Note that we have put X for the discount for two of the columns (Rules 1 and 2) – this means that this combination should not occur. You cannot be both a new customer and already hold a loyalty card! There should be an error message stating this, but even if we don't know what that message should be, it will still make a good test.

We have made an assumption in Rule 3. Since the coupon has a greater discount than the new customer discount, we assume that the customer will choose 20% rather than 15%. We cannot add them, since the coupon cannot be used with the 'new customer' discount. The 20% action is an assumption on our part, and we should check that this assumption (and any other assumptions that we make) is correct, by asking the person who wrote the specification or the users.

For Rule 5, however, we can add the discounts, since both the coupon and the loyalty card discount should apply (at least that's our assumption).

Rules 4, 6 and 7 have only one type of discount and Rule 8 has no discount, so 0%.

If we are applying this technique thoroughly, we would have one test for each column or rule of our decision table. The advantage of doing this is that we may test a combination of things that otherwise we might not have tested and that could find a defect.

However, if we have a lot of combinations, it may not be possible or sensible to test every combination. If we are time-constrained, we may not have time to test all combinations. Don't just assume that all combinations need to be tested; it is better to prioritize and test the most important combinations. Having the full table enables us to see which combinations we decided to test and which not to test this time.

There may also be many different actions as a result of the combinations of conditions. In the example above we just had one: the discount to be applied. The decision table shows which actions apply to each combination of conditions.

In the example above all the conditions are binary, i.e. they have only two possible values: True or False (or, if you prefer Yes or No). Often it is the case that conditions are more complex, having potentially many possible values. Where this is the case the number of combinations is likely to be very large, so the combinations may only be sampled rather than exercising all of them.

State transition testing A black-box test design technique in which test cases are designed to execute valid and invalid state transitions.

State diagram a diagram that depicts the states that a component or system can assume and shows the events or circumstances that cause and/or result from a change from one state to another.

4.3.3 State transition testing

State transition testing is used where some aspect of the system can be described in what is called a 'finite state machine'. This simply means that the system can be in a (finite) number of different states, and the transitions from one state to another are determined by the rules of the 'machine'. This is the model on which the system and the tests are based. Any system where you get a different output for the same input, depending on what has happened before, is a finite state system. A finite state system is often shown as a **state diagram** (see Figure 4.2).

For example, if you request to withdraw $100 from a bank ATM, you may be given cash. Later you may make exactly the same request but be refused the money (because your balance is insufficient). This later refusal is because the state of your bank account has changed from having sufficient funds to cover the withdrawal to having insufficient funds. The transaction that caused your account to change its state was probably the earlier withdrawal. A state diagram can represent a model from the point of view of the system, the account or the customer.

Another example is a word processor. If a document is open, you are able to close it. If no document is open, then 'Close' is not available. After you choose 'Close' once, you cannot choose it again for the same document unless you open that document. A document thus has two states: open and closed.

A state transition model has four basic parts:

- the states that the software may occupy (open/closed or funded/insufficient funds);
- the transitions from one state to another (not all transitions are allowed);
- the events that cause a transition (closing a file or withdrawing money);
- the actions that result from a transition (an error message or being given your cash).

Note that in any given state, one event can cause only one action, but that the same event – from a different state – may cause a different action and a different end state.

We will look first at test cases that execute valid state transitions.

Figure 4.2 shows an example of entering a Personal Identity Number (PIN) to a bank account. The states are shown as circles, the transitions as lines with arrows and the events as the text near the transitions. (We have not shown the actions explicitly on this diagram, but they would be a message to the customer saying things such as 'Please enter your PIN'.)

The state diagram shows seven states but only four possible events (Card inserted, Enter PIN, PIN OK and PIN not OK). We have not specified all of the possible transitions here – there would also be a time-out from 'wait for PIN' and from the three tries which would go back to the start state after the time had elapsed and would probably eject the card. There would also be a transition from the 'eat card' state back to the start state. We have not specified all the possible events either – there would be a 'cancel' option from 'wait for PIN'

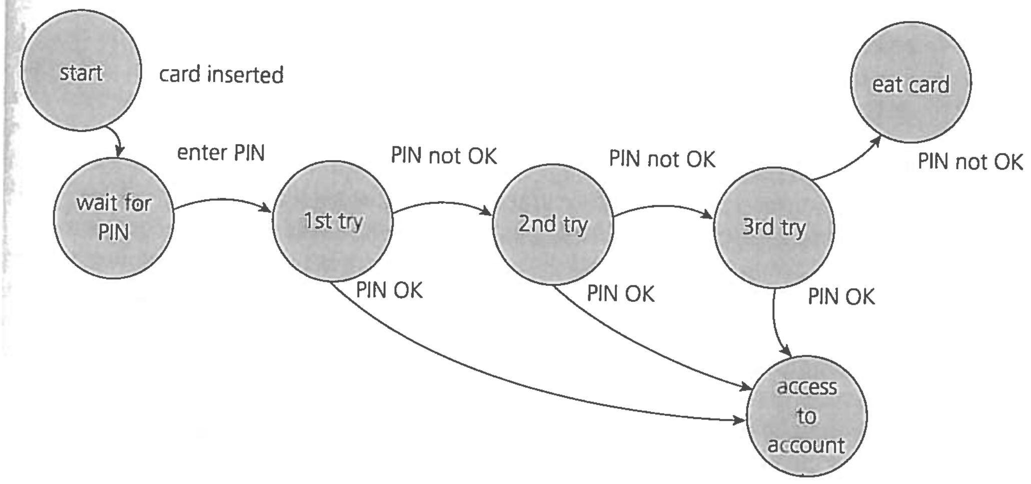

FIGURE 4.2 State diagram for PIN entry

and from the three tries, which would also go back to the start state and eject the card. The 'access account' state would be the beginning of another state diagram showing the valid transactions that could now be performed on the account.

However this state diagram, even though it is incomplete, still gives us information on which to design some useful tests and to explain the state transition technique.

In deriving test cases, we may start with a typical scenario. A sensible first test case here would be the normal situation, where the correct PIN is entered the first time. To be more thorough, we may want to make sure that we cover every state (i.e. at least one test goes through each state) or we may want to cover every transition. A second test (to visit every state) would be to enter an incorrect PIN each time, so that the system eats the card. We still haven't tested every transition yet. In order to do that, we would want a test where the PIN was incorrect the first time but OK the second time, and another test where the PIN was correct on the third try. These tests are probably less important than the first two.

Note that a transition does not need to change to a different state (although all of the transitions shown above do go to a different state). So there could be a transition from 'access account' which just goes back to 'access account' for an action such as 'request balance'.

Test conditions can be derived from the state graph in various ways. Each state can be noted as a test condition, as can each transition. In the Syllabus, we need to be able to identify the coverage of a set of tests in terms of transitions.

Going beyond the level expected in the Syllabus, we can also consider transition pairs and triples and so on. Coverage of all individual transitions is also known as 0-switch coverage, coverage of transition pairs is 1-switch coverage, coverage of transition triples is 2-switch coverage, etc. Deriving test cases from the state transition model is a black-box approach. Measuring how much you have tested (covered) is getting close to a white-box perspective. However, state transition testing is regarded as a black-box technique.

One of the advantages of the state transition technique is that the model can be as detailed or as abstract as you need it to be. Where a part of the system is more important (that is, requires more testing) a greater depth of detail can be modeled. Where the system is less important (requires less testing), the model can use a single state to signify what would otherwise be a series of different states.

Testing for invalid transitions

Deriving tests only from a state graph (also known as a state chart) is very good for seeing the valid transitions, but we may not easily see the negative tests, where we try to generate invalid transitions. In order to see the total number of combinations of states and transitions, both valid and invalid, a **state table** is useful.

State table A grid showing the resulting transitions for each state combined with each possible event, showing both valid and invalid transitions.

The state table lists all the states down one side of the table and all the events that cause transitions along the top (or vice versa). Each cell then represents a state–event pair. The content of each cell indicates which state the system will move to, when the corresponding event occurs while in the associated state. This will include possible erroneous events – events that are not expected to happen in certain states. These are negative test conditions.

TABLE 4.9 State table for the PIN example

	Insert card	Valid PIN	Invalid PIN
S1) Start state	S2	–	–
S2) Wait for PIN	–	S6	S3
S3) 1st try invalid	–	S6	S4
S4) 2nd try invalid	–	S6	S5
S5) 3rd try invalid	–	–	S7
S6) Access account	–	?	?
S7) Eat card	S1 (for new card)	–	–

Table 4.9 lists the states in the first column and the possible inputs across the top row. So, for example, if the system is in State 1, inserting a card will take it to State 2. If we are in State 2, and a valid PIN is entered, we go to State 6 to access the account. In State 2 if we enter an invalid PIN, we go to State 3. We have put a dash in the cells that should be impossible, i.e. they represent invalid transitions from that state.

We have put a question mark for two cells, where we enter either a valid or invalid PIN when we are accessing the account. Perhaps the system will take our PIN number as the amount of cash to withdraw? It might be a good test! Most of the other invalid cells would be physically impossible in this example. Invalid (negative) tests will attempt to generate invalid transitions, transitions that shouldn't be possible (but often make good tests when it turns out they are possible).

A more extensive description of state machines is found in [Marick, 1994]. State transition testing is also described in [Craig, 2002], [Copeland, 2003], [Beizer, 1990] and [Broekman, 2003]. State transition testing is described in BS7925-2, including designing tests and coverage measures.

4.3.4 Use case testing

Use case testing is a technique that helps us identify test cases that exercise the vhole system on a transaction by transaction basis from start to finish. They are lescribed by Ivar Jacobson in his book *Object-Oriented Software Engineering: A Jse Case Driven Approach* [Jacobson, 1992].

Use case testing A black-box test design technique in which test cases are designed to execute user scenarios.

A use case is a description of a particular use of the system by an actor (a iser of the system). Each use case describes the interactions the actor has with he system in order to achieve a specific task (or, at least, produce something of 'alue to the user). Actors are generally people but they may also be other ystems. Use cases are a sequence of steps that describe the interactions ietween the actor and the system.

Use cases are defined in terms of the actor, not the system, describing what he actor does and what the actor sees rather than what inputs the system xpects and what the system outputs. They often use the language and terms of ie business rather than technical terms. especially when the actor is a business user. They serve as the foundation for developing test cases mostly at the system and acceptance testing levels.

Use cases can uncover integration defects, that is, defects caused by the incorrect interaction between different components. Used in this way, the actor may be something that the system interfaces to such as a communication link or sub-system.

Use cases describe the process flows through a system based on its most likely use. This makes the test cases derived from use cases particularly good for finding defects in the real-world use of the system (i.e. the defects that the users are most likely to come across when first using the system). Each use case usually has a mainstream (or most likely) scenario and sometimes additional alternative branches (covering, for example, special cases or exceptional conditions). Each use case must specify any preconditions that need to be met for the use case to work. Use cases must also specify postconditions that are observable results and a description of the final state of the system after the use case has been executed successfully.

The PIN example that we used for state transition testing could also be defined in terms of use cases, as shown in Figure 4.3. We show a success scenario and the extensions (which represent the ways in which the scenario could fail to be a success).

For use case testing, we would have a test of the success scenario and one test for each extension. In this example, we may give extension 4b a higher priority than 4a from a security point of view.

System requirements can also be specified as a set of use cases. This approach can make it easier to involve the users in the requirements gathering and definition process.

	Step	**Description**
Main Success Scenario **A: Actor** **S: System**	1	A: Inserts card
	2	S: Validates card and asks for PIN
	3	A: Enters PIN
	4	S: Validates PIN
	5	S: Allows access to account
Extensions	2a	Card not valid S: Display message and reject card
	4a	PIN not valid S: Display message and ask for re-try (twice)
	4b	PIN invalid 3 times S: Eat card and exit

FIGURE 4.3 Partial use case for PIN entry