of inputs result in different actions being taken, this can be more difficult to show using equivalence partitioning and boundary value analysis, which tend to be more focused on the user interface. The other two specification-based techniques, decision tables and state transition testing are more focused on business logic or business rules.

Decision table A table showing combinations of inputs and/or stimuli (causes) with their associated outputs and/or actions (effects), which can be used to design test cases.

Decision table testing A black-box test design technique in which test cases are designed to execute the combinations of inputs and/or stimuli (causes) shown in a decision table.

A **decision table** is a good way to deal with combinations of things (e.g. inputs). This technique is sometimes also referred to as a 'cause–effect' table. The reason for this is that there is an associated logic diagramming technique called 'cause–effect graphing' which was sometimes used to help derive the decision table (Myers describes this as a combinatorial logic network [Myers, 1979]). However, most people find it more useful just to use the table described in [Copeland, 2003].

If you begin using decision tables to explore what the business rules are that should be tested, you may find that the analysts and developers find the tables very helpful and want to begin using them too. Do encourage this, as it will make your job easier in the future. Decision tables provide a systematic way of stating complex business rules, which is useful for developers as well as for testers. Decision tables can be used in test design whether or not they are used in specifications, as they help testers explore the effects of combinations of different inputs and other software states that must correctly implement business rules. Helping the developers do a better job can also lead to better relationships with them.

Testing combinations can be a challenge, as the number of combinations can often be huge. Testing all combinations may be impractical if not impossible. We have to be satisfied with testing just a small subset of combinations but making the choice of which combinations to test and which to leave out is not trivial. If you do not have a systematic way of selecting combinations, an arbitrary subset will be used and this may well result in an ineffective test effort.

Decision tables aid the systematic selection of effective test cases and can have the beneficial side-effect of finding problems and ambiguities in the specification. It is a technique that works well in conjunction with equivalence partitioning. The combination of conditions explored may be combinations of equivalence partitions.

In addition to decision tables, there are other techniques that deal with testing combinations of things: pairwise testing and orthogonal arrays. These are described in [Copeland, 2003]. Another source of techniques is [Pol *et al.*, 2001]. Decision tables and cause-effect graphing are described in [BS7925-2], including designing tests and measuring coverage.

Using decision tables for test design

The first task is to identify a suitable function or subsystem that has a behavior which reacts according to a combination of inputs or events. The behavior of interest must not be too extensive (i.e. should not contain too many inputs) otherwise the number of combinations will become cumbersome and difficult to manage. It is better to deal with large numbers of conditions by dividing them into subsets and dealing with the subsets one at a time.

Once you have identified the aspects that need to be combined, then you put them into a table listing all the combinations of True and False for each of the aspects. Take an example of a loan application, where you can enter the amount

of the monthly repayment or the number of years you want to take to pay it back (the term of the loan). If you enter both, the system will make a compromise between the two if they conflict. The two conditions are the loan amount and the term, so we put them in a table (see Table 4.2).

TABLE 4.2 Empty decision table

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Repayment amount has been entered				
Term of loan has been entered				

Next we will identify all of the combinations of True and False (see Table 4.3). With two conditions, each of which can be True or False, we will have four combinations (two to the power of the number of things to be combined). Note that if we have three things to combine, we will have eight combinations, with four things, there are 16, etc. This is why it is good to tackle small sets of combinations at a time. In order to keep track of which combinations we have, we will alternate True and False on the bottom row, put two Trues and then two Falses on the row above the bottom row, etc., so the top row will have all Trues and then all Falses (and this principle applies to all such tables).

TABLE 4.3 Decision table with input combinations

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Repayment amount has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F

The next step (at least for this example) is to identify the correct outcome for each combination (see Table 4.4). In this example, we can enter one or both of the two fields. Each combination is sometimes referred to as a rule.

TABLE 4.4 Decision table with combinations and outcomes

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Repayment amount has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F
Actions/Outcomes				
Process loan amount	Y	Y		
Process term	Y		Y	

At this point, we may realize that we hadn't thought about what happens if the customer doesn't enter anything in either of the two fields. The table has highlighted a combination that was not mentioned in the specification for this example. We could assume that this combination should result in an error message, so we need to add another action (see Table 4.5). This highlights the strength of this technique to discover omissions and ambiguities in specifications. It is not unusual for some combinations to be omitted from specifications; therefore this is also a valuable technique to use when reviewing the test basis.

TABLE 4.5 Decision table with additional outcomes

Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**
Repayment amount has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F
Actions/Outcomes				
Process loan amount	Y	Y		
Process term	Y		Y	
Error message				Y

Suppose we change our example slightly, so that the customer is not allowed to enter both repayment and term. Now our table will change, because there should also be an error message if both are entered, so it will look like Table 4.6.

TABLE 4.6 Decision table with changed outcomes

Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**
Repayment amount has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F
Actions/Outcomes				
Process loan amount		Y		
Process term			Y	
Error message	Y			Y

You might notice now that there is only one 'Yes' in each column, i.e. our actions are mutually exclusive – only one action occurs for each combination of conditions. We could represent this in a different way by listing the actions in the cell of one row, as shown in Table 4.7. Note that if more than one action results from any of the combinations, then it would be better to show them as separate rows rather than combining them into one row.

TABLE 4.7 Decision table with outcomes in one row

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Repayment amount has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F
Actions/Outcomes				
Result	Error message	Process loan amount	Process term	Error message

The final step of this technique is to write test cases to exercise each of the four rules in our table.

In this example we started by identifying the input conditions and then identifying the outcomes. However in practice it might work the other way around – we can see that there are a number of different outcomes, and have to work back to understand what combination of input conditions actually drive those outcomes. The technique works just as well doing it in this way, and may well be an iterative approach as you discover more about the rules that drive the system.

Credit card worked example

Let's look at another example. If you are a new customer opening a credit card account, you will get a 15% discount on all your purchases today. If you are an existing customer and you hold a loyalty card, you get a 10% discount. If you have a coupon, you can get 20% off today (but it can't be used with the 'new customer' discount). Discount amounts are added, if applicable. This is shown in Table 4.8.

TABLE 4.8 Decision table for credit card example

Conditions	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
New customer (15%)	T	T	T	T	F	F	F	F
Loyalty card (10%)	T	T	F	F	T	T	F	F
Coupon (20%)	T	F	T	F	T	F	T	F
Actions								
Discount (%)	X	X	20	15	30	10	20	0

In Table 4.8, the conditions and actions are listed in the left hand column. All the other columns in the decision table each represent a separate rule, one for each combination of conditions. We may choose to test each rule/combination and if there are only a few this will usually be the case. However, if the number of rules combinations is large we are more likely to sample them by selecting a rich subset for testing.

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