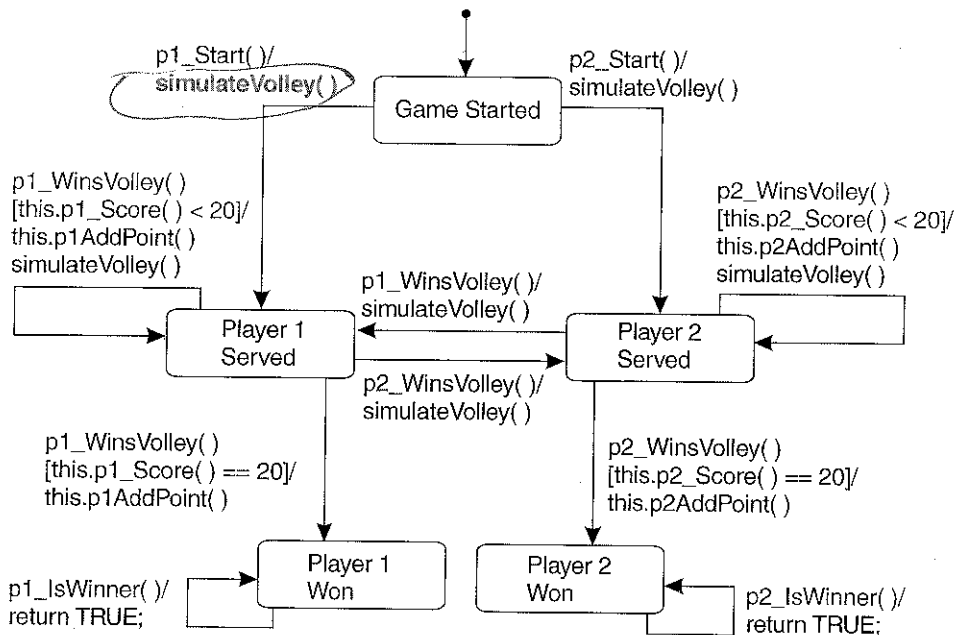


FIGURE 7.29 Missing action.

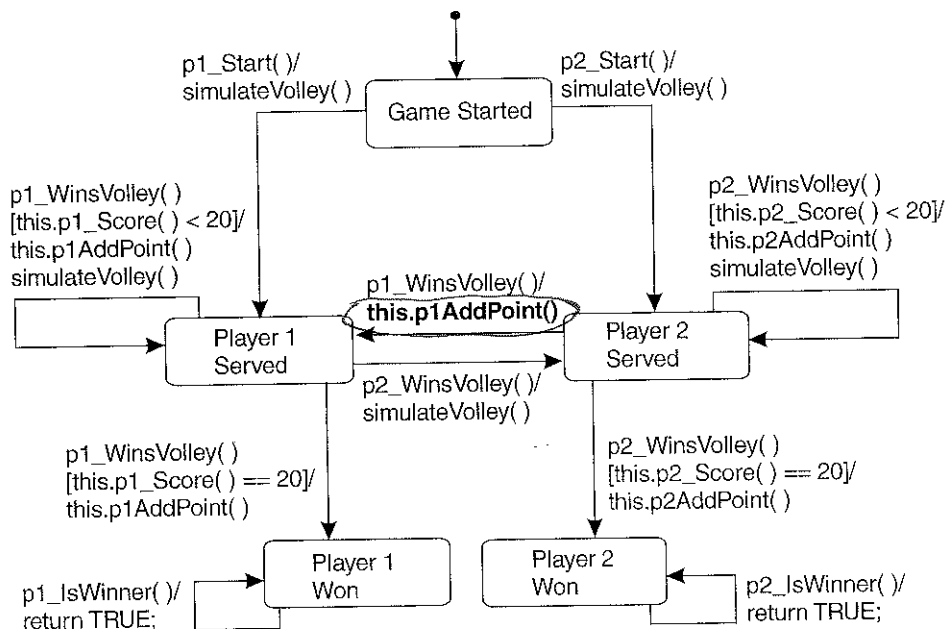


FIGURE 7.30 Wrong action.

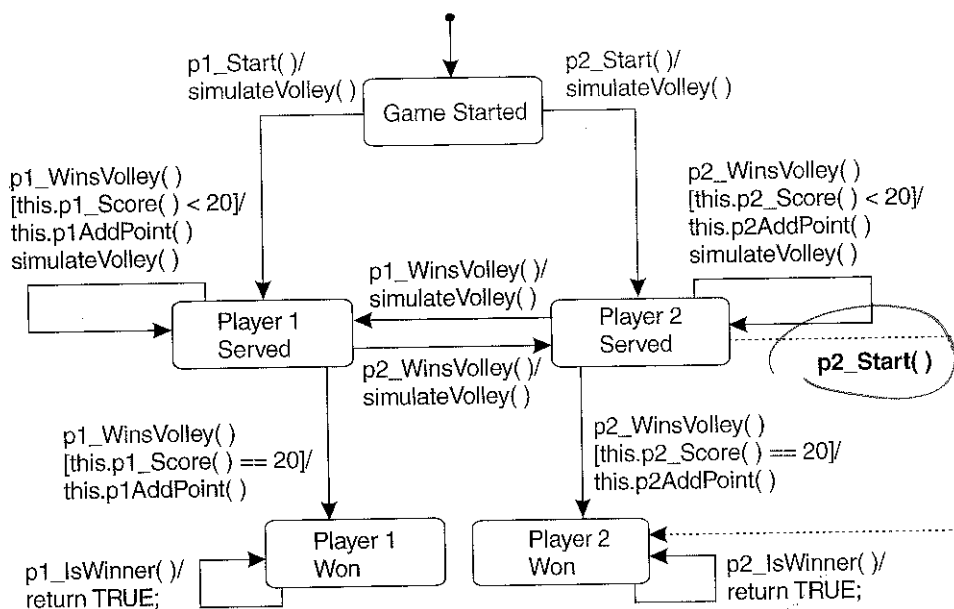


FIGURE 7.31 Sneak path.

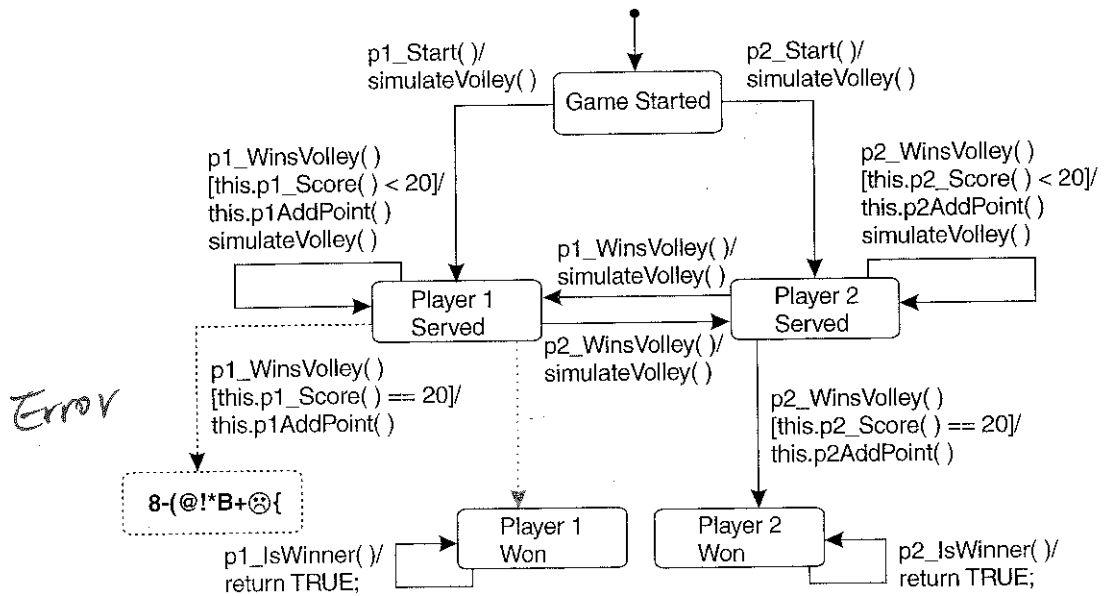


FIGURE 7.32 Transition to corrupt state.

For example, if player 2 presses the Player Select button after serving, the game crashes and can't be restarted (Figure 7.33).

- *Trap door.* The implementation accepts an event that is not defined in the specification. For example, when player 1 is serving, player 1 can win any time by pressing the Scroll Lock key (Figure 7.34). A trap door can result from: (1) obsolete features that were not removed when a class was revised, (2) inherited features that are inconsistent with the requirements of a subclass, (3) “undocumented” features added by the developer for debugging purposes, or (4) sabotage for criminal or malicious purposes.

### Incorrect Composite Behavior

Misuse of inheritance with modal classes can lead to state control bugs. Unless care is taken during design and implementation, a subclass can easily conflict with the sequential requirements for one or several of its superclasses. Bugs that result from the interaction of superclasses and subclasses are not likely to be

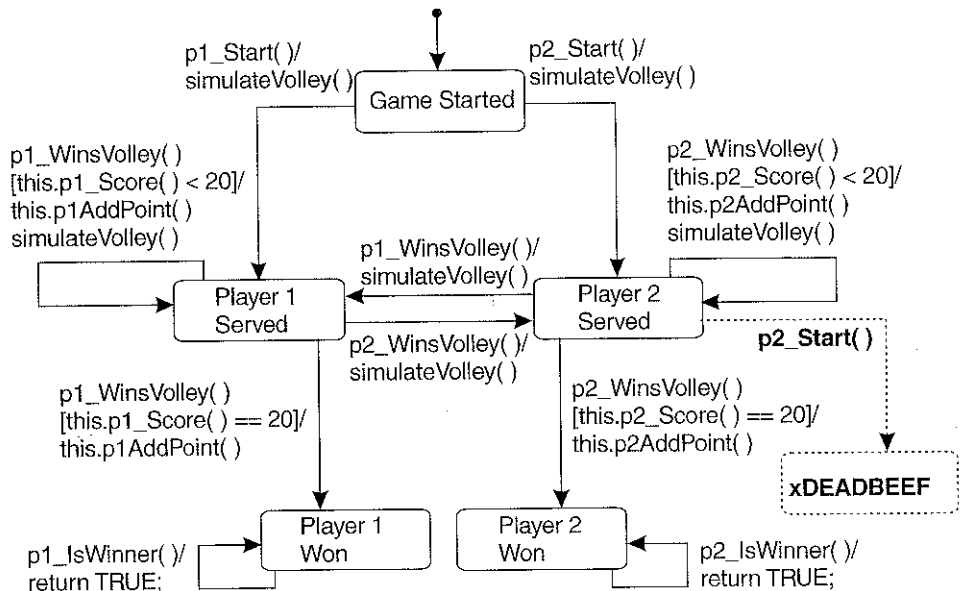


FIGURE 7.33 Sneak path to corrupt state.

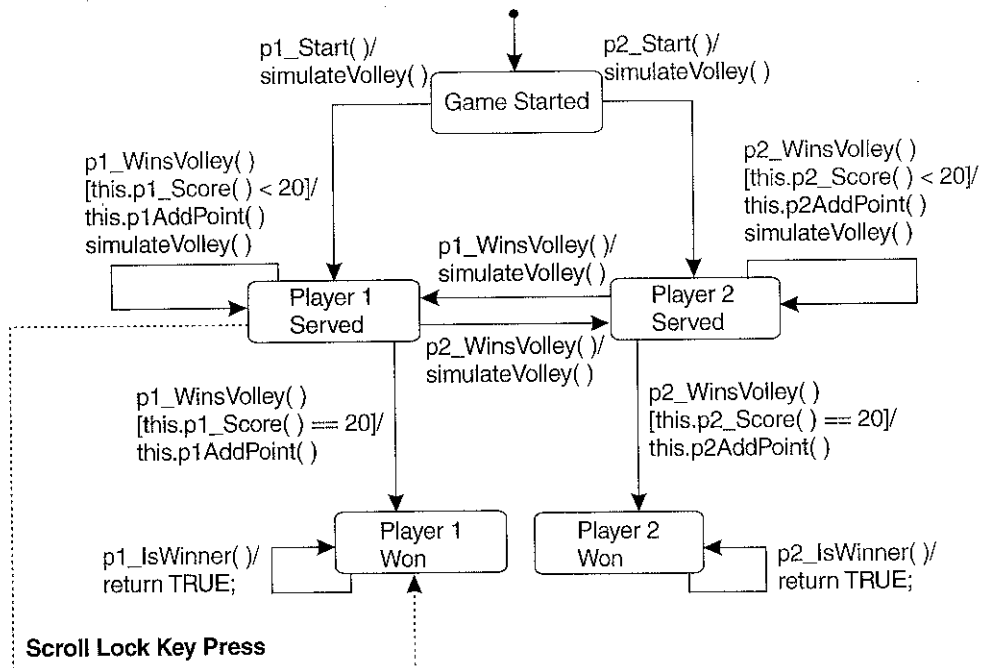


FIGURE 7.34 Trap door/extra transition.

TABLE 7.5 State Model Checklist: Structure

**1. The Model Is Structurally Complete and Consistent.**

- 1.1 One state is designated as the initial state and has only outbound transitions (this may be the  $\alpha$  state).
- 1.2 At least one state is designated as the final state and has only inbound transitions (this may be the  $\omega$  state). If not, assumptions about termination should be made explicit.
- 1.3 There are no equivalent states. Algorithms for detecting equivalent states are given in [Hopcroft+79] and [Beizer 90].
- 1.4 Every state is reachable from the initial state.
- 1.5 The final state is reachable from all other states.
- 1.6 Every state, excluding the initial state, is reachable from every other state. That is, there are no "one-shot" states. Algorithms to check reachability are provided in [Hopcroft+79] and [Marick 95]. Nonreachable states can also be found with the round-trip path algorithm (see the following).
- 1.7 Every defined event and every defined action appears in at least one transition.
- 1.8 Except for the initial and final states, every state has at least one incoming transition and at least one outgoing transition.
- 1.9 The events accepted in a particular state are unique or differentiated by mutually exclusive guard conditions (the same event may not appear on more than one transition for any given state).
- 1.10 The state machine is completely specified. Every event/state pair has at least one transition specifying that the event is (1) explicitly accepted, producing an explicit action; (2) explicitly rejected and handled by an explicit exception mechanism, resulting in a defined state; or (3) implicitly rejected and handled by an explicit exception mechanism, resulting in a defined state.

- The *robustness checklist* (Table 7.9, page 241) suggests capabilities that are necessary for safe and correct behavior under failure modes and degraded operating conditions.

I strongly recommend proactive use of these checklists. Use them to inspect and review OOAD behavior models, implementations, and any state models that you develop for test design. A state model that meets the conditions in the checklist will encourage early prevention of errors and ease testing. Cook and Daniels note, "we also require that all sequences of messages accepted by the super-type will be accepted by the sub-type and leave it in a corresponding




### 7.4.3 The N+ Test Strategy

The N+ strategy integrates elements of advanced state-based testing, UML state models, and testing considerations unique to object-oriented implementations. It builds on generic state-based testing strategies in the following ways:

- It uses a flattened state model.
- All implicit transitions are exercised to reveal sneak paths.
- The implementation must have a trusted ability to report the resultant state.

An N+ test suite that is passed achieves N+ coverage. Achieving N+ coverage will reveal all state control faults, all sneak paths, and many corrupt state bugs. Because it exercises the IUT at flattened scope, it will reveal subcontracting bugs and superclass/subclass integration bugs. If more than one alpha transition is present, this type of testing will reveal faults on each alpha transition. It will reveal faults in all omega transitions. It can suggest the presence of a trap door if used with a code coverage analyzer.

The steps to generate an N+ test suite are as follows:

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

The FREE model is not an absolute necessity. One can begin at step 2 with any other testable model of behavior that provides the same information. This procedure may be applied to develop a test suite for any implementation of sequential behavior.

The N+ test strategy is demonstrably more powerful than simpler state-based strategies. The simpler strategies require less analysis, however, and will produce smaller test suites. The cost/benefit trade-offs of this approach are considered at the end of the chapter.

### The ThreePlayerGame Example

ThreePlayerGame is a subclass of TwoPlayerGame. It reflects the rules for cut throat, a three-player version of racquetball. A player may score only when serving. The server volleys against two other players, in no particular order. The two nonserving players compete for the opportunity to return the ball, in an effort to win the volley and become the server. The rest of the two-player rules apply. A Java fragment for this game that subclasses TwoPlayerGame follows (the superclass appears on page 187):

```
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 */ }
    private void   p3_AddPoint( )      { /* Add 1 to P3's score */ }
    public boolean p3_IsWinner( )      { /* True if P3's is 21 */ }
    public boolean p3_IsServer( )      { /* True if P3 is server */ }
    public int     p3_Score( )         { /* Returns P3's score */ }
}
```

Our first order of business is to develop a FREE model for ThreePlayerGame. Figure 7.35 shows the class hierarchy and corresponding statecharts.<sup>27</sup> Alpha and omega states are added to the TwoPlayerGame statechart. The class scope model of ThreePlayerGame simply inherits all of the behavior of TwoPlayerGame and then adds symmetric transitions for the new behavior. The statechart is shown in the lower part of Figure 7.36. Expansion of the statechart yields the flattened transition diagram shown in Figure 7.36. Figure 7.37 on page 246 gives the corresponding response matrix. The codes in the response matrix correspond to the actions listed in Table 7.3. The conformance and sneak path test suites appear in Tables 7.10 (pages 251–252) and 7.11 (pages 254–255). Next, we explain how they were developed.

### Generate the Round-trip Path Tree

With a testable state model in hand, the next step is to transcribe the **transition tree** from the transition diagram.<sup>28</sup> This tree includes all transition

27. I've used a C++ destructor to show how an omega transition is modeled. This would not be part of a Java implementation.

28. This procedure is adapted from Chow's *W-method* [Chow 78]. The W-method requires (1) a complete state machine model (all states have events and actions), (2) a minimal state model (there are no redundant or unnecessary states), (3) an initial state, and (4) no unreachable states. A verified FREE model meets these conditions.

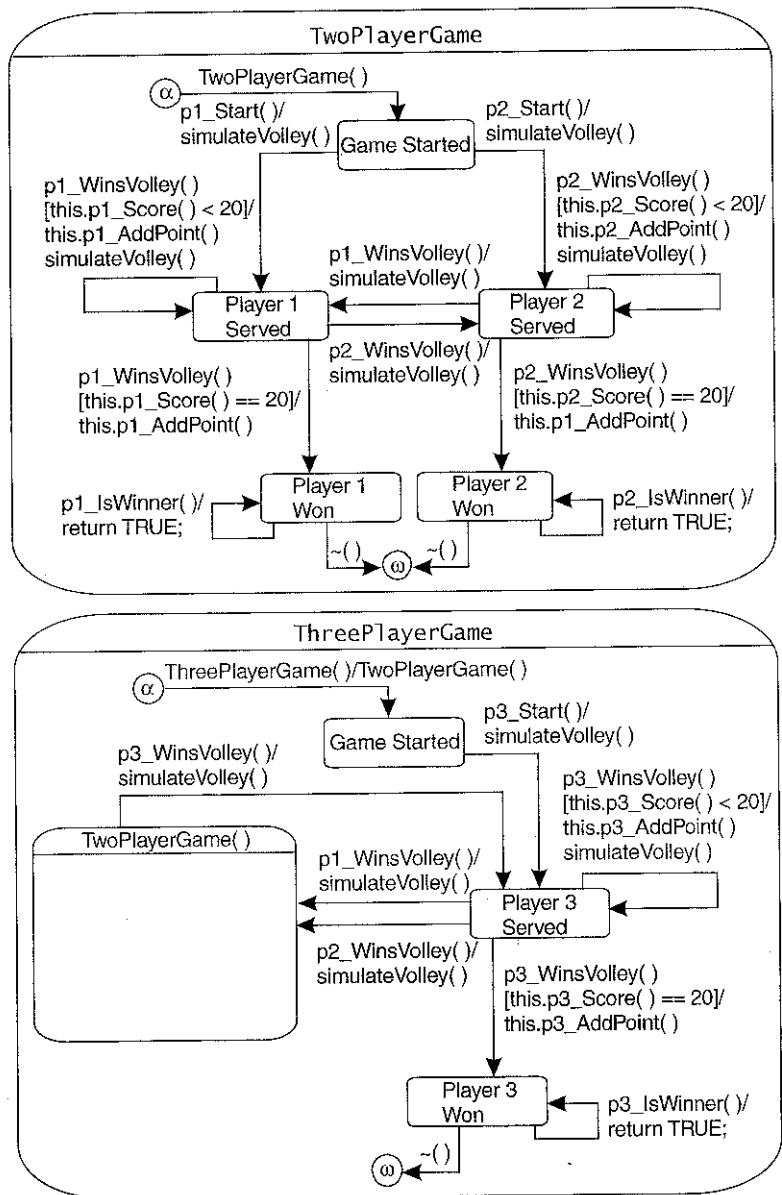
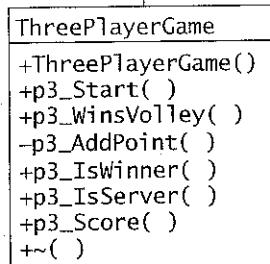
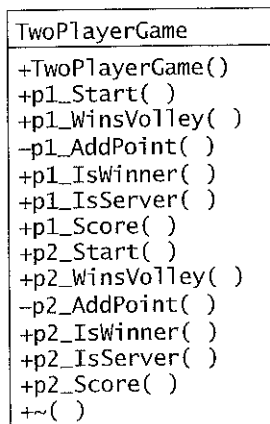


FIGURE 7.35 ThreePlayerGame class hierarchy and statecharts at class scope.

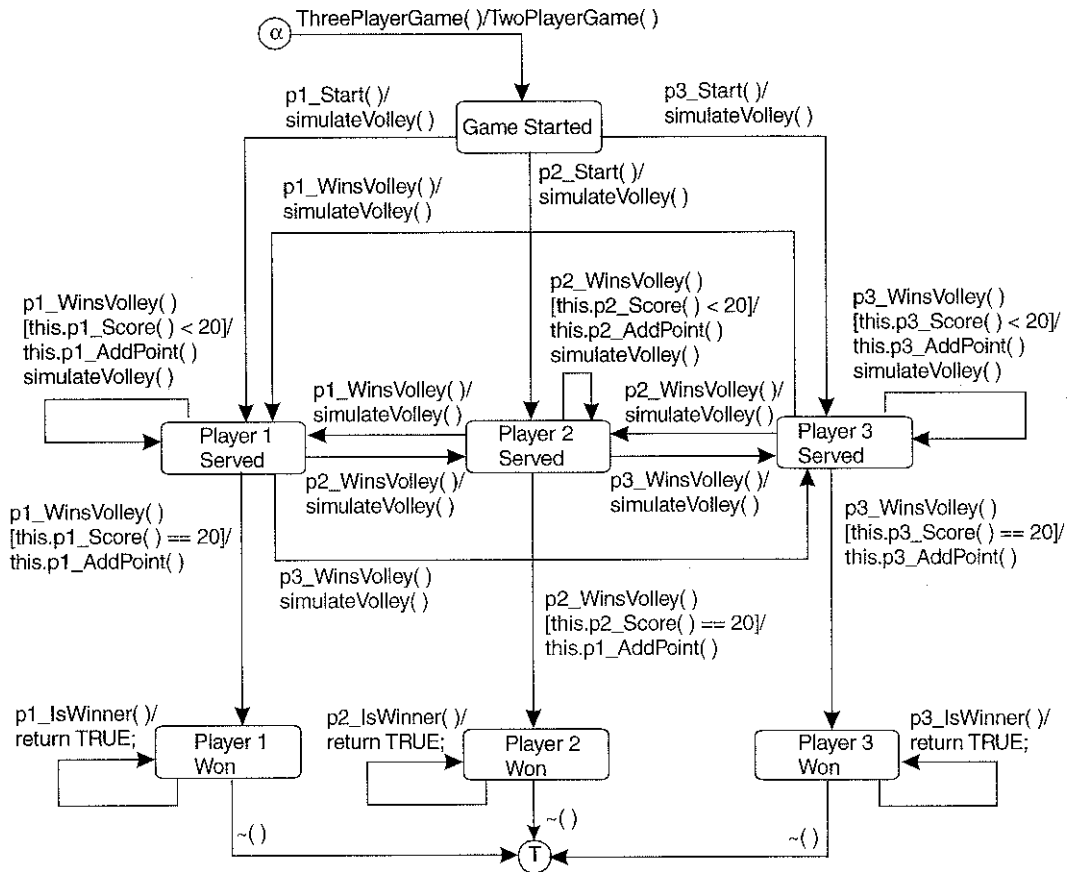


FIGURE 7.36 Flattened transition diagram, ThreePlayerGame.

sequences that begin and end with the same state, i.e., all round-trip paths. The procedure does not attempt to generate all possible alpha-omega paths. However, a round-trip test suite will exercise all transitions and loops on every possible alpha-omega path at least once.

The initial state is transcribed as the root node of the tree. This is an  $\alpha$  state if one has been specified. An edge is drawn for every transition out of the initial node, with nodes being added for resultant states. A leaf node (one with no exiting transitions) is marked as *terminal* if the state it represents has already been drawn, is a final state, or is the  $\omega$  state. No more transitions are

Events and Guards		Accepting State/Expected Response							
		Game Started	Player 1 Served	Player 2 Served	Player 3 Served	Player 1 Won	Player 2 Won	Player 3 Won	W
ctor		✓	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								
	DC	4	✓	✓	✓	4	4	4	6
	F		6						
	T		✓						
	F		✓						
	T		✓						
	T		T						
p2_WinsVolley	p2_score < 20								
	DC	4	✓	6	✓	4	4	4	6
	F								
	F			✓					
	T			✓					
	T		T						
p3_WinsVolley	p3_score < 20								
	DC	4	✓	✓	6	4	4	4	6
	F								
	F								
	T								
	T		T						
p1_isWinner			✓	✓	✓	✓	✓	✓	6
p2_isWinner		✓	✓	✓	✓	✓	✓	✓	6
p3_isWinner		✓	✓	✓	✓	✓	✓	✓	6
Other Public Accessors		✓	✓	✓	✓	✓	✓	✓	6
ctor		✓	✓	✓	✓	✓	✓	✓	6

FIGURE 7.37 Response matrix for ThreePlayerGame (for Action codes, see Table 7.3).

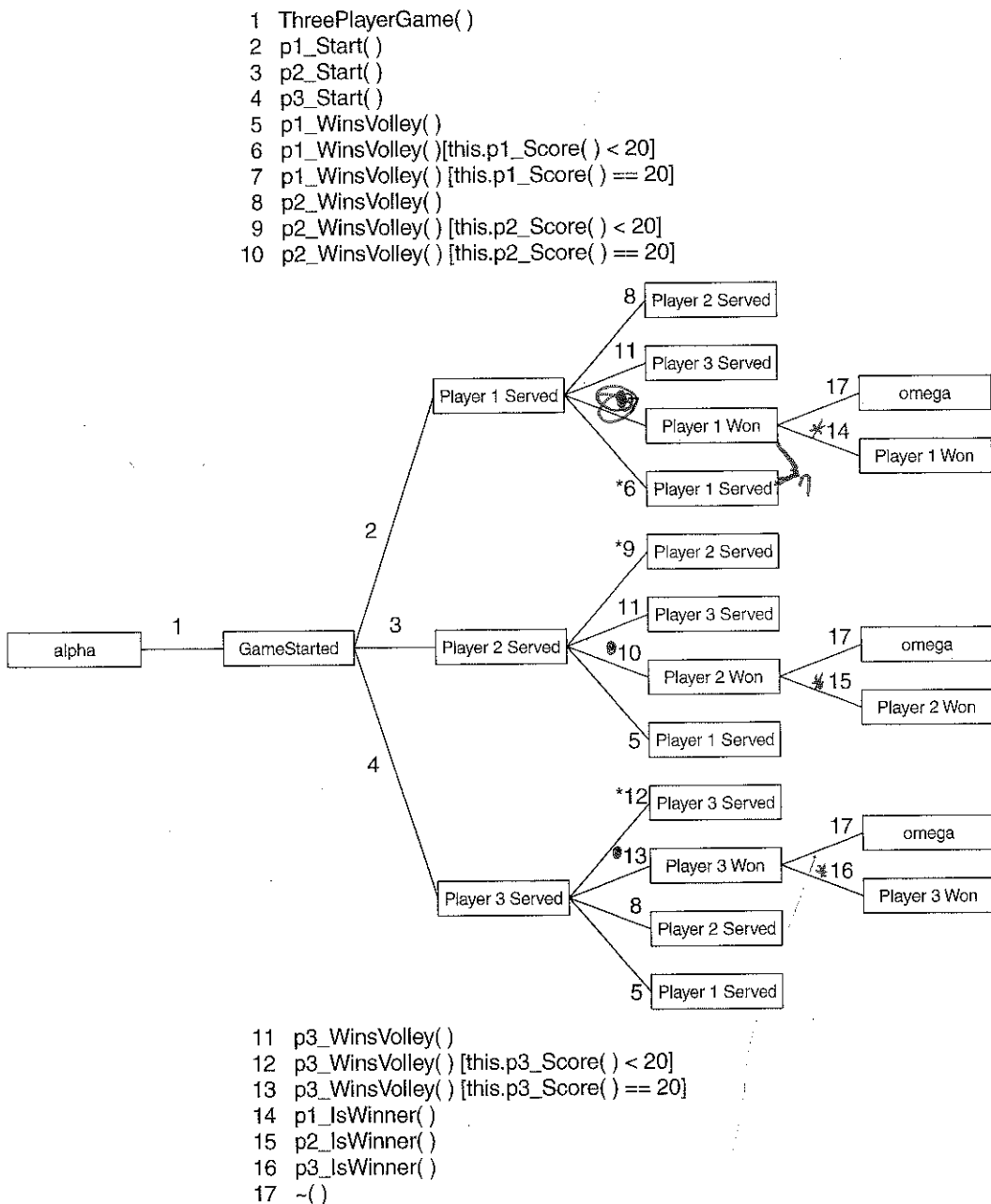


FIGURE 7.38 Transition tree for ThreePlayerGame.

TABLE 7.10 Conformance Test Suite for ThreePlayerGame

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

continued

TABLE 7.10 (continued)

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



sequences that will reach all states. Any conformance suite test sequence whose final resultant state is the same as the sneak path state can be used. Since the conformance sequences have been debugged and the IUT has passed them, they are trustworthy and a quick way to place the IUT into any desired state. Reusing conformance suite sequences has some minor disadvantages, however. The sneak path suite depends on it, so changes in the conformance suite may require changes in the sneak path test suite. In addition, repeating passing sequences means that you give up any chance of revealing additional bugs with these setup sequences.

2. Apply the illegal event by sending a message or forcing the virtual machine to generate the desired event.
3. Check that the actual response matches the specified response.
4. Check that the resultant state is unchanged. Sometimes, a new concrete state may be acceptable.

If the response and resultant state are as expected, the test passes. These steps are repeated for each possible sneak path. Table 7.11 shows the sneak path test suite for the ThreePlayerGame.

TABLE 7.11 Sneak Path Test Suite for ThreePlayerGame

TCID	Test Case			Expected Result	
	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

TABLE 7.11 (continued)

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

### Event Path Sensitization

If a test sequence exercises one or more guards, then we must find input values that will satisfy all of these guards. If the output of an event in the sequence is used in a subsequent guard, the analysis can become complex. Values (or states) necessary for conditional expression must be determined by analysis. We can use the same approach taken to identify inputs that satisfy the source code entry–exit path conditions. See Section 10.2.3, Path Sensitization.