<ch10 toc ch12>

Chapter 11
STATES, STATE GRAPHS, AND TRANSITION TESTING

1. SYNOPSIS

The **state graph** and its associated **state table** are useful models for describing software behavior. The **finite-state machine** is a functional testing tool and testable design programming tool. Methods analogous to path testing are described and discussed.

2. MOTIVATIONAL OVERVIEW

The **finite-state machine** is as fundamental to software engineering as boolean algebra. State testing strategies are based on the use of finite-state machine models for software structure, software behavior, or specifications of software behavior. Finite-state machines can also be implemented as table-driven software, in which case they are a powerful design option. Independent testers are likeliest to use a finite-state machine model as a guide to the design of functional tests—especially system tests. Software designers are likelier to want to exploit and test finite-state machine software implementations. Finally, finite-state machine models of software abound in the testing literature, much of which will be meaningless to readers who don't know this subject. Among the affected testing topics are protocols, concurrent systems, system failure and recovery, system configuration, and distributed data bases (BARN72, DAVI88A, HOLZ87, PETE76).

3. STATE GRAPHS

3.1. States

The word **"state"** is used in much the same way it's used in ordinary English, as in "state of the union," or "state of health." The Oxford English Dictionary defines "state" as: "A combination of circumstances or attributes belonging for the time being to a person or thing."

A program that detects the character sequence "ZCZC" can be in the following states:

1. Neither ZCZC nor any part of it has been detected.
2. Z has been detected.
3. ZC has been detected.
4. ZCZ has been detected.
5. ZCZC has been detected.

A moving automobile whose engine is running can have the following states with respect to its transmission:

1. Reverse gear
2. Neutral gear
3. First gear
4. Second gear
5. Third gear
6. Fourth gear

A person's checkbook can have the following states with respect to the bank balance:

1. Equal
2. Less than
3. Greater than

A word processing program menu can be in the following states with respect to file manipulation:

1. Copy document
2. Delete document
3. Rename document
4. Create document
5. Compress document
6. Copy disc
7. Format disc
8. Backup disc
9. Recover from backup

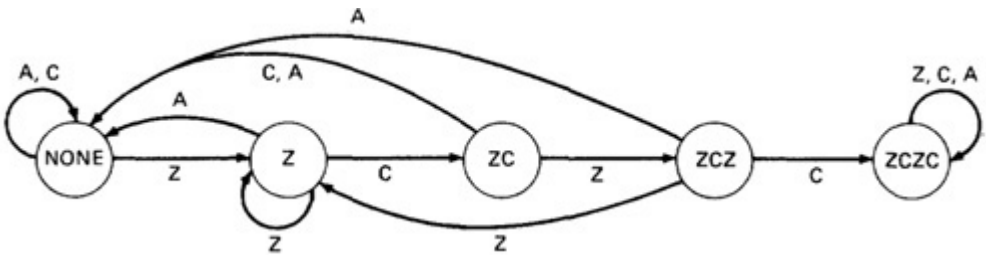


Figure 11.1. One-Time ZCZC Sequence-Detector State Graph.

States are represented by **nodes.** States are numbered or may be identified by words or whatever else is convenient. Figure 11.1 shows a typical **state graph.** The automobile example is really more complicated because: (1) the engine might or might not be running, (2) the car itself might be moving forward or backward or be stopped, and (3) the clutch might or might not be depressed. These factors multiply the above six states by 2 × 3 × 2 = 12, for a total of 72 rather than 6 states. Each additional factor that has alternatives multiplies the number of states in a model by the number of alternatives. The number of states of a computer is 2 raised to the power of the number of bits in the computer; that is, all the bits in main memory, registers, discs, tapes, and so on. Because most interesting factors are binary, and because each factor doubles the number of states, state graphs are most useful for relatively simple functional models involving at most a few dozen states and only a few factors.

3.2. Inputs and Transitions

Whatever is being modeled is subjected to inputs. As a result of those inputs, the state changes, or is said to have made a **transition.** Transitions are denoted by links that join the states. The input that causes the transition are marked on the link; that is, the inputs are link weights. There is one outlink from every state for every input. If several inputs in a state cause a transition to the same subsequent state, instead of drawing a bunch of parallel links we can abbreviate the notation by listing the several inputs as in: "input1, input2, input3. . .". A **finite-state machine** is an abstract device that can be represented by a state graph having a finite number of states and a finite number of transitions between states.

The ZCZC detection example can have the following kinds of inputs:

1. Z

2. C
3. Any character other than Z or C, which we'll denote by A

The state graph of Figure 11.1 is interpreted as follows:

1. If the system is in the "NONE" state, any input other than a Z will keep it in that state.
2. If a Z is received, the system transitions to the "Z" state.
3. If the system is in the "Z" state and a Z is received, it will remain in the "Z" state. If a C is received, it will go to the "ZC" state; if any other character is received, it will go back to the "NONE" state because the sequence has been broken.
4. A Z received in the "ZC" state progresses to the "ZCZ" state, but any other character breaks the sequence and causes a return to the "NONE" state.
5. A C received in the "ZCZ" state completes the sequence and the system enters the "ZCZC" state. A Z breaks the sequence and causes a transition back to the "Z" state; any other character causes a return to the "NONE" state.
6. The system stays in the "ZCZC" state no matter what is received.

As you can see, the state graph is a compact representation of all this verbiage.

3.3. Outputs

An output* can be associated with any link. Outputs are denoted by letters or words and are separated from inputs by a slash as follows: "input/output." As always, "output" denotes anything of interest that's observable and is not restricted to explicit outputs by devices. Outputs are also link weights. If every input associated with a transition causes the same output, then denote it as: "input 1, input 2, . . . input 3/output." If there are many different combinations of inputs and outputs, it's best to draw a separate parallel link for each output.

---

*"Output" rather than "outcome" because the outcome consists of the output *and* a transition to the new state. "Output" used in this context can mean almost anything observable and is not restricted to tangible outputs by devices, say.

Consider now, as an example, a simplified specification for a tape transport write-error recovery procedure, such as might be found in a tape driver routine:**

**Our objective here is not to design a tape driver but to illustrate how a specification, good or bad, sensible or not, can be modeled by a state graph.

"If no write errors are detected, (input = OK), no special action is taken (output = NONE). If a write error is detected (input = ERROR), backspace the tape one block and rewrite the block (output = REWRITE). If the rewrite is successful (input = OK), ignore the fact that there has been a rewrite. If the rewrite is not successful, try another backspace and rewrite. Return to the original state if and only if there have been two successive successful writes. If there have been two successive rewrites and a third error occurs, backspace ten centimeters and erase forward from that point (output = ERASE). If the erasure works (input = OK), return to the initial state. If it does not work, backspace another ten centimeters, erase and treat the next write attempt as for the first erasure. If the second erasure does not clear the problem, put the tape transport out of

service."

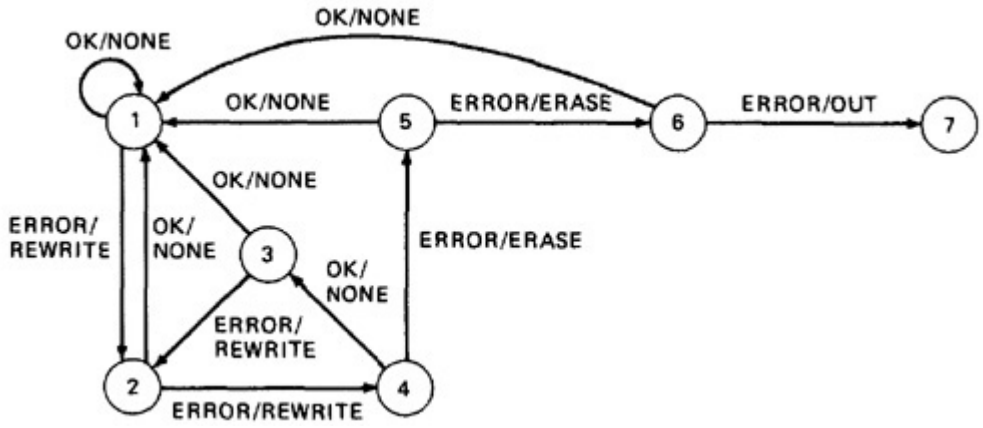


Figure 11.2. Tape Control Recovery Routine State Graph.

The state graph is shown in Figure 11.2. As in the previous example, the inputs and actions have been simplified. There are only two kinds of inputs (OK, ERROR) and four kinds of outputs (REWRITE, ERASE, NONE, OUT-OF-SERVICE). Don't confuse outputs with transitions or states. This can be confusing because sometimes the name of the output is the same as the name of a state.*Similarly, don't confuse the input with the state, as in the first transition and the second state of the ZCZC detection example.

---

*An alternate, but equivalent, representation of behavior, called a "Moore model" (MOOR56), associates outputs with states rather than with transitions. The model used in this book is called a "Mealy model" (MEAL55), in which outputs are associated with transitions. Mealy models are more useful because of the way software of this kind is usually implemented. Also, the Mealy model makes both inputs and outputs link weights, which makes it easier to use the methods of Chapter 12 for analysis.

3.4. State Tables

Big state graphs are cluttered and hard to follow. It's more convenient to represent the state graph as a table (the **state table** or **state-transition table**) that specifies the states, the inputs, the transitions, and the outputs. The following conventions are used:

	INPUT	
STATE	**OKAY**	**ERROR**
1	1/NONE	2/REWRITE
2	1/NONE	4/REWRITE
3	1/NONE	2/REWRITE
4	3/NONE	5/ERASE
5	1/NONE	6/ERASE
6	1/NONE	7/OUT
7	. . .	. . .

Table 11.1. State Table for Figure 11.2.

1. Each row of the table corresponds to a state.
2. Each column corresponds to an input condition.
3. The box at the intersection of a row and column specifies the next state (the transition) and the output, if any.

The state table for the tape control is shown in Table 11.1.

I didn't specify what happens in state 7 because it's not germane to the discussion. You would have to complete the state graph for that state and for all the other states (not shown) that would be needed to get the tape back into operation. Compare the tabular representation with the graphical representation so that you can follow the action in either notation.

3.5. Time Versus Sequence

State graphs don't represent time—they represent sequence. A transition might take microseconds or centuries; a system could be in one state for milliseconds and another for eons, or the other way around; the state graph would be the same because it has no notion of time. Although the finite-state machine model can be elaborated to include notions of time in addition to sequence, such as timed Petri nets (DAVI88A, MURA89, PETE81), the subject is beyond the scope of this book.

3.6. Software Implementation

3.6.1. Implementation and Operation

There is rarely a direct correspondence between programs and the behavior of a process described as a state graph. In the tape driver example, for instance, the inputs would occur over a period of time. The routine is probably activated by an executive, and the inputs might be status-return interrupts from the tape control hardware. Alternatively, the inputs might appear as status bits in a word in memory reserved for that transport. The tape control routine itself is probably reentrant, so it can be used simultaneously by all transports.

The state graph represents the total behavior consisting of the transport, the software, the executive, the status returns, interrupts, and so on. There is no simple correspondence between lines of code and states. The state table, however, forms the basis for a widely used implementation shown in the PDL program below. There are four tables involved:

1. A table or process that encodes the input values into a compact list (INPUT_CODE_TABLE).
2. A table that specifies the next state for every combination of state and input code (TRANSITION_TABLE).
3. A table or case statement that specifies the output or output code, if any, associated with every state-input combination (OUTPUT_TABLE).
4. A table that stores the present state of every device or process that uses the same state table—e.g., one entry per tape transport (DEVICE_TABLE).

The routine operates as follows, where # means concatenation:

```
BEGIN
PRESENT_STATE := DEVICE_TABLE(DEVICE_NAME)
ACCEPT INPUT_VALUE
INPUT_CODE := INPUT_CODE_TABLE(INPUT_VALUE)
POINTER := INPUT_CODE#PRESENT STATE
```

```
NEW_STATE := TRANSITION_TABLE(POINTER)
OUTPUT_CODE := OUTPUT_TABLE(POINTER)
CALL OUTPUT_HANDLER(OUTPUT_CODE)
DEVICE_TABLE(DEVICE_NAME) := NEW_STATE
END
```

1. The present state is fetched from memory.
2. The present input value is fetched. If it is already numerical, it can be used directly; otherwise, it may have to be encoded into a numerical value, say by use of a case statement, a table, or some other process.
3. The present state and the input code are combined (e.g., concatenated) to yield a pointer (row and column) of the transition table and its logical image (the output table).
4. The output table, either directly or via a case statement, contains a pointer to the routine to be executed (the output) for that state-input combination. The routine is invoked (possibly a trivial routine if no output is required).
5. The same pointer is used to fetch the new state value, which is then stored.

There could be a lot of code between the end of this flow and the start of a new pass. Typically, there would be a return to the executive, and the state-control routine would only be invoked upon an interrupt. Many variations are possible. Sometimes, no input encoding is required. In other situations, the invoked routine is itself a state-table-driven routine that uses a different table.

3.6.2. Input Encoding and Input Alphabet

Only the simplest finite-state machines, such as a character sequence detector in a compiler's lexical analyzer, can use the inputs directly. Typically, we're not interested in the actual input characters but in some attribute represented by the characters. For example, in the ZCZC detector, although there are 256 possible ASCII characters (including the inverse parity characters), we're only interested in three different types: "Z," "C," and "OTHER." The input encoding could be implemented as a table lookup in a table that contained the following codes: "OTHER" = 0, "Z" = 1 and "C" = 2. Alternatively, we could implement it as a process: IF INPUT = "Z" THEN CODE := 1 ELSE IF INPUT = "C" THEN CODE := 2 ELSE CODE := 0 ENDIF.

The alternative to input encoding is a huge state graph and table because there must be one outlink in every state for every possible different input. Input encoding compresses the cases and therefore the state graph. Another advantage of input encoding is that we can run the machine from a mixture of otherwise incompatible input events, such as characters, device response codes, thermostat settings, or gearshift lever positions. The set of different encoded input values is called the **input alphabet.** The word "input" as used in the context of finite-state machines always means a "character" from the input alphabet.

3.6.3. Output Encoding and Output Alphabet

There can be many different, incompatible, kinds of outputs for transitions of a finite-state machine: a single character output for a link is rare in actual applications. We might want to output a string of characters, call a subroutine, transfer control to a lower-level finite-state machine, or do nothing. Whatever we might want to do, there are only a finite number of such distinct actions, which we can encode into a convenient **output alphabet.** We then have a hypothetical (or real) output processor that invokes the action appropriate to the output code. Doing nothing is also considered an action and therefore requires its own code in the output alphabet. The word "output" as used in the context of finite-state machines means a "character" from the output alphabet.

3.6.4. State Codes and State-Symbol Products

We speak about finite-state machines as if the states are numbered by an integer. If there are *n* states and *k* different inputs, both numbered from zero, and the state code and input code are S and I respectively, then the pointer value is S*k* + I or I*n* + S depending on how you want to organize the tables. If the state machine processor is coded in an HOL then you can use a two-dimensional array and use two pointers (state code and input code); the multiplication will be done by object code. Finite-state machines are often used in time-critical applications because they have such fast response times. If a multiplication has to be done, the speed is seriously affected. A faster implementation is to use a binary number of states and a binary number of input codes, and to form the pointer by concatenating the state and input code. The speed advantage is obvious, but there are also some disadvantages. The table is no longer compact; that is, because the number of states and the number of input codes are unlikely to be both binary numbers, the resulting table must have holes in it. Like it or not, those holes correspond to state-input combinations and you have to fill them, if only with a call to an error recovery routine. The second disadvantage is size. Even in these days of cheap memory, excessive table size can be a problem, especially, for example, if the finite-state machine is part of embedded software in a ROM. For the above reasons, there may be another encoding of the combination of the state number and the input code into the pointer. The term **state-symbol product** is used to mean the value obtained by any scheme used to convert the combined state and input code into a pointer to a compact table without holes. This conversion could be done by multiplication and addition, by concatenation, or even by a hash-coding scheme for very big tables. When we talk about "states" and "state codes" in the context of finite-state machines, we mean the (possibly) hypothetical integer used to denote the state and not the actual form of the state code that could result from an encoding process. Similarly, "state-symbol product" means the hypothetical (or actual) concatenation used to combine the state and input codes.

3.6.5. Application Comments for Designers

An explicit state-table implementation is advantageous when either the control function is likely to change in the future or when the system has many similar, but slightly different, control functions. Their use in telecommunications, especially telephony, is common. This technique can provide fast response time—one pass through the above program can be done in ten to fifteen machine instruction execution times. It is not an effective technique for very small (four states or less) or big (256 states or more) state graphs. In the small case, the overhead required to implement the state-table software would exceed any time or space savings that one might hope to gain. In big state tables, the product of input values and states is big—in the thousands—and the memory required to store the tables becomes significant. The usual approach for big state graphs is to partition the problem into a hierarchy of finite-state machines. The output of the top level machine is a call to a subsidiary machine that processes the details. In telephony, for example, two-level tables are common and three- and four-level tables are not unusual.

3.6.6. Application Comments for Testers

Independent testers are not usually concerned with either implementation details or the economics of this approach but with how a state-table or state-graph representation of the behavior of a program or system can help us to design effective tests. If the programmers have implemented an explicit finite-state machine then much of our work has been done for us and we have to be concerned with the kinds of bugs that are inherent in the implementation—which is good reason for understanding such implementations. There is an interesting correlation, though: when a finite-state machine *model* is appropriate, so is a finite-state machine *implementation.* Sometimes, showing the programmers the kinds of tests developed from a state-graph description can lead them to consider it as an implementation technique.

4. GOOD STATE GRAPHS AND BAD

4.1. General

This is a book on testing so we deal not just with good state graphs, but also with bad ones. What constitutes a good or a bad state graph is to some extent biased by the kinds of state graphs that are likely to be used in a software test design context. Here are some principles for judging:

1. The total number of states is equal to the product of the possibilities of factors that make up the state.
2. For every state and input there is exactly one transition specified to exactly one, possibly the same, state.
3. For every transition there is one output action specified. That output could be trivial, but at least one output does something sensible.*

---

*State graphs without outputs can't do anything in the pragmatic world and can consequently be ignored. For output, include anything that could cause a subsequent action—perhaps setting only one bit.

---

4. For every state there is a sequence of inputs that will drive the system back to the same state.**

**In other words, we've restricted the state graphs to be strongly connected. This may seem overly narrow, because many state graphs are not strongly connected; but in a software context, the only nonstrongly connected state graphs are those used to set off bombs and other infernal machines or those that deal with bootstraps, initialization, loading, failure, recovery, and illogical, unrecoverable conditions. A state graph that is not strongly connected usually has bugs.

Figure 11.3 shows examples of improper state graphs.

A state graph must have at least two different input codes. With only one input code, there are only a few kinds of state graphs you can build: a bunch of disconnected individual states; disconnected strings of states that end in loops and variations thereof; or a strongly connected state graph in which all states are arranged in one grand loop. The latter can be implemented by a simple counter that resets at some fixed maximum value, so this elaborate modeling apparatus is not needed.

If I seem to have violated my own rules regarding outputs—I have. The ZCZC detector example didn't have output codes. There are two aspects of state graphs: (1) the states with their transitions and the inputs that cause them, and (2) the outputs associated with transitions. Just as in the flowgraph model we concentrated on control structure and tended to ignore processing that did not directly affect control flow, in state testing we may ignore outputs because it is the states and transitions that are of primary interest. Two state graphs with identical states, inputs, and transitions could have vastly different outputs, yet from a state-testing point of view, they could be identical. Consequently, we reduce the clutter caused by explicit output specifications if outputs are not interesting at the moment.

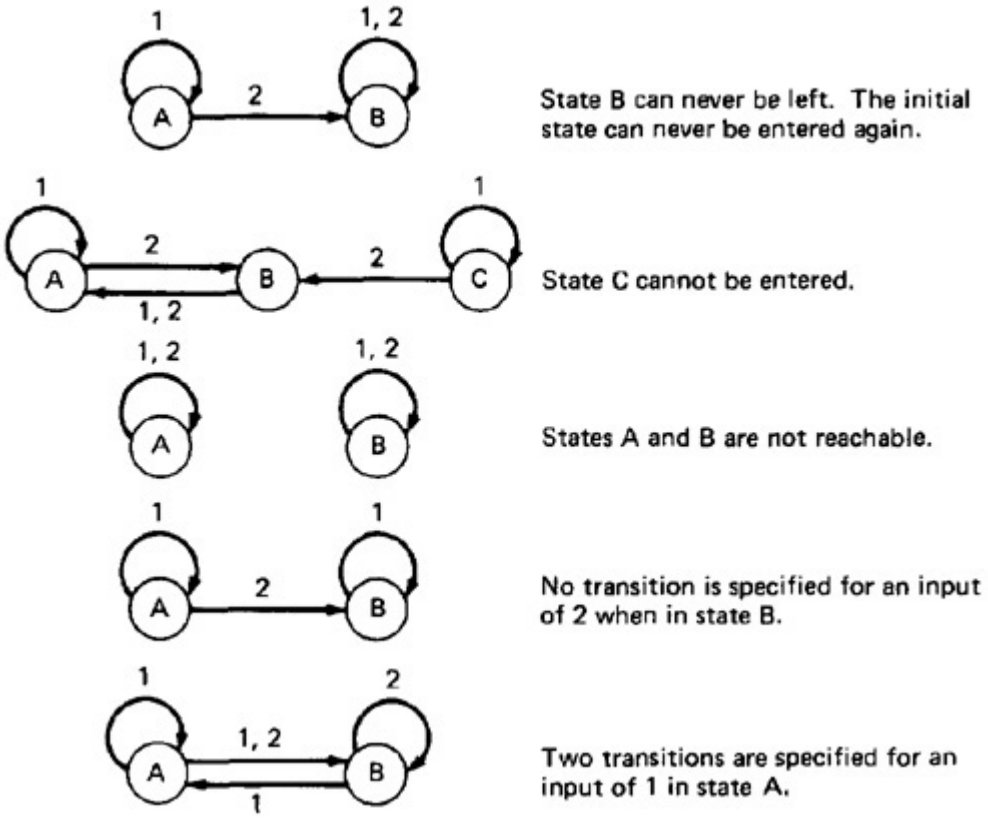


Figure 11.3. Improper State Graphs.

4.2. State Bugs

4.2.1. Number of States

The number of states in a state graph is the number of states we choose to recognize or model. In practice, the state is directly or indirectly recorded as a combination of values of variables that appear in the data base. As an example, the state could be composed of the value of a counter whose possible values ranged from 0 to 9, combined with the setting of two bit flags, leading to a total of $2 \times 2 \times 10 = 40$ states. When the state graph represents an explicit state-table implementation, this value is encoded so bugs in the number of states are less likely; but the encoding can be wrong. Failing to account for all the states is one of the more common bugs in software that can be modeled by state graphs. Because an explicit state-table mechanization is not typical, the opportunities for missing states abound. Find the number of states as follows:

1. Identify all the component factors of the state.
2. Identify all the allowable values for each factor.
3. The number of states is the product of the number of allowable values of all the factors.

Before you do anything else, before you consider one test case, discuss the number of states you think there are with the number of states the programmer (or you, if you're wearing a programmer's hat) thinks there are. Differences of opinion are common. There's no point in designing tests intended to check the system's behavior in various states if there's no agreement on how many states there are. And if there's no agreement on how many states there are, there must be disagreement on what the system does in which states and on the transitions and the outputs. If it seems that I'm giving undue emphasis to the seemingly trivial act of counting states, it's because that act often exhumes fundamental design deficiencies. You don't need to wait until the design is done. A functional specification is usually

enough, inasmuch as state testing is primarily a functional test tool. I read the functional specification and identify the factors and then the number of possible values for each factor. Then I question the designer. I want to get an identification or recognition for each state—with one state corresponding to each combination of condition values. It's gratifying work. It's gratifying to hear, "Oh yeah, I forgot about that one." Make up a table, with a column for every factor, such that all combinations of factors are represented. Before you get concurrence on outputs or transitions or the inputs that cause the transitions, get concurrence from the designer (or confirm for yourself) that every combination listed makes sense.

4.2.2. Impossible States

Some combinations of factors may appear to be impossible. Say that the factors are:

GEAR	R, N, 1, 2, 3, 4	= 6 factors
DIRECTION	Forward, reverse, stopped	= 3 factors
ENGINE	Running, stopped	= 2 factors
TRANSMISSION	Okay, broken	= 2 factors
ENGINE	Okay, broken	= 2 factors
TOTAL		= 144 states

But broken engines can't run, so the combination of factors for engine condition and engine operation yields only 3 rather than 4 states. Therefore, the total number of states is at most 108. A car with a broken transmission won't move for long, thereby further decreasing the number of feasible states. The discrepancy between the programmer's state count and the tester's state count is often due to a difference of opinion concerning "impossible states."

We should say "supposedly impossible" rather than "impossible." There are always alpha particles and bugs in other routines. The implicit or explicit record of the values of the factors in the computer is not always the same as the values of those factors in the world—as was learned at Three Mile Island. One of the contributing factors to that fiasco was a discrepancy between the actual position of an actuator and its reported position. The designers had falsely assumed that it was "impossible" for the actuator's actual position to be at variance with its reported position. Two states, "Actuator-UP/Actuator Position-DOWN" and "Actuator-DOWN/Actuator-Position-UP," were incorrectly assumed to be impossible.

Because the states we deal with inside computers are not the states of the real world but rather a numerical representation of real-world states, the "impossible" states can occur. Wrack your brains for a devious scenario that gets the program into the impossible states, even if the world can't produce such states. If you can't come by such states honestly, invoke alpha particles or lighting storms. A robust piece of software will not ignore impossible states but will recognize them and invoke an illogical-condition handler when they appear to have occurred. That handler will do whatever is necessary to reestablish the system's correspondence to the world.*

---

*The most bizarre case I have ever seen of correspondence loss between the computer's notion of the state and the world's notion of the state occurred at a major international air freight depot more than two decades ago. The computer controlled a vast automated warehouse used to transship and rearrange air cargo loads. Pallets were taken off aircraft and loaded into the automatic warehouse. Automated forklift trucks trundled up and down the aisles and lofted the pallets into push-through bins that were stacked six stories high. Other forklift trucks pulled the pallets out and automatically put them onto conveyors bound for the aircraft on which the pallets belonged. Unfortunately, the designers had made several errors: (1) the power-transient protection was

inadequate for the environment, (2) the hardware was not duplicated, (3) there appeared to be no automatic recovery software, (4) the data-validation checks that should have continually verified the correspondence between the computer's notion of what the bins contained (stored as disc sectors) and the actual contents of the bins were either missing or faulty, but surely inadequate, and (5) test sophistication matched design sophistication. It worked fine for a few days; then came the lightning storm. Correspondence between the computer's version of the bins and the actual bins was lost, but the system kept doing it's thing—a thing out of Walt Disney's *Fantasia* (The Sorcerer's Apprentice). The warehouse was glutted and gutted in hours as automatic forklift trucks tried to stuff more pallets into full bins and remove nonexistent pallets from empty bins. They shut down. The old warehouse, of course, had been decommissioned by then. They then hired hundreds of university students to clamber about the six-story warehouse to identify just what was where. The tons of frozen liver that had not been placed in a refrigerated section were found in a few days. It took much longer to find the corpses in their coffins.

4.2.3. Equivalent States

Two states are **equivalent** if every sequence of inputs starting from one state produces exactly the same sequence of outputs when started from the other state. This notion can also be extended to sets of states. Figure 11.4 shows the situation.

Say that the system is in state S and that an input of *a* causes a transition to state A while an input of *b* causes a transition to state B. The blobs indicate portions of the state graph whose details are unimportant. If, starting from state A, *every* possible sequence of inputs produces *exactly* the same sequence of outputs that would occur when starting from state B, then there is no way that an outside observer can determine which of the two sets of states the system is in without looking at the record of the state. The state graph can be reduced to that of Figure 11.5 without harm.

The fact that there is a notion of state equivalency means that there is an opportunity for bugs to arise from a difference of opinion concerning which states are equivalent. If you insist that there is another factor, not recognized by the programmer, such that the resulting output sequence for a given input sequence is different depending on the value of that factor, then you are asserting that two inequivalent sets of states have been inadvertently merged. Conversely, if you cannot find a sequence of inputs that results in at least one different output when starting from either of two supposedly inequivalent states, then the states *are* equivalent and should be merged if only to simplify the software and thereby reduce the probability of bugs. Be careful, though, because equivalent states could come about as a result of good planning for future enhancements. The two states are presently indistinguishable but could in the future become distinguished as a result of an enhancement that brings with it the distinguishing factor.

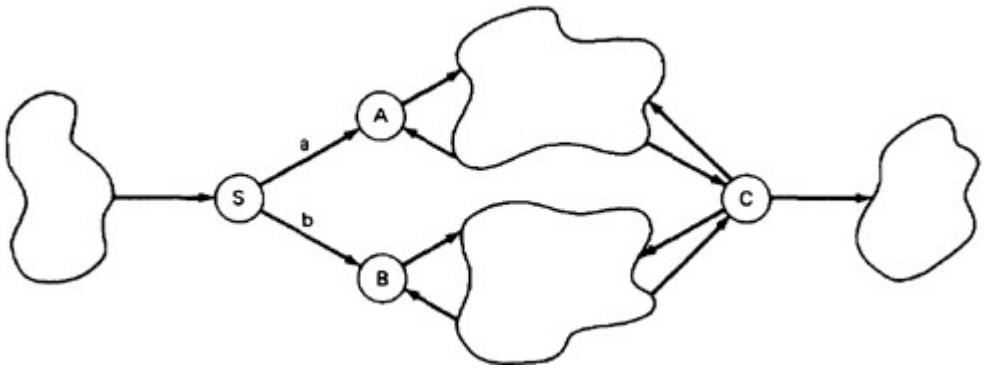
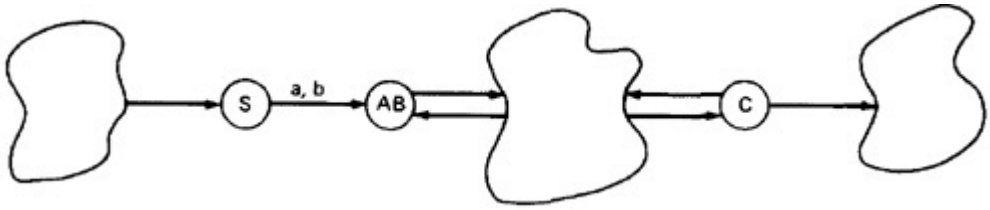


Figure 11.4. Equivalent States.

Figure 11.5. Equivalent States of Figure 11.4



Merged.

Equivalent states can be recognized by the following procedures:

1. The rows corresponding to the two states are identical with respect to input/output/next state but the name of the next state could differ. The two states are differentiated only by the input that distinguishes between them. This situation is shown in Figure 11.6. Except for the *a,b* inputs, which distinguish between states A and B, the system's behavior in the two states is identical for every input sequence; they can be merged.

2. There are two sets of rows which, except for the state names, have identical state graphs with respect to transitions and outputs. The two sets can be merged (see Figure 11.7).

The rows are not identical, but except for the state names (A1 = B2, A2 = B2, A3 = B3), the system's action, when judged by the relation between the output sequence produced by a given input sequence, is identical for either the A or the B set of states. Consequently, this state graph can be replaced by the simpler version shown in Figure 11.7c.

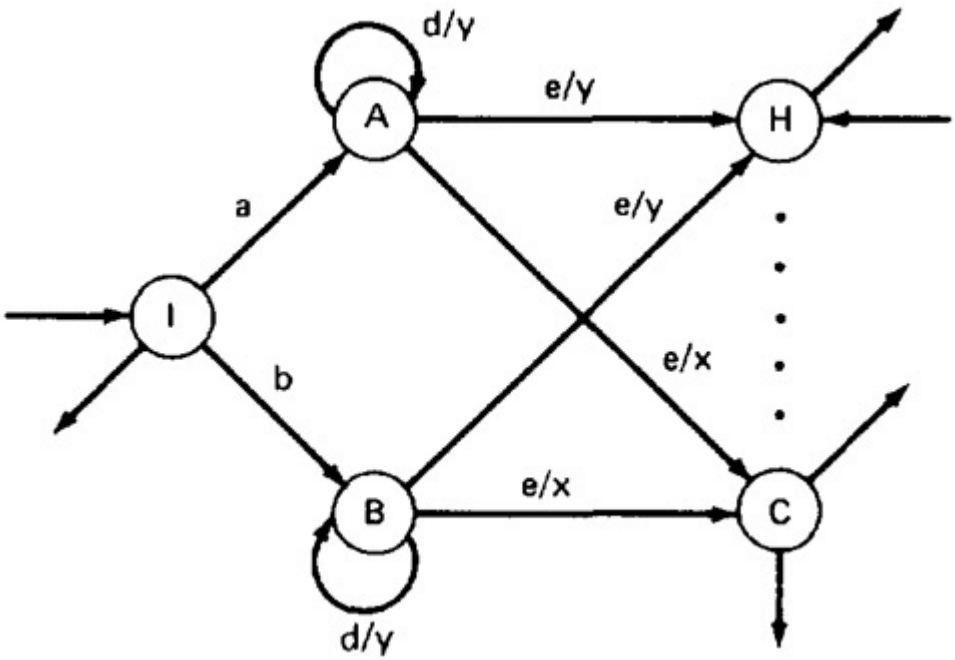
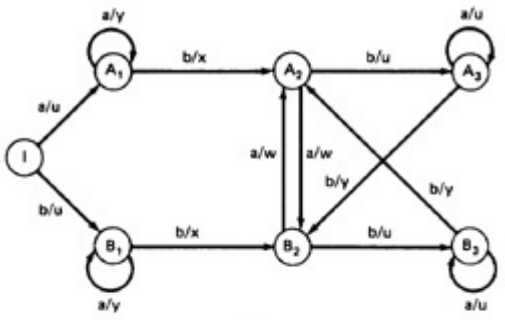


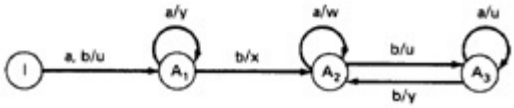
Figure 11.6. Equivalent States.



(a)

STATE	INPUT a	b
I	A_1/Y	B_1/U
A_1	A_1/Y	A_2/X
B_1	B_1/Y	B_2/X
A_2	B_2/W	A_3/U
B_2	A_2/W	B_3/U
A_3	A_3/U	B_2/Y
B_3	B_3/U	A_2/Y

(b)



(c)

Figure 11.7. Merged Equivalent States.

Don't expect to have corresponding states or sets of states so neatly labeled in terms of equivalences. There are more formal methods (beyond the scope of this book—see MILL66) for identifying and spotting such equivalences. The process can also be automated. Because we are using state graphs as a test design tool rather than as a program design tool and because the state graphs we deal with are usually small, a sketch of the state graphs of the two versions (the tester's and the designer's) is usually enough to expose the similarities and the possibility of merging equivalent states.

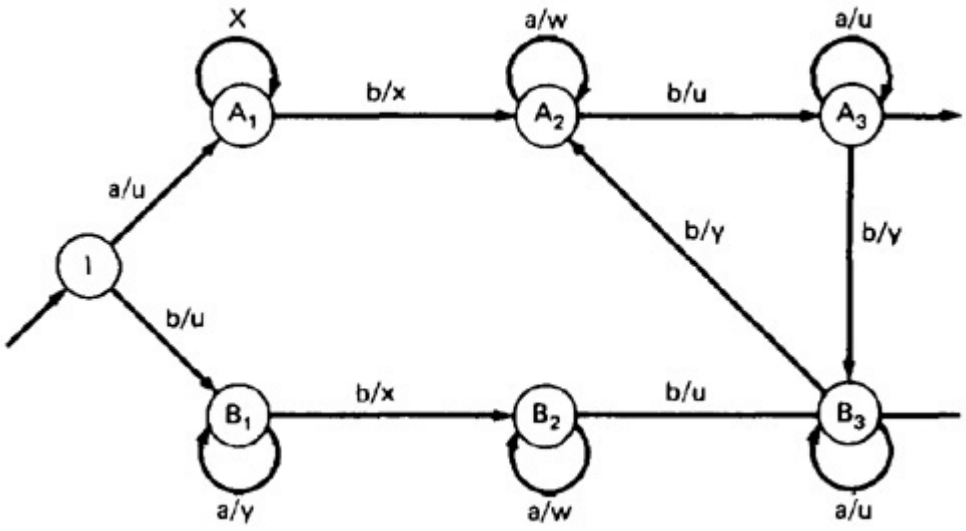


Figure 11.8. Unmergeable States.

Bugs are often the result of the unjustifiable merger of seemingly equivalent states. Two states or two sets of states appear to be equivalent because the programmer has failed to carry through to a proof of

equivalence for *every* input sequence. The first few sequences looked good. Figure 11.8 is an example.

The input sequence *abbbb* produces the output sequence *uxuyy*, while the input sequence *bbbb* produces the output sequence *uxuyu*. The two sets of states are not equivalent, although an incomplete analysis might lead you to believe that they are.

4.3. Transition Bugs

4.3.1. Unspecified and Contradictory Transitions

Every input-state combination must have a specified transition. If the transition is impossible, then there must be a mechanism that prevents that input from occurring in that state—look for it. If there is no such mechanism, what will the program do if, through a malfunction or an alpha particle, the impossible input occurs in that state? The transition for a given state-input combination may not be specified because of an oversight. *Exactly one transition must be specified for every combination of input and state.* However you model it or test it, the system will do *something* for every combination of input and state. It's better that it does what you want it to do, which you assure by specifying a transition rather than what some bugs want it to do.

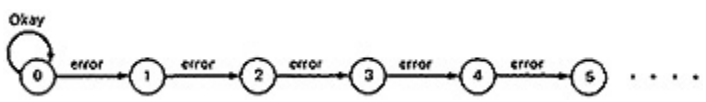
A program can't have contradictions or ambiguities. Ambiguities are impossible because the program will do *something* (right or wrong) for every input. Even if the state does not change, by definition this is a transition to the same state. Similarly, software can't have contradictory transitions because computers can only do one thing at a time. A seeming contradiction could come about in a model if you don't account for *all* the factors that constitute the state and all the inputs. A single bit may have escaped your notice; if that bit is part of the definition of the state it can double the number of states, but if you're not monitoring that factor of the state, it would appear that the program had performed contradictory transitions or had different outputs for what appeared to be the same input from the same state. If you, as a designer, say white debugging "sometimes it works and sometimes it doesn't," you've admitted to a state factor of which you're not aware—a factor probably caused by a bug. Exploring the real state graph and recording the transitions and outputs for each combination of input and state may lead you to discover the bug.

4.3.2. An Example

Specifications are one of the most common source of ambiguities and contradictions. Specifications, unlike programs, can be full of ambiguities and contradictions. The following example illustrates how to convert a specification into a state graph and how contradictions can come about. The tape control routine will be used. Start with the first statement in the specification and add to the state graph one statement at a time. Here is the first statement of the specification:

Rule 1: The program will maintain an error counter, which will be incremented whenever there's an error.

I don't yet know how many states there will be, but I might as well start by naming them with the values of the error counter. The initial state graph might look like this:



There are only two input values, "okay" and "error." A state table will be easier to work with, and it's

much easier to spot ambiguities and contradictions. Here's the first state table:

	INPUT	
STATE	**OKAY**	**ERROR**
0	0/none	1/
1		2/
2		3/
3		4/
4		5/
5		6/
6		7/
7		8/

There are no contradictions yet, but lots of ambiguities. It's easy to see how ambiguities come about—just stop the specification before it's finished. Let's add the rules one at a time and fill in the state graph as we go. Here are the rest of the rules; study them to see if you can find the problems, if any:

Rule 2: If there is an error, rewrite the block.

Rule 3: If there have been three successive errors, erase 10 centimeters of tape and then rewrite the block.

Rule 4: If there have been three successive erasures and another error occurs, put the unit out of service.

Rule 5: If the erasure was successful, return to the normal state and clear the error counter.

Rule 6: If the rewrite was unsuccessful, increment the error counter, advance the state, and try another rewrite.

Rule 7: If the rewrite was successful, decrement the error counter and return to the previous state.

Adding rule 2, we get

	INPUT	
STATE	**OKAY**	**ERROR**
0	0/NONE	1/REWRITE
1		2/REWRITE
2		3/REWRITE
3		4/REWRITE
4		5/REWRITE
5		6/REWRITE
6		7/REWRITE
7		8/REWRITE

Rule 3: If there have been three successive errors, erase 10 centimeters of tape and then rewrite the block.

	INPUT	
STATE	**OKAY**	**ERROR**
0	0/NONE	1/REWRITE
1		2/REWRITE
2		3/REWRITE, ERASE, REWRITE
3		4/REWRITE, ERASE, REWRITE
4		5/REWRITE, ERASE, REWRITE
5		6/REWRITE, ERASE, REWRITE
6		7/REWRITE, ERASE, REWRITE
7		8/REWRITE, ERASE, REWRITE

Rule 3, if followed blindly, causes an unnecessary rewrite. It's a minor bug, so let it go for now, but it pays to check such things. There might be an arcane security reason for rewriting, erasing, and then rewriting again.

Rule 4: If there have been three successive erasures and another error occurs, put the unit out of service.

	INPUT	
STATE	**OKAY**	**ERROR**
0	0/NONE	1/RW
1		2/RW
2		3/ER, RW
3		4/ER, RW
4		5/ER, RW
5		6/OUT
6		
7		

Rule 4 terminates our interest in this state graph so we can dispose of states beyond 6. The details of state 6 will not be covered by this specification; presumably there is a way to get back to state 0. Also, we can credit the specifier with enough intelligence not to have expected a useless rewrite and erase prior to going out of service.

Rule 5: If the erasure was successful, return to the normal state and clear the counter.

	INPUT	
STATE	**OKAY**	**ERROR**
0	0/NONE	1/RW
1		2/RW
2		3/ER, RW
3	0/NONE	4/ER, RW

4	0/NONE	5/ER, RW
5	0/NONE	6/OUT
6		

Rule 6: If the rewrite was unsuccessful, increment the error counter, advance the state, and try another rewrite.

Because the value of the error counter is the state, and because rules I and 2 specified the same action, there seems to be no point to rule 6 unless yet another rewrite was wanted. Furthermore, the order of the actions is wrong. If the state is advanced before the rewrite, we could end up in the wrong state. The proper order should have been: output = attempt-rewrite and then increment the error counter.

Rule 7: If the rewrite was successful, decrement the error counter and return to the previous state.

	INPUT	
STATE	**OKAY**	**ERROR**
0	0/NONE	1/RW
1	0/NONE	2/RW
2	1/NONE	3/ER, RW
3	0/NONE 2/NONE	4/ER, RW
4	0/NONE 3/NONE	5/ER, RW
5	0/NONE 4/NONE	6/OUT
6		

Rule 7 got rid of the ambiguities but created contradictions. The specifier's intention was probably:

Rule 7A: If there have been no erasures and the rewrite is successful, return to the previous state.

We're guessing, of course, and we could be wrong, especially if the issue is obscure and the technical details unfamiliar. The only thing you can assume is that it's unlikely that a satisfactory implementation will result from a contradictory specification. If the state graph came from a design specification, be especially stupid. Be literal in your interpretation and smile when the designer accuses you of semantic nit-picking. It's tragic how such "purely semantic" issues turn out to cause tough bugs.

4.3.3. Unreachable States

An **unreachable state** is like unreachable code—a state that no input sequence can reach. An unreachable state is not impossible, just as unreachable code is not impossible. Furthermore, there may be transitions from the unreachable state to other states; there usually are because the state became unreachable as a result of incorrect transitions.

Unreachable states can come about from previously "impossible" states. You listed all the factors and laid out a state table. Some of these states corresponded to previously "impossible" states. The designer,

perhaps after some rough persuasion, agrees that something should be done about the unreachable states. "Easy," he thinks, "provide no transitions into them." Yet there should still be a transition *out* of all such states. At least there should be a transition to an error-recovery procedure or an exception handler.

An isolated, unreachable state here and there, which clearly relates to impossible combinations of real-world state-determining conditions, is acceptable, but if you find groups of connected states that are isolated from others, there's cause for concern. There are two possibilities: (1) There is a bug; that is, some transitions are missing. (2) The transitions are there, but you don't know about it; in other words, there are other inputs and associated transitions to reckon with. Typically, such hidden transitions are caused by software operating at a higher priority level or by interrupt processing.

4.3.4. Dead States

A **dead state**, (or set of dead states) is a state that once entered cannot be left. This is not necessarily a bug, but it is suspicious. If the software was designed to be the fuse for a bomb, we would expect at least one such state. A set of states may appear to be dead because the program has two modes of operation. In the first mode it goes through an initialization process that consists of several states. Once initialized, it goes to a strongly connected set of working states, which, within the context of the routine, cannot be exited. The initialization states are unreachable to the working states, and the working states are dead to the initialization states. The only way to get back might be after a system crash and restart. Legitimate dead states are rare. They occur mainly with system-level issues and device handlers. In normal software, if it's not possible to get from any state to any other, there's reason for concern.

4.4. Output Errors

The states, the transitions, and the inputs could be correct, there could be no dead or unreachable states, but the output for the transition could be incorrect. Output actions must be verified independently of states and transitions. That is, you should distinguish between a program whose state graph is correct but has the wrong output for a transition and one whose state graph is incorrect. The likeliest reason for an incorrect output is an incorrect call to the routine that executes the output. This is usually a localized and minor bug. Bugs in the state graph are more serious because they tend to be related to fundamental control-structure problems. If the routine is implemented as a state table, both types of bugs are comparably severe.

4.5. Encoding Bugs

It would seem that encoding bugs for input coding, output coding, state codes, and state-symbol product formation could exist as such only in an explicit finite-state machine implementation. The possibility of such bugs is obvious for a finite-state machine implementation, but the bugs can also occur when the finite-state machine is implicit. If the programmer has a notion of state and has built an implicit finite-state machine, say by using a bunch of program flags, switches, and "condition" or "status" words, there may be an encoding process in place.

Make it a point *not* to use the programmer's state numbers and/or input codes. As a tester, you're dealing with an abstract machine that you're going to use to develop tests. *The behavior of a finite-state machine is invariant under all encodings.* That is, say that the states are numbered 1 to n. If you renumber the states by an arbitrary permutation, the finite-state machine is unchanged—similarly for input and output codes. Therefore, if you present your version of the finite-state machine with a different encoding, and if the programmer objects to the renaming or claims that behavior is changed as a result, then use that as a signal to look for encoding bugs. You may have to look at the implementation for these, especially the data dictionary. Look for "status" codes and read the list carefully. The key words

are “unassigned,” “reserved,” “impossible,” “error,” or just gaps.

The implementation of the fields as a bunch of bits or bytes tells you the potential size of the code. If the number of code values is less than this potential, there is an encoding process going on, even if it’s only to catch values that are out of range. In strongly typed languages with user-defined semantic types, the encoding process is probably a type conversion from a set membership, say, to a pointer type or integer. Again, you may have to look at the program to spot potential bugs of this kind.

5. STATE TESTING

5.1. Impact of Bugs

Let’s say that a routine is specified as a state graph that has been verified as correct in all details. Program code or tables or a combination of both must still be implemented. A bug can manifest itself as one or more of the following symptoms:

1. Wrong number of states.
2. Wrong transition for a given state-input combination.
3. Wrong output for a given transition.
4. Pairs of states or sets of states that are inadvertently made equivalent (factor lost).
5. States or sets of states that are split to create inequivalent duplicates.
6. States or sets of states that have become dead.
7. States or sets of states that have become unreachable.

5.2. Principles

The strategy for state testing is analogous to that used for path-testing flowgraphs. Just as it’s impractical to go through every possible path in a flowgraph, it’s impractical to go through every path in a state graph. A path in a state graph, of course, is a succession of transitions caused by a sequence of inputs. The notion of coverage is identical to that used for flowgraphs—pass through each link (i.e., each transition must be exercised). Assume that some state is especially interesting—call it the initial state. Because most realistic state graphs are strongly connected, it should be possible to go through all states and back to the initial state, when starting from there. But don’t do it. Even though most state testing can be done as a single case in a grand tour, it’s impractical to do it that way for several reasons:

1. In the early phases of testing, you’ll never complete the grand tour because of bugs.
2. Later, in maintenance, testing objectives are understood, and only a few of the states and transitions have to be retested. A grand tour is a waste of time.
3. There’s so much history in a long test sequence and so much has happened that verification is difficult.

The starting point of state testing is:

1. Define a set of covering input sequences that get back to the initial state when starting from the initial state.
2. For each step in each input sequence, define the expected next state, the expected transition, and the expected output code.

A set of tests, then, consists of three sets of sequences:

1. Input sequences.
2. Corresponding transitions or next-state names.
3. Output sequences.

5.3. Limitations and Extensions

Just as link coverage in a flowgraph model of program behavior did not guarantee "complete testing," state-transition coverage in a state-graph model does not guarantee complete testing. Things are slightly better because it's not necessary to consider any sequence longer than the total number of states. *Note:* Everything discussed in this section applies equally well to control flowgraphs with suitable translation.

Chow (CHOW78) defines a hierarchy of paths and methods for combining paths to produce covers of a state graph. The simplest is called a "0 switch," which corresponds to testing each transition individually. The next level consists of testing transition sequences consisting of two transitions, called "1 switches." The maximum-length switch is an $n - 1$ switch, where n is the number of states. Chow's primary result shows that in general, a 0 switch cover (which we recognize as branch cover for control flowgraphs) can catch output errors but may not catch some transition errors. In general, one must use longer and longer covering sequences to catch transition errors, missing states, extra states, and the like. The theory of what constitutes a sufficient number of tests (i.e., input sequences) to catch specified kinds of state-graph errors is still in its infancy and is beyond the scope of this book. Furthermore, practical experience with the application of such theory to software as exists is limited, and the efficacy of such methods as bug catchers has yet to be demonstrated sufficiently well to earn these methods a solid place in the software tester's tool repertoire. Work continues and progress in the form of semiautomatic test tools and effective methods are sure to come. Meanwhile, we have the following experience:

1. Simply identifying the factors that contribute to the state, calculating the total number of states, and comparing this number to the designer's notion catches some bugs.
2. Insisting on a justification for all supposedly dead, unreachable, and impossible states and transitions catches a few more bugs.
3. Insisting on an explicit specification of the transition and output for every combination of input and state catches many more bugs.
4. A set of input sequences that provide coverage of all nodes and links is a mandatory minimum requirement.
5. In executing state tests, it is essential that means be provided (e.g., instrumentation software) to record the sequence of states (e.g., transitions) resulting from the input sequence and not just the outputs that result from the input sequence.

5.4. What to Model

Because every combination of hardware and software can in principle be modeled by a sufficiently complicated state graph, this representation of software behavior is applicable to every program. The utility of such tests, however, is more limited. The state graph is a behavioral model—it is functional rather than structural and is thereby far removed from the code. As a testing method, it is a bottom-line method that ignores structural detail to focus on behavior. It is advantageous to look into the database to see how the factors that create the states are represented in order to get a state count. More than most test methods, state testing yield the biggest payoffs during the design of the tests rather than during the running thereof. Because the tests can be constructed from a design specification long before coding, they help catch deep bugs early in the game when correction is inexpensive. Here are some situations in which state testing may prove useful:

1. Any processing where the output is based on the occurrence of one or more sequences of

events, such as detection of specified input sequences, sequential format validation, parsing, and other situations in which the order of inputs is important.
2. Most protocols between systems, between humans and machines, between components of a system (CHOI84, CHUN78, SARI88).
3. Device drivers such as for tapes and discs that have complicated retry and recovery procedures if the action depends on the state.
4. Transaction flows where the transactions are such that they can stay in the system indefinitely—for example, online users, tasks in a multitasking system.
5. High-level control functions within an operating system. Transitions between user states, supervisor's states, and so on. Security handling of records, permission for read/write/modify privileges, priority interrupts and transitions between interrupt states and levels, recovery issues and the safety state of records and/or processes with respect to recording recovery data.
6. The behavior of the system with respect to resource management and what it will do when various levels of resource utilization are reached. Any control function that involves responses to thresholds where the system's action depends not just on the threshold value, but also on the direction in which the threshold is crossed. This is a normal approach to control functions. A threshold passage in one direction stimulates a recovery function, but that recovery function is not suspended until a second, lower threshold is passed going the other way.
7. A set of menus and ways that one can go from one to the other. The currently active menus are the states, the input alphabet is the choices one can make, and the transitions are invocations of the next menu in a menu tree. Many menu-driven software packages suffer from dead states—menus from which the only way out is to reboot.
8. Whenever a feature is directly and explicitly implemented as one or more state-transition tables.

5.5. Getting the Data

As is so often the case in the independent tester's life, getting the data on which the model is to be based is half the job or more. There's no magic for doing that: reading documents, interviews, and all the rest. State testing, more than most functional test strategies, tends to have a labor-intensive data-gathering phase and tends to need many more meetings to resolve issues. This is the case because most of the participants don't realize that there's an essential state-machine behavior. For nonprogrammers, especially, the very concept of finite-state machine behavior may be missing. Be prepared to spend more time on getting data than you think is reasonable and be prepared to do a lot of educating along the way.

5.6. Tools

Good news and bad news: The telecommunications industry, especially in telephony, has been using finite-state machine implementations of control functions for decades (BAUE79). They also use several languages/systems to code state tables directly. Similarly, there are tools to do the same for hardware logic designs. That's the good news. The bad news is that these systems and languages are proprietary, of the home-brew variety, internal, and/or not applicable to the general use of software implementations of finite-state machines. The most successful tools are not published and are unlikely to be published because of the competitive advantage they give to the users of those tools.

6. TESTABILITY TIPS

6.1. A Balm for Programmers

Most of this chapter has taken the independent tester's viewpoint and has been a prescription for making

programmers squirm. What is testability but means by which programmers can protect themselves from the ravages of sinister independent testers? What is testability but a guide to cheating—how to design software so that the pesticide paradox works and the tester's strongest technique is made ineffectual? The key to testability design is easy: build explicit finite-state machines.

6.2. How Big, How Small?

I understand every two-state finite-state machine because, including the good and bad ones, there are only eight of them. There are about eighty possible good and bad three-state machines, 2700 four-state machines, 275,000 five-state machines, and close to 100 million six-state machines, most of which are bad. We learned long ago, as hardware logic designers, that it paid to build explicit finite-state machines for even very small machines. I think you can safely get away with two states, it's getting difficult for three states, a heroic act for four, and beyond human comprehension for five states. That doesn't mean that you have to build your finite-state machine as in the explicit PDL example given above, but that you must do a finite-state machine model and identify how you're implementing every part of that model for anything with four or more states.

6.3. Switches, Flags, and Unachievable Paths

Something may look like a finite-state machine but not be one. Figure 11.9a shows a program with a switch or flag. Someplace early in the routine we set a flag, A, then later we test the flag and go one way or the other depending on its value. In Figure 11.9b we've rewritten the routine to eliminate the flag. As soon as the flag value is calculated, we branch. The cost is the cost of converting segment V into a subroutine and calling it twice. But note that we went from four paths, two of which are unachievable to two paths, both of which are achievable and both of which are needed to achieve branch coverage.

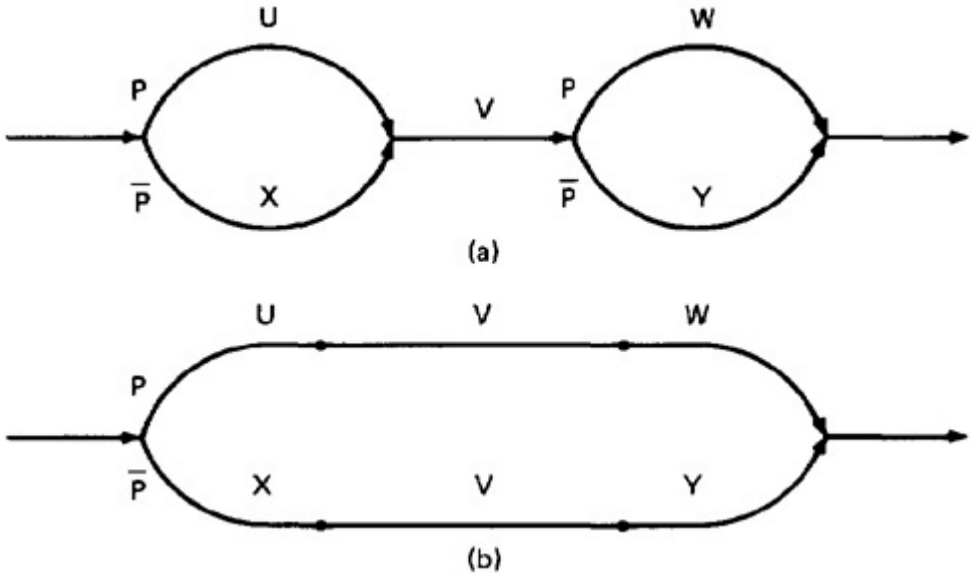


Figure 11.9. Program with One Switch Variable.

In Figure 11.10, the situation is more complicated. There are three switches this time. Again, where we go depends on the switch settings calculated earlier in the program. We can put the decision up front and branch directly, and again use subroutines to make each path explicit and do without the switches. The advantages of this implementation is that if any of the combinations are not needed, we merely clip out that part of the decision tree, as in Figure 11.10c. Again, all paths are achievable and all paths are needed for branch cover.

Figure 11.11 is similar to the previous two except that we've put the switched parts in a loop. It's even worse if the loop includes the switch value calculations (dotted link). We now have a very difficult situation. We don't know which of these paths are achievable and which are or are not required. What is or is not achievable depends on the switch settings. Branch coverage won't do it: we must do or attempt branch coverage in every possible state.

6.4. Essential and Inessential Finite-State Behavior

Program flags and switches are predicates deferred. There is a significant, qualitative difference between finite-state machines and combinational machines. A combinational machine selects paths based on the values of predicates, the predicates depend only on prior processing and the predicates' truth values will not change once they have been determined. Any path corresponds to a boolean algebra expression over the predicates. Furthermore, it does not matter in which order the decisions are made. The fact that there is an ordering is a consequence of a sequential, Von Neumann computer architecture. In a parallel-data-flow machine, for example, the decisions and path selections could be made simultaneously. Sequence and finite-state behavior are in this case implementation consequences and not essential. The combinational machine has exactly one state and one transition back to itself for all possible inputs. The control logic of a combinational program can be described by a decision table or a decision tree.

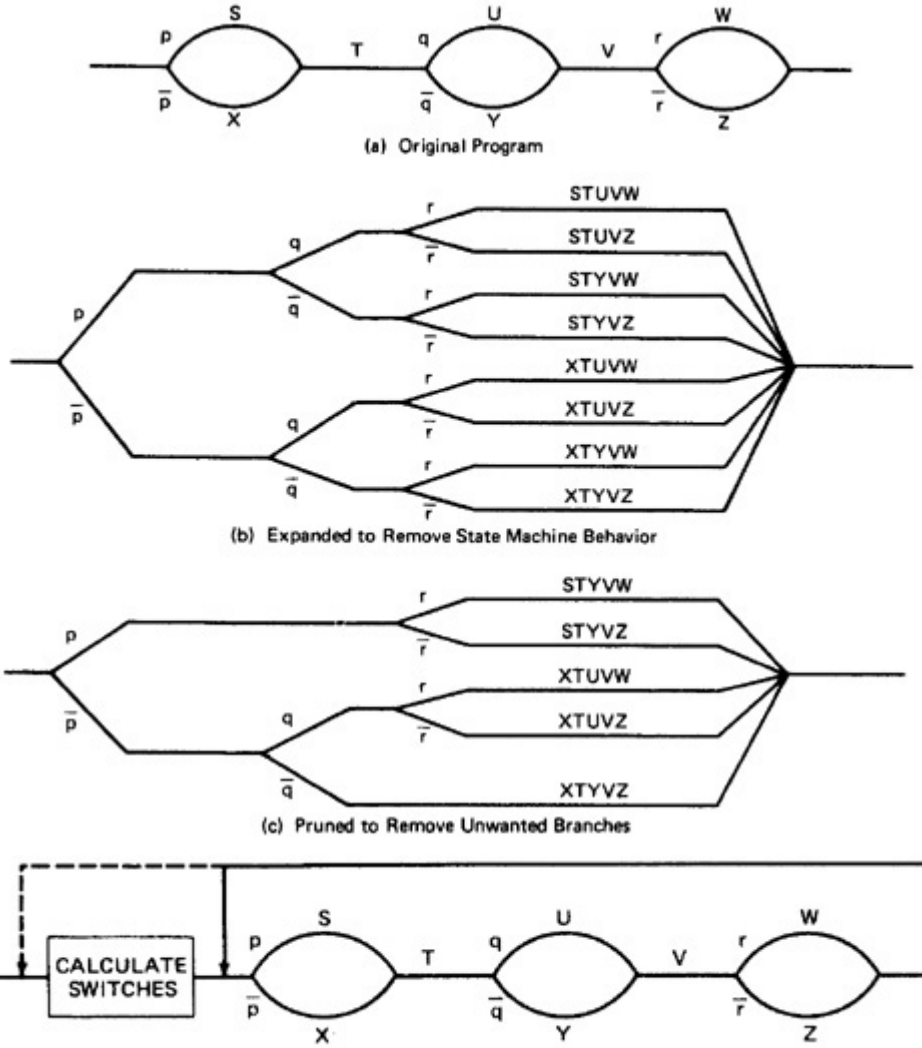


Figure 11.10. Three-Switch-Veriable Program.

Figure 11.11. Switches in a Loop.

The simplest essential finite-state machine is a flip-flop. There is no logic that can implement it without

some kind of feedback. You cannot describe this behavior by a decision table or decision tree unless you provide feedback into the table or call it recursively. It must have a loop or the equivalent.

The problem with nontrivial finite-state machine behavior is that to do the equivalent of branch testing, say, you must do it over for every state. Why take on that extra burden if the finite-state machine behavior isn't essential?

Most programmers' implementation of finite-state behavior is not essential—it appears to be convenient. Most programmers, having implemented finite-state behavior, will not test it properly. I've yet to see a programmer who implemented a routine with 10 flags rerun the tests for all 1024 possible flag settings.

Learn to distinguish between essential and inessential finite-state behavior. It's not essential if you can do it by a parallel program in a hypothetical data-flow machine. It's not essential if a decision-table model will do it for you or if you can program it as a big decision tree. It's not essential if the program's exit expression (Chapter 10), even with explicit loops, equals unity. It's not essential if there's a nonunity exit expression but it turns out that you don't really want to loop under the looping conditions. I'm not telling you to throw away your "harmless" little flags and switches and not to implement inessential finite-state machine behavior. All I ask is that you be prepared to repeat your tests in every state.

6.5. Design Guidelines

I'll assume that you've checked the design and the specification and that you're not about to implement inessential finite-state machine behavior. What should you do if you must build finite-state machines into your code?

1. Learn how it's done in hardware. I know of no books on finite-state machine design for programmers. There are only books on hardware logic design and switching theory, with a distinct hardware flavor and you'll have to adapt their methods to software.
2. Start by designing the abstract machine. Verify that it is what you want to do. Do an explicit analysis, in the form of a state graph or table, for anything with three states or more.
3. Start with an explicit design—that is, input encoding, output encoding, state code assignment, transition table, output table, state storage, and how you intend to form the state-symbol product. Do this at the PDL level. But be sure to document that explicit design.
4. Before you start taking shortcuts, see if it really matters. Neither the time nor the memory for the explicit implementation usually matters. Do a prototype based on the explicit design and analyze that or measure it to see what the processing time actually is and if that's significant. Remember that explicit finite-state machines are usually very fast and that the penalty is likelier to be a memory cost than a time cost. Test the prototype thoroughly, as discussed above. The prototype test suite should be kept for later use.
5. Take shortcuts by making things implicit only as you must to make significant reductions in time or space and only if you can show that such savings matter in the context of the whole system. After all, doubling the speed of your implementation may mean nothing if all you've done is shaved 100 microseconds from a 500-millisecond process. The order in which you should make things implicit are: output encoding, input encoding, state code, state-symbol product, output table, transition table, state storage. That's the order from least to most dangerous.
6. Consider a hierarchical design if you have more than a few dozen states.
7. Build, buy, or implement tools and languages that implement finite-state machines as software if you're doing more than a dozen states routinely.
8. Build in the means to initialize to any arbitrary state. Build in the transition verification instrumentation (the coverage analyzer). These are much easier to do with an explicit machine.

7. SUMMARY

1. State testing is primarily a functional testing tool whose payoff is best in the early phases of design.
2. A program can't have contradictory or ambiguous transitions or outputs, but a specification can and does. Use a state table to verify the specification's validity.
3. Count the states.
4. Insist on a specification of transition and output for every combination of input and states.
5. Apply a minimum set of covering tests.
6. Instrument the transitions to capture the sequence of states and not just the sequence of outputs.
7. Count the states.

<ch10 toc ch12>