

Testing State Machine with AdaTEST and CANTATA

Executive Summary

This paper describes how AdaTEST and Cantata can be used to test software which implements state machines. Topics addressed include test case design, coverage measures, test implementation, and automated coverage analysis.

IPL is an independent software house founded in 1979 and based in Bath. IPL was accredited to ISO9001 in 1988, and gained TickIT accreditation in 1991. Both Cantata and AdaTEST have been produced to these standards.

Copyright

This document is the copyright of IPL Information Processing Ltd. It may not be copied or distributed in any form, in whole or in part, without the prior written consent of IPL.

*IPL
Eveleigh House
Grove Street
Bath
BA1 5LR
UK*

*Phone: +44 (0) 1225 444888
Fax: +44 (0) 1225 444400
email ipl@iplbath.com*



Certificate Number FM 01589

*Last Update: 03/07/1997 08:36:00
File: TSTATE.DOC*

1. Introduction

The use of state machines is a common approach to the implementation of a wide range of computer applications, used in just about every area of the software industry. State machines are particularly popular with designers of telecommunications systems, communications protocols and control systems.

This paper discusses some particular issues concerned with the testing of state machines. It looks at testing strategies for state machines, describes coverage measures appropriate to testing state machines, and describes how AdaTEST and Cantata can be used to test software which implements state machines.

Some readers of this paper will be new to the subject of formalised software testing. It is intended that this paper will help such readers to design and implement tests for state machines. Other readers may already have considerable experience of designing, implementing and testing state machines, nevertheless, the application of AdaTEST and Cantata should be of interest to them.

The widespread use of state machines has resulted in an equally diverse set of terminology. The terminology used in this paper is defined in Section 2. Section 3 discusses test strategies and measures. Test implementation using AdaTEST and Cantata is described in section 4. Implementation of associated coverage measures follows in section 5. The application of assertions, a feature introduced by AdaTEST and Cantata version 3.0 is described in section 6.

2. Definitions

Associated with the widespread use of state machines is an equally diverse set of terminology. For the purposes of this paper, the function of a state machine can be summarised by the following statement:

A state machine will exist in a **current state**. When an **event** occurs, the state machine may take an **action** and may make a **transition** to a **new state**.

A range of representations exist for modelling the behaviour of a state machine in the design of the software. These include State-Transition Diagrams, State Graphs, State-Transition Matrices, and State Tables. The generic term **state model** will be used in this paper.

The state of a state machine will normally be represented in software by a **state variable**, represented by a discrete data type.

State A state machine has a fixed set of possible states. The properties and behaviour of the state machine are identical whenever it is in a particular state.

Event An event is caused by inputs to a state machine. In response to an event, a state machine may take an action and make a transition to a new state. In any particular state, some events will cause associated transitions to new states, whilst other events will not cause transitions.

Action Associated with a particular state and event may be an action. An action may include a transition to a new state, but may also result in an output from the state machine.

Transition A transition is triggered by an event occurring in a particular state. In response to the event, a transition is made from one state of a state machine (the **current** state before the transition), to another state of a state machine (the **new** state after a transition).

3. Test Strategies and Measures

Tests for a state machine can be designed from the functional specification of the state machine, independent of the structural design of the actual implementation. But what tests should be designed, and why?

An obvious starting point is to test every transition. That is, to design test cases to set a current state, create the circumstances which lead to an event, to observe the action taken, the transition made, and the new state. As there is a defined set of transitions in the state model, a coverage measure associated with this strategy is to measure the proportion of transitions exercised by a set of test cases.

Transition Coverage =
$$\frac{\text{Number of transitions exercised}}{\text{Total number of transitions in the state model}}$$

This measure is often referred to as **0-Switch Coverage**, as described by the paper "Testing Software Design Modelled by Finite State Machines", T S Chow, IEEE Transactions on Software Engineering, 4 1978.

Developments of this strategy are to test sequences of two transitions, three transitions, n transitions etc. For example:

2-Transition Coverage =
$$\frac{\text{Number of sequences of two transitions exercised}}{\text{Total number of sequences of two transitions in the state model}}$$

As the number of transitions exercised by each test case increases, the number of possible permutations and combinations of transitions within a sequence also increases, consequently increasing the number of test cases required to achieve the associated coverage metrics. Increased complexity consequently makes achievement of coverage for long sequences of transitions impractical.

Coverage measures for sequences of transitions are also known respectively as 1-Switch Coverage, 2-Switch Coverage, through to [n-1]-Switch Coverage, where n is the number of transitions in the sequence.

As a minimum, each test case should be specified as:

- An initial "current state";
- Inputs to the software (to cause an event);
- The expected action (if any), including a transition (if any) and outputs (if any);
- The resulting "next state".

This gets more complicated when test cases are designed to execute sequences of two or more transitions. For such test cases, the specification of the test case should include the sequence of events, actions, transitions and states involved in the test case.

Any strategy aimed at testing specified transitions is biased towards **positive testing**. That is, test cases are designed to exercise that the software does what it is supposed to do. A thorough test should also include **negative testing**, to verify that the software does not do things that it is not supposed to do.

Within existing positive test cases, checks should be incorporated to ensure that there have been no unwanted side affects. Furthermore, additional test cases should be designed to ensure that invalid actions and transitions cannot be induced.

Example 3.1:

Consider a state machine which exists in one of three states: {A, B, C}, as shown in figure 3.1.

A number of valid transitions are defined: {A-B, B-C, C-A, C-B}, from A to B, from B to C, from C to A, and from C to B respectively. Note that there are no transitions from B to A, or from A to C.

The state machine responds to four events: {W, X, Y, Z}. Event W causes transition A-B, event X causes transition B-C, event Y causes transition C-A, and event Z causes transition C-B.

Actions other than transitions are immaterial to this example. Events occurring in other states are ignored.

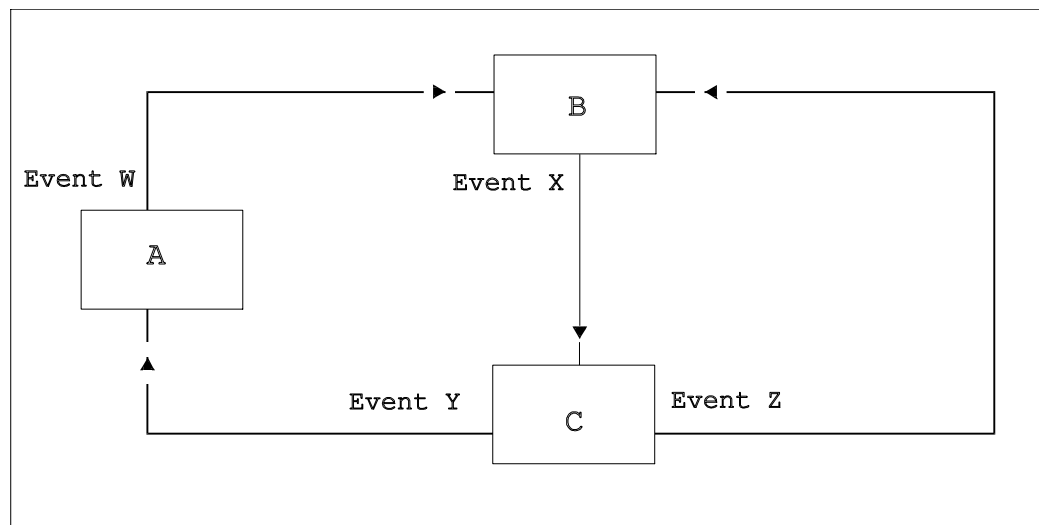


Figure 3.1 - Example State Machine

100% transition coverage could be achieved by four test cases:

<u>Test Case</u>	<u>Current State</u>	<u>Event</u>	<u>Transition</u>	<u>New State</u>
1	A	W	A-B	B
2	B	X	B-C	C
3	C	Y	C-A	A
4	C	Z	C-B	B

However, what happens if an event other than W occurs when the current state is A? It should be ignored, but we have not verified this with a test case. Can the illegal transition A-C be induced? Conscientious negative testing requires test cases to be designed to verify the behaviour of the software under such circumstances, to ensure that the state machine is correctly implemented.

A strategy for ensuring that negative tests are designed to verify freedom from such bugs is to design test cases which subject each state of the state machine to each event in the total set of events, not just the legal events for that state.

$$\text{State-Event Coverage} = \frac{\text{Number of state-event pairs exercised}}{\text{Number of states} * \text{Number of events}}$$

To achieve 100% State-Event Coverage in the above example would therefore require a minimum of twelve test cases (3 states, with 4 events to be tested in each state).

If the state machine were *fully specified*, such that in each state, for each event, a unique transition was defined (in some cases, null transitions), then Transition Coverage or 0-Switch Coverage would be the same as State-Event Coverage. In practice, it is common for state machines to not be fully specified.

4. Test Implementation with AdaTEST and Cantata

An essential prerequisite for thorough testing of state machines is providing visibility of the state variable to the test environment. Designers will often deliberately hide a state variable within the software which implements a state machine, where it is not directly visible to software outside of the state machine. For example, a state variable may be declared within the body of an Ada package.

The alternative, of assuming the current state based on test case history and outputs from the software, makes it difficult if not impossible to thoroughly test the software, and is likely to result in purely superficial testing. To test a state machine thoroughly, the state variable must be visible to the test environment.

Options available for providing the test environment with visibility include:

- Designing software so that the state variables are visible.
- Building in test points to provide visibility of hidden data, including state variables. (As described in the IPL paper "Achieving Testability when Using Ada Packaging and Data Hiding Methods").
- Designing State_Set and State_Enquiry functions into the software.
- Using AdaTEST or Cantata stubs to simulate calls made by the state machine software, with visibility of the state variable being inherited by the stubs.
- AdaTEST or Cantata assertions can be used to verify the contents of a state variable. (See section 6 of this paper).

The tried and tested approach is to use test points to provide access to hidden data. Having achieved visibility of the state variable, the state variable must be placed into the initial "current state" required by each test case. This can be achieved in one of three ways:

- (a) Designing test cases to follow one another, so that the "next state" of one test becomes the "current state" of the following test.
- (b) Manoeuvring the state variable to the desired 'current state' by applying a sequence of events.
- (c) Assigning the prescribed "current state" to the state variable.

Although methods (a) and (b) are intuitively more attractive, they can both lead to long chains of secondary failures within a test script. An incorrect next state following one test case can invalidate all subsequent tests cases.

It may also be possible to become stuck in dead ends, or for there to be mutually exclusive parts of the state machine which cannot be fully tested with methods (a) and (b). Finally, depending solely on methods (a) and (b) can lead to a high maintenance overhead.

In practice, a compromise is necessary. Small groups of test cases can use methods (a) or (b), with direct assignment to the state variable (method (c)) being used to provide robustness and maintainability between each group. Test cases then proceed by making the applicable inputs to the software under test to cause an event, capturing and checking outputs from the software under test, and checking the subsequent value of the state variable against the expected "next state".

Such a test implementation should be considered a minimum. Two criteria used in specifying a test case have not been directly verified by this test implementation:

- It has not directly verified which event the inputs caused;
- It has not directly verified which transition occurred.

In order to verify that the correct event was processed and the correct transition occurred, we need to keep a record of which event was encountered and which transition occurred, so that they can be checked from the test script.

The simple solution is to design the software which implements the state machine so that it stores the event encountered and the transition made as static data, in the same way that the state variable is maintained. The test script can then check the event variable, the transition variable and the state variable following each test case.

There may be test cases where a sequence of inputs are designed to exercise a sequence of transitions. A good test script should therefore verify each intermediate state within such a sequence, together with the associated events and transitions. On any such sequence of transitions, checks should be made of the event variable, transition variable and state variable following each individual input to the state machine software.

The test implementation could still miss transient states, where the a state machine implies an additional event from a transition made in response to an initial event. In such a state machine, a single input could cause a chain of transitions to occur before control is returned to the test script. Solutions to this problem rely on the designer of the state machine software considering testability from the outset (which is what good designers should do anyway). Careful design of the boundaries between routines within the software can be used to achieve a design in which transient states are fully testable.

5. Measuring Coverage with AdaTEST and Cantata

In the previous section of this paper, methods of implementing test cases and verifying the outcome of each test case have been described. However, overall test coverage still has to be measured. Two details need to be known:

- What is the complete set of coverage items?
- Which of these have been executed?

The algorithm used to implement structural coverage metrics such as STATEMENT_COVERAGE and DECISION_COVERAGE within AdaTEST and Cantata can be adapted for other metrics. The generic algorithm used by IPL is:

- Build a score board representing the complete set of coverage items;
- As each coverage item is executed by a test case, the corresponding score is incremented;
- Coverage is calculated from the number of non-zero scores divided by the number of slots in the score board.

This algorithm could be simplified by using an array of Boolean values, but the counting score board also enables detailed reports to be generated, such as the STATEMENT_STATISTICS and DECISION_STATISTICS reports provided by AdaTEST and Cantata.

Users of AdaTEST and Cantata can add code to a test script to implement State-Event coverage based on this algorithm:

- Declare a two dimensional array of scores, with one axis being indexed by the state, and the other axis being indexed by the event. Initialise all cells to zero;
- Within each test case, write statements to increment the corresponding cell of the array according to the current state and event (not the next state);
- At the end of the test script, calculate the proportion of non-zero cells in the array. Use the CHECK_ANALYSIS command to check the calculated coverage against the objective level.

Example 5.1

This example continues with the state machine from example 3.1. A scoreboard array for the state machine, with scores accumulated following execution of test cases 1, 2, 3 and 4, can be visualised as:

		<u>Event</u>			
		<u>W</u>	<u>X</u>	<u>Y</u>	<u>Z</u>
<u>State</u>	<u>A</u>	1	0	0	0
	<u>B</u>	0	1	0	0
	<u>C</u>	0	0	1	1

*Applying the formula, State-Event Coverage achieved is $4/(3*4) = 33\%$.*

A similar approach can be used to measure transition coverage, using a single dimensional array indexed by an enumeration variable identifying the set of valid transitions.

Extensions of the technique to measure sequences of more than one transition leads to problems in defining the complete set of coverage items. However, coverage of sequences of more than one event could be measured using a multi dimensional score board.

Example5.2

This example continues with the state machine from example 5.1.

Suppose we wish to determine coverage of sequences of two events from each state. For lack of a better term, this could be called State-2-Event Coverage.

*The score board would be a three dimensional array of state {A, B, C}, event {W, X, Y, Z} and second event {W, X, Y, Z}. There would be $3*4*4 = 48$ cells in the score board.*

The size of the scoreboard and the number of test cases required to achieve 100% coverage increases exponentially.

6. AdaTEST and Cantata Assertions

AdaTEST and Cantata assertions can be used to simplify the testing of state machines. To verify State-Event Coverage, assertions can be used to ensure that all events have been exercised in all states.

Example 6.1:

Consider the state machine described in example 3.1. This could be implemented as an Ada case or C switch statement:

```
get (EVENT);
switch (STATE)
{
  case A:
    process_state_A(EVENT);
    break;
  case B:
    process_state_B(EVENT);
    break;
  case C:
    process_state_C(EVENT);
    break;
}
```

By inserting a set of assertions before the event processing for each state, coverage of all events for all states can be verified:

```

get (EVENT);
switch (STATE)
{
  case A:
    /* _cth_assert_always (STATE=A) */
    /* _cth_assert_once (EVENT=W) */
    /* _cth_assert_once (EVENT=X) */
    /* _cth_assert_once (EVENT=Y) */
    /* _cth_assert_once (EVENT=Z) */
    process_state_A (EVENT);
    break;
  case B:
    /* _cth_assert_always (STATE=B) */
    /* _cth_assert_once (EVENT=W) */
    /* _cth_assert_once (EVENT=X) */
    /* _cth_assert_once (EVENT=Y) */
    /* _cth_assert_once (EVENT=Z) */
    process_state_B (EVENT);
    break;
  case C:
    etc.

```

Note that assertions have also been used to verify the value of the state variable.

Similar uses of assertions can also be employed to verify that all specified transitions have been tested, especially where there is more than one transition available between a given pair of states.

7. Conclusion

This paper has described how AdaTEST and Cantata can be used to test state machine software.

State machine software is particularly suited to functional testing. Definitive sets of test cases can be designed from a functional specification of a state machine and practical metrics exist to measure the completeness of sets of test cases.

In addition to the usual structural coverage metrics provided by AdaTEST and Cantata, functional test coverage metrics such as Transition Coverage and State-Event Coverage can be automated. Such automation requires some relatively straight forward programming work by the AdaTEST or Cantata user.

A comprehensive testing strategy for state machine software should set targets for both structural coverage measures (as provided by AdaTEST and Cantata) and functional coverage measures as described in this paper.