

**Step 3.** Generate all combinations of conditions, and eliminate infeasible combinations of conditions. For each feasible combination, a separate column is created in the table with the values of each condition listed in this column.

**Step 4.** Identify, for each identified combination of conditions, which actions and how they should occur. This results in completing the bottom part of the corresponding column of the decision table.

**Step 5.** For each column of the decision table, design a test case in which the test input represents combination of conditions specified in this column. The test is passed if, after its execution, the system takes actions as described at the bottom part of the table in the corresponding column. These action entries serve as the expected output for the test case.

### Notation and Possible Entries in the Decision Table

Typically, condition and action values take the form of logical values TRUE or FALSE. They can be represented in various ways, such as the symbols T and F or Y and N (yes/no) or 1 and 0 or as the words “true” and “false.” However, the values of conditions and actions can in general be any objects, such as numbers, ranges of numbers, category values, equivalence partitions, and so on. For example, in our table (Table 4.4), the values of the “discount granted” action are categories expressing different types of discounts: 0%, 5%, and 10%. In the same table, there can be conditions and actions of different types, e.g., logical, numerical, and categorical conditions can occur simultaneously.

The decision table with only Boolean (true/false) values is called a *limited-entry decision table*. If for any condition or action there are other than Boolean entries, such a table is called an *extended-entry decision table*.

### How to Determine all Combinations of Conditions

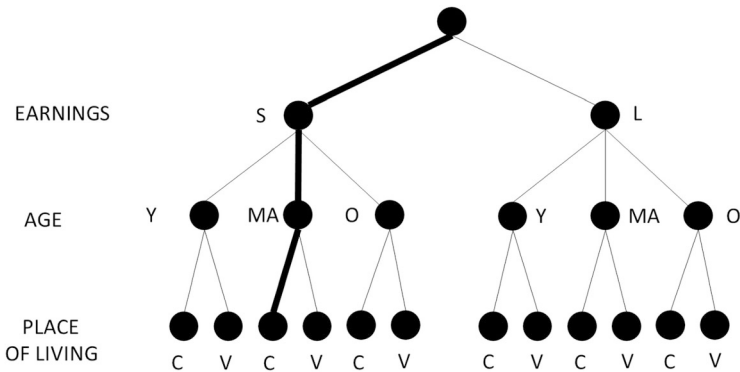
If we need to manually determine combinations of conditions, and we are afraid that we will miss some combinations, we can use a very simple tree method to systematically determine all combinations of conditions. Consider the following example:

**Example** Suppose a decision table has three conditions:

- Earnings (two possible values—S, small; L, large)
- Age (three possible values—Y, young; MA, middle aged; O, old)
- Place of living (two possible values—C, city; V, village)

To create all combinations of values of triples (age, earnings, residence), we build a tree, from the root of which we derive all possibilities of the first condition (earnings). This is the first level of the tree. Next, from each vertex of this level, we derive all the possibilities of the second condition (age). We get the second level of the tree. Finally, from each vertex of this level, we derive all the possible values of the third condition (place of living). Of course, if there were more conditions, we would proceed analogously. Our final tree looks like the one in Fig. 4.8.

Each possible combination of conditions is a combination of vertex labels on the paths leading from the root (the vertex at the top of the tree) to any vertex at the very bottom of the tree. Thus, there will be as many combinations as there are vertices at



**Fig. 4.8** Support tree for identifying combinations of conditions

the lowest level (in our case—12; this, of course, follows from the number of combinations:  $2 \times 3 \times 2 = 12$ ). In the figure, the bold lines indicate the path corresponding to the example combination (S, MA, C), which means small earnings, middle age, and city as the place of living. We can now enter each of these combinations into the individual columns of our decision table. At the same time, we are sure that no combination has been left out. The conditions of the decision table thus created are shown in Table 4.5.

### Infeasible Combinations

Sometimes, after listing all possible combinations of conditions, you may find that some of them are *infeasible* for various reasons. For example, suppose we have the following two conditions in the decision table:

- Customer's age  $> 18$ ? (possible values: YES, NO)
- Customer's age  $\leq 18$ ? (possible values: YES, NO)

It is obvious that although we have four possible combinations of these conditions, (YES, YES), (YES, NO), (NO, YES), and (NO, NO), only two of them are feasible, (YES, NO) and (NO, YES), since it is impossible to be more than 18 and less than 19 years old at the same time. Of course, in this case, we could replace these two conditions with one, “age”, and with two possible values: greater than 18 and less than or equal to 18.

Sometimes, the decision table will not contain some combinations not for purely logical reasons but for semantic reasons. For example, if we have two conditions:

- Was the goal defined? (YES, NO)
- Was the goal achieved? (YES, NO)

then the combination (NO, YES) is infeasible (nonsensical), because it is impossible to achieve a goal that you have not previously defined.

**Table 4.5** Combinations of decision table conditions formed from the support tree

	1	2	3	4	5	6	7	8	9	10	11	12
Earnings	S	S	S	S	S	S	L	L	L	L	L	L
Age	Y	Y	MA	MA	O	O	Y	Y	MA	MA	O	O
Place of residence	C	V	C	V	C	V	C	V	C	V	C	V

### Minimizing the Decision Table

Sometimes, some conditions may not have any effect on the actions taken by the system. For example, if the system allows only adult customers to buy insurance, and depending on whether they smoke or not they get a discount on that insurance, then as long as the customer is a minor, the system will not allow them to buy insurance regardless of whether the customer smokes or not. Such irrelevant values are most often marked in the decision table with a dash symbol or N/A (*not applicable*). Typically, a dash is used when the corresponding condition occurs, but its value is irrelevant for determining the action. The N/A symbol, on the other hand, is used when the condition cannot occur. For example, consider two conditions: “payment type” (card or cash) and “is the PIN correct?” (yes or no). If the payment type is cash, we can’t even check the value of the condition “is the PIN correct?” because the condition doesn’t occur at all. So we have only three possible combinations of conditions: (card, yes), (card, no) and (cash, N/A).

This minimization, or collapsing, makes the decision table more compact with fewer columns and therefore fewer test cases to execute. On the other hand, for a test with an irrelevant value in the actual test case, we have to choose some specific, concrete value for this condition. So, there is a risk that if a defect occurs for some specific combination of values marked as irrelevant in the decision table, we can easily miss it and not test it.

Minimizing decision tables is however a risk mitigation exercise: maybe the current mock-up of the GUI does not allow certain input, but the actual implementation might or a future API might just as well.

Consider the collapsed decision table shown in Table 4.6. If we wanted to design a concrete test case for the first column, we would have to decide whether the customer smokes or not (although from the point of view of the specification this is irrelevant), because this input should be given. We can decide on the combination (adult = NO, smokes = NO), and such a test will pass, but we can imagine that due to some defect in the code, the program does not work properly for the combination (adult = NO, smokes = YES). Such a combination has not been tested by us, and failure will not be detected.

On the actual exam, there may be questions involving minimized decision tables, but the candidate is not required to be able to perform minimization but only to be able to understand, interpret, and use decision tables that are already minimized. The minimization is required on the Advanced Level—Test Analyst certification exam. Therefore, in this book, we do not present an algorithm for minimizing decision tables.

### Coverage

In decision table testing, the coverage items are the individual columns of the table, containing possible combinations of conditions (i.e., the so-called feasible columns).

**Table 4.6** Decision table with irrelevant values

CONDITIONS			
Adult?	NO	YES	YES
Smokes?	–	YES	NO
ACTIONS			
Grant insurance?	NO	YES	YES
Grant discount?	NO	NO	YES

For a given decision table, full 100% coverage requires that at least one test case corresponding to each feasible column be prepared and executed. The test is passed if the system actually executes the actions defined for that column.

The important thing is that coverage counts against the number of (feasible) columns of the decision table, not against the number of all possible combinations of conditions. Usually, these two numbers are equal, but in the case of the occurrence of infeasible combinations, as we discussed earlier, this might not be the case.

For example, in order to achieve 100% coverage for the decision table in Table 4.6, we need three (not four, as the number of combinations would suggest) test cases. If we had the following tests:

- Adult = YES, smokes = YES.
- Adult = NO, smokes = YES.
- Adult = NO, smokes = NO.

then we would achieve 2/3 coverage (or about 66%), since the last two test cases cover the same, first column of the table.


**Decision Tables as a Static Testing Technique**

The decision table testing is excellent for detecting problems with requirements, such as their absence or contradiction. Once the decision table is created from the specification or even while it is still being created, it is very easy to discover such specification problems as:

- Incompleteness—no defined actions for a specific combination of conditions
- Contradiction—defining in two different places of specification two different behaviors of the system against the same combination of conditions
- Redundancy—defining the same system behavior in two different places in the specification (perhaps described differently)

**4.2.4 State Transition Testing**

**Application**

**State transition testing**  is a technique used to check *the behavior* of a component or system. Thus, it checks its behavioral aspect—how it behaves over time and how it changes its state under the influence of various types of events.

The model describing this behavioral aspect is the so-called state transition diagram. In the literature, different variants of this model are called a finite automaton, finite state automaton, state machine, or *Labeled Transition System*. The

syllabus uses the name “state transition diagram” to denote the graphical form of the state transition model and “state transition table” to denote the equivalent, tabular form of the model.

### Construction of the State Transition Diagram

A state transition diagram is a graphical model, as described in the UML standard. From a theoretical point of view, it is a labeled directed graph. The state transition diagram consists of the following elements:

- States—represent possible situations in which the system may be
- Transitions—represent possible (correct) changes of states
- Events—represent phenomena, usually external to the system, the occurrence of which triggers the corresponding transitions
- Actions—actions that the system can take during the transition between states
- Guard conditions—logical conditions associated with transitions; a transition can be executed only if the associated guard condition is true

Figure 4.9 shows an example of a state transition diagram. It is not trivial, although in practice even more complicated models are often used, using richer notation than that discussed in the syllabus. However, this diagram allows us to show all the essential elements of the state transition model described in the syllabus while keeping the example practical.

The diagram represents a model of the system’s behavior for making a phone call to a cell phone user with a specific number. The user dials a nine-digit phone number by pressing the keys corresponding to successive digits of the number one by one. When the ninth digit is entered, the system automatically attempts the call.

The state transition diagram modeling this system consists of five states (rectangles). The possible transitions between them are indicated by arrows. The system starts in the initial state *Welcome screen* and waits for an event (labeled *EnterDigit*) involving the user selecting the first digit of a nine-digit phone number. When this event occurs, the system transitions to the *Entering* state, and in addition, during this transition, the system performs an action to set the value of the *x* variable to 1. This variable will represent the number of digits of the dialed phone number entered by the user so far.

In the *Entering* state, only the *EnterDigit* event can occur, but depending on how many digits have been entered so far, transitions to two different states are possible. As long as the user has not entered the ninth digit, the system remains in the *Entering* state, each time increasing the *x* variable by one. This is because the guard condition “ $x < 8$ ” is true in this situation. Just before the user enters the last, ninth digit, the variable *x* has a value of 8. This means that the guard condition “ $x < 8$ ” is false and the guard condition “ $x = 8$ ” is true. Therefore, selecting the last, ninth digit of the number will switch from the *Entering* state under the *EnterDigit* event to the *Connect* state.

When the call succeeds (the occurrence of the *ConnectionOK* event), the system enters the *Call* state, in which it remains until the user terminates the call, which will be signaled by the occurrence of the *EndConnection* event. At this point, the system goes to the final state *End*, and its operation ends. If the system is in the *Connect* state and the *ConnectionError* event occurs, the system transitions to the final state, but unlike the analogous transition from the *Call* state, it will additionally perform the

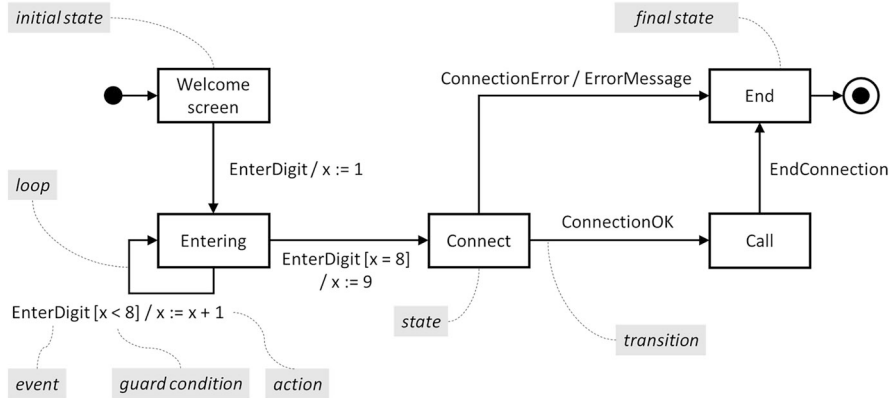


Fig. 4.9 Example of state transition diagram

*ErrorMessage* action, signaling to the user the inability to connect to the selected number.

The system at each moment of time is in exactly one of the states, and the change of states occurs as a result of the occurrence of corresponding events. The concept of state is abstract. A state can denote a very-high-level situation (e.g., the system being in a certain application screen), but it can also describe low-level situations (e.g., the system executing a certain program statement). The level of abstraction depends on the model adopted, i.e., what the model actually describes and at what level of generality. It is assumed that when a certain event occurs, the change of states is instantaneous (it can be assumed that it is a zero-duration event). The transition labels (i.e., arrows on the diagram) are in general of the form:

event [guard condition] / action

If, in a given case, the guard condition or action does not exist or is not relevant from the tester's point of view, they can be omitted. Thus, transition labels can also take one of the following three forms:

Event

Event/action

Event [guard condition]

A guard condition for a given transition allows that transition to be executed only if the condition holds. Guard conditions allow us to define two different transitions under the same event while avoiding non-determinism. For example, in the diagram in Fig. 4.9 from the *Entering* state, we have two transitions under the *EnterDigit* event, but only one of them can be executed at any time because the corresponding guard conditions are disjoint (*either* *x* is less than 8 *or* is equal to 8).

An example test case for the state transition diagram in Fig. 4.9, verifying the correctness of a successful connection, might look like the one in Table 4.7.

**Table 4.7** Example of a sequence of transitions in the state transition diagram of Fig. 4.9

Step	State	Event	Action	Next state
1	Welcome screen	EnterDigit	$x := 1$	Entering
2	Entering	EnterDigit	$x := x + 1$	Entering
3	Entering	EnterDigit	$x := x + 1$	Entering
4	Entering	EnterDigit	$x := x + 1$	Entering
5	Entering	EnterDigit	$x := x + 1$	Entering
6	Entering	EnterDigit	$x := x + 1$	Entering
7	Entering	EnterDigit	$x := x + 1$	Entering
8	Entering	EnterDigit	$x := x + 1$	Entering
9	Entering	EnterDigit	$x := 9$	Connect
10	Connect	ConnectionOK		Call
11	Call	EndConnection		End

The scenario is triggered by a sequence of 11 events: *EnterDigit* (9 times), *ConnectionOK*, and *EndConnection*. In each step, we trigger the corresponding event and check whether, after its occurrence, the system actually goes to the state described in the *Next state* column.

**Equivalent Forms of State Transition Diagram**

There are at least two equivalent forms of representation of the state transition diagram. Consider a simple, four-state machine with states S1, S2, S3, and S4 and events A, B, and C. The state transition diagram is shown in Fig. 4.10a.

Equivalently, however, it can be presented in the form of a *state table*, where individual rows (respectively, columns) of the table represent successive states (respectively, events, together with guard conditions if they exist), and in the cell at the intersection of the column and row corresponding to the specified state S and event E is written the target state to which the system is to transition if, being in state S, event E occurs. For example, if the system is currently in state S2 and event B has occurred, the system is to move to state S1. If our diagram used actions, we would have to annotate them in the corresponding cells of the tables in Fig. 4.10b, c.

One more way of representing the state machine is by using the *full transition table* shown in Fig. 4.10c. Here the table represents *all possible combinations of* states, events, and guard conditions (if they exist). Since we have four states, three different events, and no guard conditions, the table will contain  $4 * 3 = 12$  rows. The last column contains the target state to which the machine is to transition if, when it is in the state defined in the first column of the table, the event described in the second column of the table occurs. If the transition in question is undefined, the *Next state column* indicates this, e.g., with a dash or other fixed symbol.

The absence of a transition (i.e., the absence of a state/event combination) on the state transition diagram is represented simply by the absence of the corresponding arrow. For example, being in state S1, the machine has no defined behavior for the occurrence of event B. Therefore, there is no outgoing arrow from S1 labeled by B.

Both the state table and the full transition table allow us to directly show so-called invalid transitions, which can also be, and in some situations should be, tested. Invalid transitions (the “missing arrows” in the state transition diagram) are

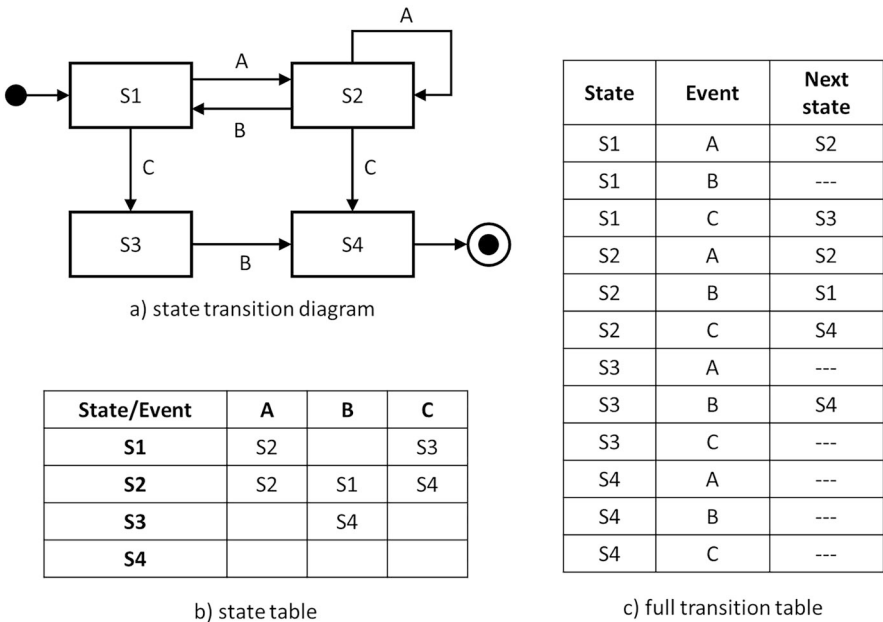


Fig. 4.10 Different forms of state machine presentation

represented by empty cells in the tables. In other words, an invalid transition is any combination of state and event that does not appear on the state transition diagram. For example, the diagram in Fig. 4.10 is missing six transitions: (S1, B), (S3, A), (S3, C), (S4, A), (S4, B), and (S4, C). Thus, we have a total of six invalid transitions, corresponding to six empty cells in the table in Fig. 4.10b or six empty cells in the “Next state” column in the table in Fig. 4.10c.

Sometimes, for the sake of simplicity, only one arrow between two states is drawn on the state transition diagram, even if there are more parallel transitions between these states. It is important to pay attention to this, because sometimes, the question of the number of transitions on the diagram is very important. Figure 4.11 shows an example of a “simplified” form of drawing transitions and an equivalent “full” form.

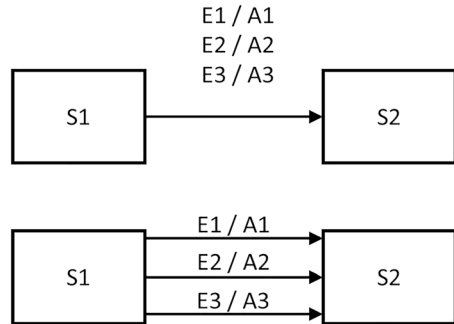
Test Design: Coverage

For state transition testing, there are many different coverage criteria. Here are the three most popular criteria, described in the Foundation Level syllabus:

- **All states coverage**—the weakest coverage criterion. The coverage items are the states. All states coverage therefore requires that the system has been in each state at least once.
- **Valid transitions coverage** (also called 0-switch coverage or Chow’s coverage)—the most popular coverage criterion. The coverage items are transitions between states. Valid transitions coverage therefore requires that each transition defined in the state transition diagram is executed at least once.



**Fig. 4.11** Equivalent forms of graphical representation of parallel transitions



- **All transitions coverage**—in which the coverage items are all transitions indicated in the state table. Thus, this criterion requires that all valid transitions be covered and, in addition, the execution of every invalid transition be attempted. It is good practice to test one such event per test (to avoid defect masking).

Other coverage criteria can be defined, such as transition pair coverage (also called 1-switch coverage), which requires that every possible sequence of two consecutive valid transitions be tested. Criteria of this type can be generalized: we can require the coverage of all threes of transitions, all fours of transitions, etc. In general—we can define a whole family of coverage criteria regarding  $N+1$  consecutive transitions, for any non-negative  $N$  (called  $N$ -switch coverage). It is also possible to consider coverage of some particular paths, coverage of loops, etc. (all these coverage criteria are not in the scope of the Foundation Level syllabus).

Thus, in the case of state transition testing, we are dealing with a potentially infinite number of possible coverage criteria. From a practical point of view, the most commonly used criteria are the valid transitions coverage and the all transitions coverage, since the main thing we are usually concerned with when testing the state transition diagram is verification of the valid implementation of *transitions* between states.

Coverage is defined as the number of elements covered by the tests relative to the number of all coverage items defined by the criterion. For example, for the all states coverage criterion applied to the transition diagram in Fig. 4.10, we have four elements to cover: states S1, S2, S3, and S4; for the valid transitions coverage criterion, we have as many elements as there are transitions between states (six).

The usual requirement is to design *the smallest possible* number of test cases that are sufficient to achieve full coverage. Let us see how different types of coverage can be realized for the state transition diagram shown in Fig. 4.10a.

For the *all states coverage* criterion, we have four states to cover: S1, S2, S3, and S4. Note that this can be achieved within a single test case, for example:

S1 (A) S2 (B) S1 (C) S3 (B) S4.

The convention used above describes the sequence of transitions between states under the influence of events (indicated in parentheses). The notation “S (E) T” denotes the transition from state S under the influence of event E to state T. In the

**Table 4.8** Test scenario for state transition testing

Step	Initial condition	Event	Expected result
1	S1	A	Transition to S2
2	S2	B	Transition to S1
3	S1	C	Transition to S3
4	S3	B	Transition to S4
5	S4		

above example, we went through all four states within one test case, so this single test case achieved 100% state coverage. In practice, the expected test result is the actual passing of states one by one as described by the model. The test scenario corresponding to the test case:

S1 (A) S2 (B) S1 (C) S3 (B) S4

could look like this, as described in Table 4.8.

Note that a test case *is not equated* with a test condition. In our example, the test conditions were individual states, but a *single* test case could cover them all. A test case is a sequence of transitions between states, starting with the initial state and ending with the final state (possibly interrupting this trek earlier if necessary). So, it is possible to cover *more than one* test condition within a single test case.

For the *valid transitions coverage* criterion, we have six transitions to cover. Let us denote them as T1–T6:

- T1: S1 (A) S2
- T2: S1 (C) S3
- T3: S2 (A) S2
- T4: S2 (B) S1
- T5: S2 (C) S4
- T6: S3 (B) S4

We want to design as few tests as possible to cover all of these six transitions. The strategy is to try to cover as much as possible the previously uncovered items within each successive test. For example, if we started the first test case with the sequence S1 (A) S2, it would not be worthwhile at this point to trigger event C and move to the final state S4, when we can still cover several other transitions, such as S2 (A) S2 (B) S1.

Note that it is impossible to cover, within a single test case, the transitions S2 (C) S4 and S3 (B) S4, because as soon as S4 is reached, the test case must end. So, we will need at least two test cases to cover all transitions. Indeed, two cases are enough to achieve full valid transitions coverage. An example set of such two test cases is shown in Table 4.9.

In this table, the first column gives the sequence of transitions, and the second column gives the corresponding covered transitions. Transitions covered for the first time are indicated in bold (e.g., in the second test, we cover T1, which was already covered previously with the first test case).

**Table 4.9** Test cases achieving full valid transitions coverage

Test case	Transitions covered
S1 (A) S2 (A) S2 (B) S1 (C) S3 (B) S4	<b>T1, T3, T4, T2, T6</b>
S1 (A) S2 (C) S4	<b>T1, T5</b>

To achieve the all transitions coverage, we must provide test cases covering all valid transitions (see above) and—in addition—try to exercise each invalid transition. According to the good practice described above, we will test each such transition in a separate test case. Thus, the number of test cases will be equal to the number of test cases covering all valid transitions plus the number of invalid transitions. In our model, this will be the following six invalid transitions (you can quickly write them out by analyzing the full state table—see Fig. 4.10c):

- S1 (B) ?
- S3 (A) ?
- S3 (C) ?
- S4 (A) ?
- S4 (B) ?
- S4 (C) ?

Remember that each test case starts with an initial state. Thus, for each invalid transition, we must first call the sequence of events that reaches the given state and then, once we are in it, try to call the invalid transition. For the six invalid transitions described above, the corresponding six test cases might look like in Table 4.10.

If we manage to trigger an event that is not defined in the model, we can interpret it in at least two ways:

- If the system changed its state, this should be considered a failure, because since the model does not allow such transition, it should not be possible to trigger it.
- If the system did not change its state, then this can be considered correct behavior (ignoring the event). However, we must be sure that semantically, this is an acceptable situation.

There are also at least two possible solutions to such a problematic situations:

- Fix the system so that the event is not possible (the invalid transition is impossible to be triggered).

**Table 4.10** Test cases covering invalid transitions

Test case	Incorrect transition covered
S1 (B) ?	S1 (B) ?
S1 (C) S3 (A) ?	S3 (A) ?
S1 (C) S3 (C) ?	S3 (C) ?
S1 (C) S3 (B) S4 (A) ?	S4 (A) ?
S1 (C) S3 (B) S4 (B) ?	S4 (B) ?
S1 (C) S3 (B) S4 (C) ?	S4 (C) ?

- Add a transition under the influence of this event to the model so that it models the “ignoring” of the event by the system (e.g., a loop to the same state).

Let us go back to the practical example of the state transition diagram in Fig. 4.9 modeling a phone call. An example of an invalid transition is the transition from the *Connect* state in response to the *EnterDigit* event. Such a situation is triggered very simply—at the time of establishing a call, we simply press one of the keys representing digits. If the system does not react to this in any way, we consider that the test is passed.

Finally, let us discuss one of the coverage criteria not described in the syllabus—the coverage of pairs of valid transitions (1-switch coverage). This is an extra material, non-examinable at the Foundation Level certification exam.

We consider this example to show that the stronger the criterion we adopt, the more difficult it is, and it usually requires designing more test cases than for a weaker criterion (in this case, the weaker criterion is the valid transitions coverage).

To satisfy the 1-switch coverage, we need to define all allowed pairs of valid transitions. We can do this in the following way: for each single, valid transition, we consider all its possible continuations in the form of the following single, valid transition. For example, for the transition S1 (A) S2, the possible continuations are all transitions coming out of S2, that is, S2 (A) S2, S2 (B) S1, and S2 (C) S4. All possible pairs of valid transitions thus look like this:

- PP1: S1 (A) S2 (A) S2
- PP2: S1 (A) S2 (B) S1
- PP3: S1 (A) S2 (C) S4
- PP4: S1 (C) S3 (B) S4
- PP5: S2 (A) S2 (A) S2
- PP6: S2 (A) S2 (B) S1
- PP7: S2 (A) S2 (C) S4
- PP8: S2 (B) S1 (A) S2
- PP9: S2 (B) S1 (C) S3

Note (analogous to the case of valid transitions coverage) that the transition pairs PP3, PP4, and PP7 terminate in the final state, so no two of them can occur in a single test case, since the occurrence of any of these transition pairs results in the termination of the test case. Thus, we need at least three test cases to cover pairs of valid transitions, and indeed three test cases are sufficient. An example test set is given in Table 4.11 (pairs covered for the first time are indicated in bold). So, we need one more test case compared to the case of valid transitions coverage.

**Table 4.11** Test cases covering pairs of valid transitions

Test case	Covered pairs of valid transitions
S1 (A) S2 (A) S2 (A) S2 (B) S1 (A) S2 (B) S1 (C) S3 (B) S4	<b>PP1, PP5, PP6, PP8, PP2, PP9, PP4</b>
S1 (A) S2 (C) S4	<b>PP3</b>
S1 (A) S2 (A) S2 (C) S4	PP1, <b>PP7</b>

### 4.2.5 (\*) *Use Case Testing*

#### **Application**

A use case is a requirements document that describes the interaction between so-called actors, which are most often the user and the system.

A use case is a description of a sequence of steps that the user and the system perform, which ultimately leads to the user obtaining some benefit. Each use case should describe a *single*, well-defined scenario. For example, if we are documenting the requirements for an ATM (a cash machine), use cases could, among other things, describe the following scenarios:

- Rejecting an invalid ATM card
- Logging into the system by entering the correct PIN
- Correctly withdrawing money from an ATM
- Attempted failed withdrawal of money due to insufficient funds in the account
- Locking the card by entering the wrong PIN three times

For each use case, the tester can construct a corresponding test case, as well as a set of test cases to check the occurrence of unexpected events during the scenario run.

#### **Use Case Construction**

A correctly built use case, in addition to “technical” information such as a unique number and name, should consist of:

- Pre-conditions (including input data)
- Exactly one, sequential main scenario
- (Optional) Markings of the locations of so-called alternative flows and exceptions
- Post-conditions (describing the state of the system after successful execution of the scenario and the user benefit obtained)

From the tester’s point of view, the primary purpose of the use case is to test the main scenario, i.e., to verify that actually performing all the steps as the scenario describes leads to the defined user benefit. However, in doing so, it is also necessary to test the system’s behavior for “unsuspected” events, that is, alternative flows and exceptions. The difference between these two is as follows:

- An alternative flow causes an unexpected event to occur but allows the user to complete the use case, i.e., it allows the user to return to the main scenario and happily complete it.
- An exception interrupts the execution of the main scenario—if an exception occurs, it is not possible to complete the main scenario correctly. The user gets no benefit, and the use case either ends with an error message or is aborted. The occurrence of the exception itself can be handled through a scenario defined in a separate use case.

A use case should not contain business logic, i.e., the main scenario should be linear and contain no branching. The existence of such branching suggests that we