

# Advanced Software Test Design Techniques State Diagrams, State Tables, and Switch Coverage

Provided by Rex Black Consulting Services ([www.rbc-us.com](http://www.rbc-us.com))

## Introduction

The following is an excerpt from my recently-published book, *Advanced Software Testing: Volume 1*. This is a book for test analysts and test engineers. It is especially useful for ISTQB Advanced Test Analyst certificate candidates, but contains detailed discussions of test design techniques that any tester can – and should – use. In this second article in a series of excerpts, I discuss the powerful test techniques of state transition diagrams, state transition tables, and switch coverage.

## State-Based Testing and State Transition Diagrams

At the start of this series, I said we would cover three techniques that would prove useful for testing business logic, often more useful than equivalence partitioning and boundary value analysis. We covered decision tables, which are best in transactional testing situations. In the next article, we'll cover use cases, where preconditions and postconditions help to insulate one workflow from the previous workflow and the next workflow.

In this article, we look at state-based testing. State-based testing is ideal when we have sequences of events that occur and conditions that apply to those events, and the proper handling of a particular event/condition situation depends on the events and conditions that have occurred in the past. In some cases, the sequences of events can be potentially infinite, which of course exceeds our testing capabilities, but we want to have a test design technique that allows us to handle arbitrarily-long sequences of events.

The underlying model is a state transition diagram or table. The diagram or table connects beginning states, events, and conditions with resulting states and actions.

In other words, some status quo prevailed and the system was in a current state. Then some event occurs, some event that the system must handle. The handling of that event might be influenced by one or more conditions. The event/condition combination triggers a state transition, either from the current state to a new state or from the current state back to the current state again. In the course of the transition, the system takes one or more actions.

Given this model, we generate tests that traverse the states and transitions. The inputs trigger events and create conditions, while the expected results of the test are the new states and actions taken by the system.

Various coverage criteria apply for state based testing. The weakest criterion requires that the tests visit every state and traverse every transition. We can apply this criterion to state transition diagrams. A higher coverage criterion is at least one test cover every row in a state transition table. Achieving “every-row” coverage will achieve “every state and transition” coverage, which is why I said it was a high coverage criterion.

Another potentially higher coverage criterion requires that at least one test cover each transition sequence of N or less length. The N can be 1, 2, 3, 4, or higher. This is called alternatively “Chow’s switch coverage” – after the Professor Chow who developed it – or “N-1 switch coverage,” after the level given to the degree of coverage. If you cover all transitions of length one, then “N-1 switch coverage” means “0 switch coverage.” Notice that this is the same as the lowest level of coverage discussed. If you cover all transitions of length one and two, then “N-1 switch coverage” means “1 switch coverage”. This is a higher level of coverage than the lowest level, of course.

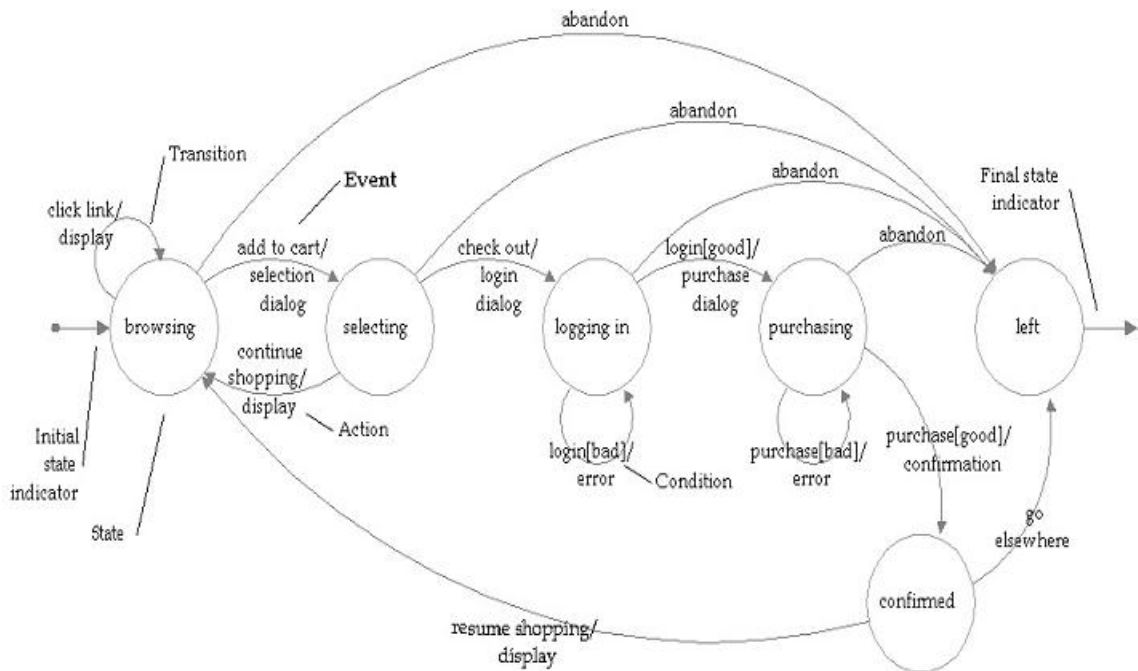
Now, “1 switch coverage” is not necessarily a higher level of coverage than “every-row” coverage. This is because the state transition table forces testing of state and event/condition combinations that do not occur in the state-transition diagram. The so-called “switches” in “N-1 switch coverage” are derived from the state transition diagram, not the state transition table.

You might find all this a bit confusing if you’re fuzzy on the test design material covered at the Foundation level. This is a common problem for those who took “brain cram” courses to prepare for the Foundation exam. Don’t worry, though, it will be clear to you shortly.

So, what is the bug hypothesis in state-based testing? We’re looking for situations where the wrong action or the wrong new state occurs in response to a particular event under a given set of conditions based on the history of event/condition combinations so far.

Figure 1 shows the state transition diagram for shopping and buying items online from an e-commerce application. It shows the interaction of the system with a customer, from the customer’s point of view. Let’s walk through it, and I’ll point out the key elements of state transition diagrams in general and the features of this one in particular.

First, notice that we have at the left-most side a small dot-and-arrow element labeled “initial state indicator”. This notation shows that, from the customer’s point of view, the transaction starts when she starts browsing the Web site. We can click on links and browse the catalog of items, remaining in a browsing state. Notice that the looping arrow above the browsing state. The nodes or bubbles represent states, as shown by the label below the browsing state. The arrows represent transitions, as shown by the label above the looping arrow.



**Figure 1: State Transition Diagram Example**

Next, we see that we can enter a “selecting” state by adding an item to the shopping cart. “add to cart” is the event, as shown by the label above. The system will display a “selection dialog” where we ask the customer to tell us how many of the item she wants, along with any other information we need to add the item to the cart. Once that’s done, the customer can tell the system she wants to continue shopping. In which case, the system displays the home screen again, and the customer is back in a browsing state. From a notation point of view, notice that the actions taken by the system are shown under the event and after the slash symbol, on the transition arrow, as shown by the label below.

Alternatively, the customer can choose to check out. At this point, she enters a logging-in state. She enters login information. A condition applies to that login information: either it was good or it was bad. If it was bad, the system displays an error and the customer remains in the logging-in state. If it was good, the system displays the first screen in the purchasing dialog. Notice that the “bad” and “good” shown in brackets are, notationally, conditions.

While in the purchasing state, the system will display screens and the customer will enter payment information. Either that information is good or bad – conditions again – which determines whether we can complete and confirm the transaction. Once the transaction is confirmed, the customer can either resume shopping or go somewhere else.

Notice also that the user can always abandon the transaction and go elsewhere.

When we talk about state-based testing during our training courses, people often ask, “How do I distinguish a state, an event, and an action?” The main distinctions are as follows:

A state persists, until something happens – something external to the thing itself, usually – to trigger a transition. A state can persist for an indefinite period.

An event occurs, either instantly or in a limited, finite period. It is the something that happened – the external occurrence – that triggered the transition.

An action is the response the system has during the transition. An action, like an event, is either instantaneous or requires a limited, finite period.

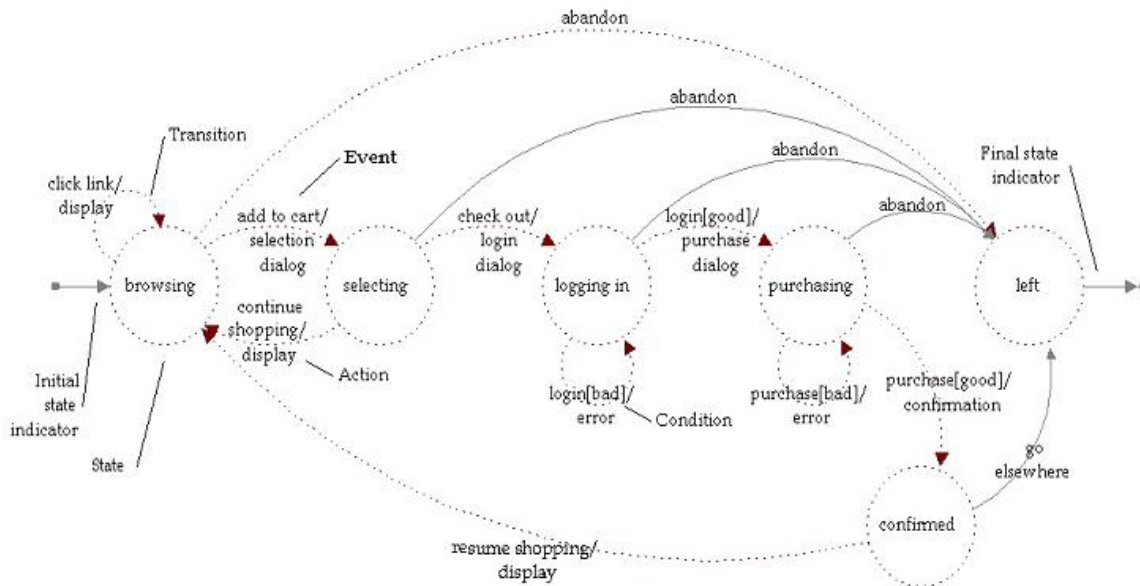
That said, it is sometimes possible to draw the same situation differently, especially when a single state or action can be split into a sequence of finer-grained states, events, and actions. We’ll see an example of that in a moment, splitting the purchase state into substates.

Finally, notice that, at the outset, I said that I drew Figure 1 from the customer’s point of view. Notice that, if I drew this from the system’s point of view, it would look different. Maintaining a consistent point of view is critical when drawing these charts, otherwise nonsensical things will happen.

State-based testing uses a formal model, so we can have a formal procedure for deriving tests from them. The list below shows a procedure that will work to derive tests that achieve state/transition cover (i.e., “0 switch cover”).

1. Adopt a rule for where a test procedure or test step must start and where it may or must end. An example is to say that a test step must start in an initial state and may only end in a final state. The reason for the “may” or “must” wording on the ending part is because, in situations where the initial and final states are the same, you might want to allow sequences of states and transitions that pass through the initial state more than once.
2. From an allowed test starting state, define a sequence of event/condition combinations that leads to an allowed test ending state. For each transition that will occur, capture the expected action that the system should take. This is the expect result.
3. As you visit each state and traverse each transition, mark it as covered. The easiest way to do this is to print the state transition diagram and then use a marker to highlight each node and arrow as you cover it.
4. Repeat steps 2 and 3 until all states have been visited and all transitions traversed. In other words, every node and arrow has been marked with the marker.

This procedure will generate logical test cases. To create concrete test cases, you’d have to generate the actual input values and the actual output values. For this course, I intend to generate logical tests to illustrate the techniques, but remember, as I mentioned before, at some point before execution, the implementation of concrete test case must occur.



**Figure 2: Coverage Check 1**

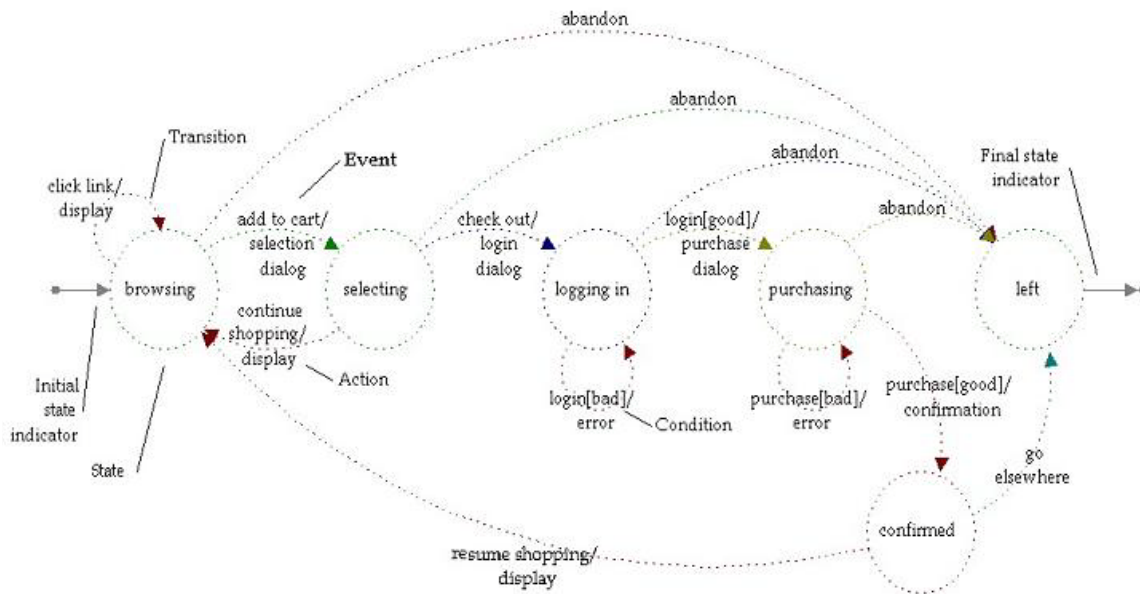
Let's apply this process to the example e-commerce application we've just looked at, as shown in Figure 2, where the dashed lines indicate states and transitions that were covered. Here, we see two things:

First, we have the rule that says that a test must start in the initial state and must end in the final state.

Next, we generate the first test step. (browsing, click link, display, add to cart, selection dialog, continue shopping, display, add to cart, selection dialog, checkout, login dialog, login[bad], error, login[good], purchase dialog, purchase[bad], error, purchase[good], confirmation, resume shopping, display, abandon, left).

At this point, we check completeness of coverage, which we've been keeping track of on our state transition diagram.

As you can see, we covered all of the states and most transitions, but not all of the transitions. We need to create some more tests.



**Figure 3: Coverage Check Completed**

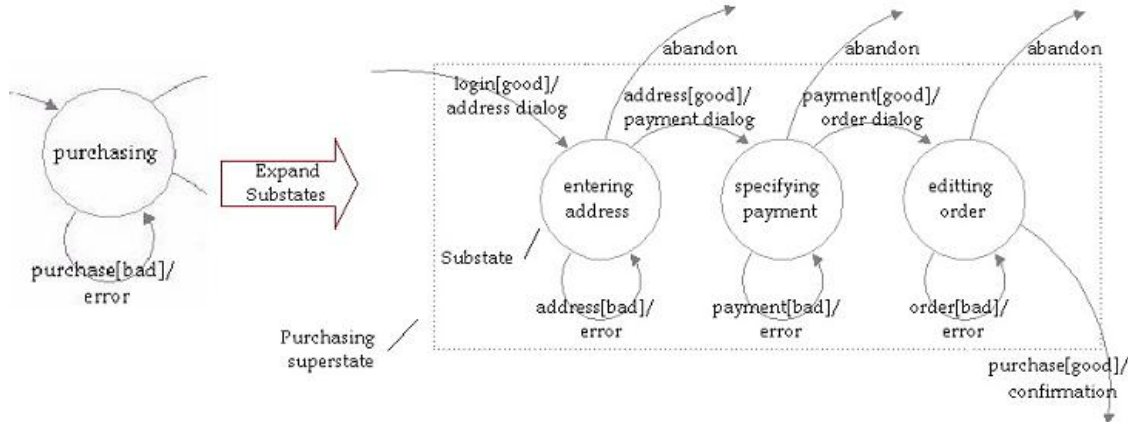
Figure 3 shows the test coverage achieved by the following addition test steps, which covers the state transition diagram:

1. (browsing, click link, display, add to cart, selection dialog, continue shopping, display, add to cart, selection dialog, checkout, login dialog, login[bad], error, login[good], purchase dialog, purchase[bad], error, purchase[good], confirmation, resume shopping, display, abandon, left)
2. (browsing, add to cart, selection dialog, abandon, <no action>, left)
3. (browsing, add to cart, selection dialog, checkout, login dialog, abandon, <no action>, left)
4. (browsing, add to cart, selection dialog, checkout, login dialog, login[good], purchase dialog, abandon, <no action>, left)
5. (browsing, add to cart, selection dialog, continue shopping, display, add to cart, selection dialog, checkout, login dialog, login[good], purchase dialog, purchase[good], confirmation, go elsewhere, <no action>, left)

Again, Figure 3 shows the coverage tracing for the states and transitions. You're not done generating tests until every state and every transition has been highlighted, as shown here.

## Superstates and Substates

In some cases, it makes sense to unfold a single state into a superstate consisting of two or more substates. In Figure 4, you see that I've taken the purchasing state from the e-commerce example and



expanded it into three substates.

**Figure 4: Superstates and Substates**

The rule for basic coverage here follows simply. Cover all transitions into the superstate, all transitions out of the superstate, all substates, and all transitions within the superstate.

Notice that, in our example, this would increase the number of tests, because we now have three “abandon” transitions to the “left” state out of the purchasing superstate, rather than just one transition from the purchasing state. This would also add a finer-grained element to our tests—i.e., more events and actions—as well as making sure we tested at least three different types of bad purchasing entries.

## State Transition Tables

State transition tables are useful because they force us—and the business analysts and the system designers—to consider combinations of states with event/condition combinations that they might have forgotten.

To construct a state transition table, you first list all the states from the state transition diagram. Next, you list all the event/condition combinations shown on the state transition diagram. Then, you create a table that has a row for each state with every event/condition combination. Each row has four fields:

- . Current state
- . Event/condition

□. Action



□. New state

For those rows where the state transition diagram specifies the action and new state for the given combination of current state and event/condition, we can populate those two fields from the state transition diagram. However, for the other rows in the table, we have found undefined situations.

We can now go to the business analysts, system designers, and other such people and ask, "So, what exactly should happen in each of these situations?"

You might hear them say, "Oh, that can never happen!" As a test analyst, you know what that means. Your job now is to figure out how to make it happen.

You might hear them say, "Oh, well, I'd never thought of that." That probably means you just prevented a bug from ever happening, if you are doing test design during system design.

Browsing Selecting Logging in Purchasing Confirmed Left		Click link Add to cart Continue shopping Check out Login[bad] Login[good] Purchase[bad] Purchase[good] Abandon Resume shopping Go elsewhere		<table border="1"> <thead> <tr> <th>Current State</th> <th>Event/cond</th> <th>Action</th> <th>New State</th> </tr> </thead> <tbody> <tr><td>Browsing</td><td>Click link</td><td>Display</td><td>Browsing</td></tr> <tr><td>Browsing</td><td>Add to cart</td><td>Selection dia</td><td>Selecting</td></tr> <tr><td>Browsing</td><td>Continue shopping</td><td>Undefined</td><td>Undefined</td></tr> <tr><td>Browsing</td><td>Check out</td><td>Undefined</td><td>Undefined</td></tr> <tr><td>Browsing</td><td>Login[bad]</td><td>Undefined</td><td>Undefined</td></tr> <tr><td>Browsing</td><td>Login[good]</td><td>Undefined</td><td>Undefined</td></tr> <tr><td>Browsing</td><td>Purchase[bad]</td><td>Undefined</td><td>Undefined</td></tr> <tr><td>Browsing</td><td>Purchase[good]</td><td>Undefined</td><td>Undefined</td></tr> <tr><td>Browsing</td><td>Abandon</td><td>&lt;no action&gt;</td><td>Left</td></tr> <tr><td>Browsing</td><td>Resume shopping</td><td>Undefined</td><td>Undefined</td></tr> <tr><td>Browsing</td><td>Go elsewhere</td><td>Undefined</td><td>Undefined</td></tr> <tr><td>Selecting</td><td>Click link</td><td>Undefined</td><td>Undefined</td></tr> <tr><td>Left</td><td>Go elsewhere</td><td>Undefined</td><td>Undefined</td></tr> </tbody> </table> <p><i>{Fifty-three rows, generated in the pattern shown above, not shown}</i></p>	Current State	Event/cond	Action	New State	Browsing	Click link	Display	Browsing	Browsing	Add to cart	Selection dia	Selecting	Browsing	Continue shopping	Undefined	Undefined	Browsing	Check out	Undefined	Undefined	Browsing	Login[bad]	Undefined	Undefined	Browsing	Login[good]	Undefined	Undefined	Browsing	Purchase[bad]	Undefined	Undefined	Browsing	Purchase[good]	Undefined	Undefined	Browsing	Abandon	<no action>	Left	Browsing	Resume shopping	Undefined	Undefined	Browsing	Go elsewhere	Undefined	Undefined	Selecting	Click link	Undefined	Undefined	Left	Go elsewhere	Undefined	Undefined
Current State	Event/cond	Action	New State																																																									
Browsing	Click link	Display	Browsing																																																									
Browsing	Add to cart	Selection dia	Selecting																																																									
Browsing	Continue shopping	Undefined	Undefined																																																									
Browsing	Check out	Undefined	Undefined																																																									
Browsing	Login[bad]	Undefined	Undefined																																																									
Browsing	Login[good]	Undefined	Undefined																																																									
Browsing	Purchase[bad]	Undefined	Undefined																																																									
Browsing	Purchase[good]	Undefined	Undefined																																																									
Browsing	Abandon	<no action>	Left																																																									
Browsing	Resume shopping	Undefined	Undefined																																																									
Browsing	Go elsewhere	Undefined	Undefined																																																									
Selecting	Click link	Undefined	Undefined																																																									
Left	Go elsewhere	Undefined	Undefined																																																									

**Figure 5: State Transition Table Example**

Figure 5 shows an excerpt of the table we would create for the ecommerce example we've been looking at so far. We have six states:

- . Browsing
- . Selecting
- . Logging in
- . Purchasing
- . Confirmed
- . Left

We have eleven event/condition combinations:

- Click link
- Add to cart
- Continue shopping
- Check out
- Login[bad]
- Login[good]
- Purchase[bad]
- Purchase[good]
- Abandon
- Resume shopping
- Go elsewhere

That means our complete state transition table would have sixty-six rows, one for each possible pairing of a specific state with a specific event/condition combination.

To derive a set of tests that covers the state transition table, we can follow the procedure shown in the steps below. Notice that we build on an existing set of tests created from the state transition diagram to achieve state/transition or 0-switch cover

1. Start with a set of tests (including the starting and stopping state rule), derived from a state transition diagram, that achieves state/transition covered.
2. Construct the state transition table and confirm that the tests cover all the defined rows. If they do not, then either you didn't generate the existing set of tests properly, or you didn't generate the table properly, or the state transition diagram is screwed up. Do not proceed until you have identified and resolved the problem, including re-creating the state transition table or the set of tests, if necessary.
3. Select a test that visits a state for which one or more undefined rows exists in the table. Modify that test to attempt to introduce the undefined event/condition combination for that state. Notice that the action in this case is undefined.
4. As you modify the tests, mark the row as covered. The easiest way to do this is to take a printed version of the table and use a marker to highlight each row as covered.
5. Repeat steps 3 and 4 until all rows have been covered.

Again, this procedure will generate logical test cases.

- Existing test
  - (browsing, add to cart, selection dialog, checkout, login dialog, login[good], purchase dialog, abandon, <no action>, left)
- Modified tests (to cover undefined browsing event/ conditions only)
  - (browsing, *attempt: continue shopping, action undefined*, add to cart, selection dialog, checkout, login dialog, login[good], purchase dialog, abandon, <no action>, left)
  - (browsing, *attempt: check out, action undefined*, add to cart, selection dialog, checkout, login dialog, login[good], purchase dialog, abandon, <no action>, left)
  - There are six other modified tests for browsing, not shown...
- Don't try to cover undefined event/ conditions combinations for more than one state in any test, because you don't know whether the system will remain testable!
- Best case scenario is that the undefined event/ condition combination is ignored or rejected with an intelligent error message, and processing continues normally from there

## Figure 6: Deriving Tests Example

Figure 6 shows an example of deriving table-based tests, building on the e-commerce example already shown. At the top, you can see that I've selected an existing test from the larger set of tests derived before.

(browsing, add to cart, selection dialog, checkout, login dialog, login[good], purchase dialog, abandon, <no action>, left)

Now, from here I started to create modified tests to cover undefined browsing event/conditions, and those undefined conditions only.

One test is: (browsing, *attempt: continue shopping, action undefined*, add to cart, selection dialog, checkout, login dialog, login[good], purchase dialog, abandon, <no action>, left)

Another test is: (browsing, *attempt: check out, action undefined*, add to cart, selection dialog, checkout, login dialog, login[good], purchase dialog, abandon, <no action>, left)

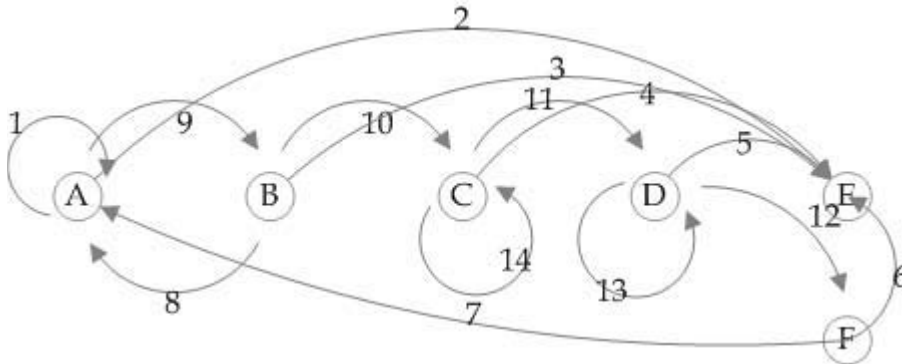
There are six other modified tests for browsing, which I've not shown. As you can see, it's a mechanical process to generate these tests. As long as you are careful to keep track of which rows you've covered – using the marker trick I mentioned earlier, for example – it's almost impossible to forget a test.

Now, you'll notice that I only included one undefined event/condition combination in each test step. Why? This is a variant of the equivalence partitioning rule that we should not create invalid test cases that combine multiple invalids. In this case, each row corresponds to an invalid. If we try to cover two rows in a single test step, we can't be sure the system will remain testable after the first invalid.

Notice that I indicated that the action is undefined. What is the ideal system behavior under these conditions? Well, the best-case scenario is that the undefined event/condition combination is ignored or – better yet – rejected with an intelligent error message. At that point, processing continues normally from there. In the absence of any meaningful input from business analysts, the requirements specification, system designers, or any other authority, I would take the position that any other outcome is a bug, including some inscrutable error message like, "What just happened can't happen." (No, I'm not making that up. An RBCS course attendee once told me she had seen exactly that message when inputting an unexpected value.)

## Switch Coverage

Figure 7 and Figure 8 show how we can generate sequences of transitions, using the concept of switch coverage. I'm going to illustrate this concept with the e-commerce example we've used so far.



**Figure 7:**  
**Notationally**  
**Compressed State**  
**Diagram**

In Figure 7, you see the same state transition as before. Except, I have

replaced the state labels with letters, and the transition labels with numbers. Now, a state/transition pair can be specified as a letter followed by a number. Notice that I'm not bothering to list, in the table in Figure 8, a letter after the number, because it's unambiguous from the diagram what state we'll be in after the given transition. There is only one arrow labeled with a given number that leads out of a state labeled with a given letter, and that arrow lands on exactly one state.

The table contains two types of columns. The first is the state/transition pairs that we must cover to achieve 0-switch coverage. Study this for a moment, and assure yourself that, by designing tests that cover each state/transition pair in the 0-switch columns, you'll achieve state/transition coverage as

0-switch			1-switch								
A1	A2	A9	A1A1	A1A2	A1A9				A9B10	A9B8	A9B3
B10	B8	B3	B10C14	B10C11	B10C4	B8A1	B8A2	B8A9			
C14	C11	C4	C14C14	C14C11	C14C4	C11D13	C11D12	C11D5			
D13	D12	D5	D13D13	D13D12	D13D5	D12F6	D12F7				
F6	F7					F7A1	F7A2	F7A9			

discussed previously.

### Figure 8: N-1 Switch Coverage Example

Constructing the 0-switch columns is easy. The first row consists of the first state, with a column for each transition leaving that state. There are at most three transitions from the A state. Repeat that process for each state for which there is an outbound transition. Notice that the E state doesn't have a row, because E is a final state and there's not outbound transition. Notice also that, for this example, there are at most three transitions from any given state.

The 1-switch columns are a little trickier to construct, but there's a regularity here that makes it mechanical if you are meticulous. Notice, again, that after each transition occurs in the 0-switch situation, we are left in a state, which is implicit in the 0-switch cells. As mentioned above, there are at most three transitions from any given state. So, that means that, for this example, each 0-switch cell can expand to at most three 1-switch cells.

So, we can take each 0-switch cell for the A row and copy it into three cells in the 1-switch columns, for nine cells for the A row. Now, we ask ourselves, for each triple of cells in the A row of the 1-switch columns, what implicit state did we end up in? We can then refer to the appropriate 0-switch cells to populate the remainder of the 1-switch cell.

Notice that the blank cells in the 1-switch columns indicate situations where we entered a state in the first transition from which there was no outbound transition. In this example, that is the state labeled "E" in Figure 7, which was labeled "Left" on the full-sized diagram.

So, given a set of state/transition sequences like those shown – whether 0-switch, 1-switch, 2-switch, or even higher – how do we derive test cases to cover those sequences and achieve the desired level of coverage? Again, I’m going to build on an existing set of tests created from the state transition diagram to achieve state/transition or 0-switch cover.

1. Start with a set of tests (including the starting and stopping state rule), derived from a state transition diagram, that achieves state/transition coverage
2. Construct the switch table using the technique shown previously. Once you have, confirm that the tests cover all of the cells in the 0-switch columns. If they do not, then either you didn’t generate the existing set of tests properly, or you didn’t generate the switch table properly, or the state transition diagram is screwed up. Do not proceed until you have identified and resolved the problem, including re-creating the switch table or the set of tests, if necessary. Once you have that done, check for higher-order switches already covered by the tests.
3. Now, using 0-switch sequences as needed, construct a test that reaches a state from which an uncovered higher-order switch sequence originates. Include that switch sequence in the test. Check to see what state this left you in. Ideally, another uncovered higher-order switch sequence originates from this state, but, if not, see if you can use 0-switch sequences to reach such a state. You’re crawling around in the state transition diagram looking for ways to cover higher-order sequence. Repeat this for the current test until the test must terminate.
4. As you construct tests, mark the switch sequences as covered once you include them in a test. The easiest way to do this is to take a printed version of the switch table and use a marker to highlight each cell as covered.
5. Repeat steps 3 and 4 until all switch sequences have been covered.

Again, this procedure will generate logical test cases.

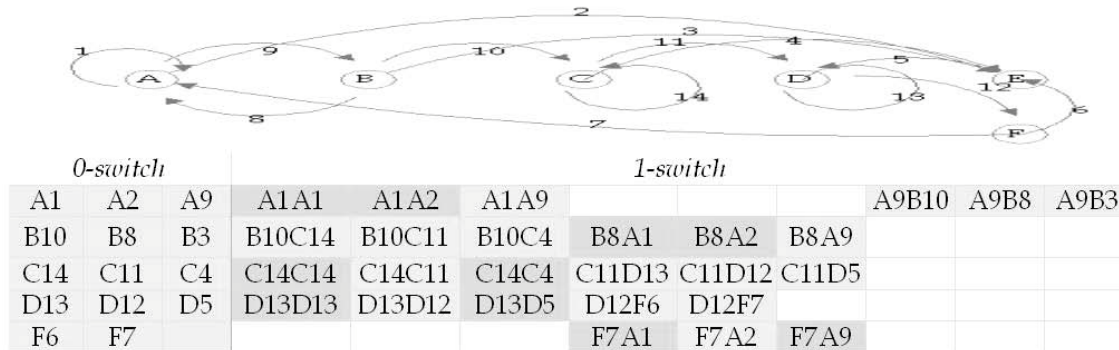
In Figure 8, we see the application of the derivation technique covered previously to the e-commerce example we’ve used. After finishing the second step, that of assessing coverage already attained via 0-switch coverage, we can see that most of the table is already shaded. Those are the light gray shaded cells, which are covered by the five existing state/transition cover tests.

Now, we generate five new tests to achieve 1-switch cover. Those are shown in Figure 9. The dark gray shaded cells are covered by five new 1-switch cover tests.

- . (A1A1A2).
- . (A9B8A1A9B8A2).
- . (A9B10C14C14C4).
- . (A9B10C11D13D13D5).

□. (A9B10C11D12F7A1A9B10C11D12F7A9).

Let me mention something about this algorithm for deriving higher-order switch coverage tests, as well as the one given previously for row-coverage tests. Both build on an existing set of tests that achieve state/transition coverage. That is efficient from a test design point of view. It's also conservative from a test execution point of view, because we cover the less challenging stuff first, then move on to the more difficult tests.



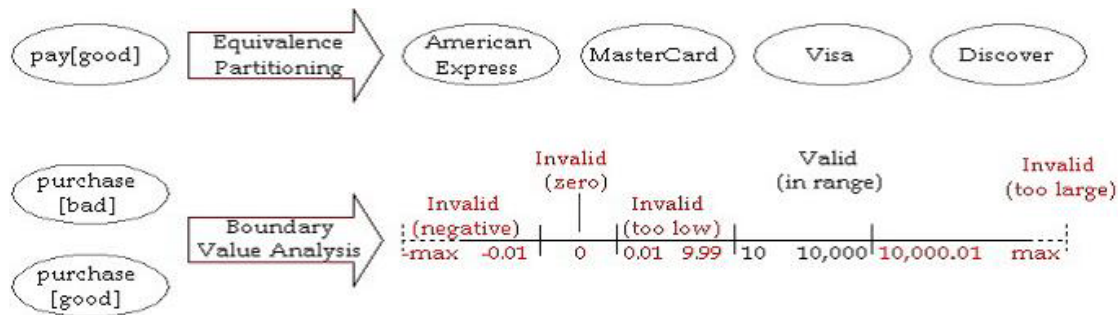
**Figure 9: Deriving Tests Example**

However, it is quite possible that, starting from scratch, a smaller set of tests could be build, both for the row coverage situation and for the 1-switch coverage situation. If the most important thing is to create the minimum number of tests, then you should look for ways to reduce the tests created, or modify the derivation procedures given here to start from scratch rather than to build on an existing set of 0-switch tests.

### State Testing with Other Techniques

Let's finish our discussion on state-based testing by looking at a couple interesting questions. First, how might equivalence partitioning and boundary value analysis combine with state-based testing? The answer is, quite well.

From the e-commerce example, suppose that the minimum purchase is \$10 and the maximum is \$10,000. In that case, we can perform boundary value analysis on the purchase[good] and purchase[bad] event/condition combinations as shown in Figure 10. By covering not only transitions, or rows, or transitions sequence, but also boundary values, this forces us to try different purchase amounts.



**Figure 10: Equivalence Partitions and Boundary Values**

We can also apply equivalence partitioning to the `pay[good]` event/condition combination. For example, suppose we accept four different types of credit cards. By covering not only transitions, or rows, or transition sequences, but also equivalence partitions, this forces us to try different payment types.

Now, to come full circle on a question I brought up in the previous article. When do we use decision tables and when do we use state diagrams?

This can be, in some cases, a matter of taste. The decision table is easy to use and compact. If we're not too worried about the higher-order coverage, or the effect of states on the tests, we can model many state-influenced situations as decision tables, using conditions to model states. However, if the decision table's conditions section starts to become very long, you're probably stretching the technique. Also, keep in mind that test coverage is usually more thorough using state-based techniques. In most cases, one technique or the other will clearly fit better. If you are at a loss, try both and see which feels most appropriate.

## Conclusion

In this article, I've shown how to apply state transition diagram, state transition tables, and switch coverage analysis to the testing of sophisticated and complex system, especially embedded systems and other systems where the system response to a given set of inputs depends on what the system has dealt with in the past. We have looked at decision tables as a way to test detailed business rules, and now state-based methods test state-dependent systems. However, we will need to look at an additional technique, use cases, to deal with the full range of internal business logic testing we need to do. I'll address that technique in the next article in this series.

## Author Bio

With a quarter-century of software and systems engineering experience, Rex Black is President of RBCS ([www.rbc-us.com](http://www.rbc-us.com)), a leader in software, hardware, and systems testing. For over a dozen years, RBCS has delivered services in consulting, outsourcing and training for software and hardware testing. Employing the industry's most experienced and recognized consultants, RBCS conducts product testing, builds and improves testing groups and hires testing staff for hundreds of clients worldwide. Ranging from Fortune 20 companies to start-ups, RBCS clients save time and money

through improved product development, decreased tech support calls, improved corporate reputation and more. As the leader of RBCS, Rex is the most prolific author practicing in the field of software testing today. His popular first book, *Managing the Testing Process*, has sold over 40,000 copies around the world, including Japanese, Chinese, and Indian releases, and is now in its third edition. His five other books on testing, *Advanced Software Testing: Volume I*, *Advanced Software Testing: Volume II*, *Critical Testing Processes*, *Foundations of Software Testing*, and *Pragmatic Software Testing*, have also sold tens of thousands of copies, including Hebrew, Indian, Chinese, Japanese and Russian editions. He has written over thirty articles, presented hundreds of papers, workshops, and seminars, and given about thirty keynote speeches at conferences and events around the world. Rex is the former President of the International Software Testing Qualifications Board and the American Software Testing Qualifications Board.