**UNIT-V**

**TERMINOLOGY**

|  |  |
| --- | --- |
| STATE | A state can be defined as a combination of circumstances or attributes belonging for the time being to a person or thing. |
| STATE GRAPH | The state graph and its associated state table are useful models for describing the software behaviour. |
| FINITE STATE MACHINE | The finite state machine is a functional testing tool and testable design programming tool.   * The finite state machine is a fundamental to software engineering as Boolean algebra.   FSM can also be implemented as table driven software, in which case they are powerful design options. |
| STATE TABLE | A state table is used to represent the state graph in tabular format that specifies the states, the inputs, the transitions and outputs.  The following are the conventions used in a state table or state transition table:-  Each row in the table corresponds to a state.  Each column in the table corresponds to an input condition.  The box at the intersection of a row and a column specifies the next state(the transition) and the output if any. |
| GOOD STATE GRAPH | The total number of states is equal to the product of the possibilities of factors that make up the state.  For every state and input there is exactly one transition specified to exactly one possibly the same, state.  For every transition there is one output action specified.  For every state there is a sequence of inputs that will drive the system back to the same state. |
| EQUIVALENT STATE | Two states are said to be equivalent if every sequence of inputs starting from one state produces exactly the same sequence of outputs when started from the other state. |
| UNREACHABLE STATE | A state that no input sequence can reach |
| DEAD STATE | A dead state is a state that once entered cannot be left. |
| GRAPH MATRIX | A graph matrix is a square array with one row and one column for every node in the graph. Each row-column combination corresponds to a relation between the node corresponding to the row and the node corresponding to a column. |
| EQUIVALENCE RELATION | An equivalence relation is relation if it satisfies the reflexive, transitive and symmetric properties. |
| PARTIAL ORDERING RELATION | A partial ordering relation is a relation if it satisfies the reflexive, transitive and asymmetric properties. |
| BUILDING TOOLS OF MATRIX REPRESENTATION | Node degree and graph density  What’s wrong with arrays/  Space  Weights  Variable length Weights  Processing time  Linked list representation |
| OPERATIONS ON MATRIX | Parallel reduction :  Loop reduction |

**Concepts**

|  |  |
| --- | --- |
| STATE GRAPHS | 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.  *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  *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 umber 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.  *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.  *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:    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.  *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.  *Software Implementation*  *IMPLEMENTATION AND OPERATION*  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 stateinput  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.  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.  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.  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.  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.  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. |
| GOOD STATE GRAPHS AND BAD | 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.  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.  *State Bugs:*  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 × 2 × 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.  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.  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.  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.  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. 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.  *Transition Bugs:*  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. 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.  An Example:  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.  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:    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    Rule 3: If there have been three successive errors, erase 10 centimeters of tape and then rewrite the block.    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.    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.    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.    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.  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.  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.  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.  *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.  *Encoding Bugs:*  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.  *The behavior of a finite-state machine is invariant underall 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. |
| STATE TESTING | *Impact of Bugs:*  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.  *Principles:*  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.  *Limitations and Extensions:*  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.  *What to Model:*  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.  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.  *Getting the Data:*  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.  *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.  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. |
| TESTABILITY TIPS | *A Balm for Programmers:*  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.  *How Big, How Small?*  Long ago, as hardware logic designers, that it paid to build explicit finite-state machines for even very small machines. 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.  *Switches, Flags, and Unachievable Paths:*  Something may look like a finite-state machine but not be one. 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.  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.  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. 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.  *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.  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.  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.  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, even with explicit loops, equals unity. It‘s not essential if there‘s a non unity 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.  *Design Guidelines:*  *To* 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. |
| GRAPH MATRICES AND APPLICATIONS | OVERVIEW:  *The Problem with Pictorial Graphs:*  Whenever a graph is used as a model, sooner or later we trace paths through it—to find a set of covering paths, a set of values that will sensitize paths, the logic function that controls the flow, the processing time of the routine, the equations that define a domain, whether the routine pushes or pops, or whether a state is reachable or not.  Even algebraic representations such as BNF and regular expressions can be converted to equivalent graphs. Much of test design consists of tracing paths through a graph and most testing strategies define some kind of cover over some kind of graph.  Path tracing is not easy, and it‘s subject to error. You can miss a link here and there or cover some links twice— even if you do use a marking pen to note which paths have been taken. You‘re tracing a long complicated path through a routine when the telephone rings—you‘ve lost your place before you‘ve had a chance to mark it. I get confused tracing paths, so naturally I assume that other people also get confused.  One solution to this problem is to represent the graph as a matrix and to use matrix operations equivalent to path tracing. These methods aren‘t necessarily easier than path tracing, but because they‘re more methodical and mechanical and don‘t depend on your ability to ―see‖ a path, they‘re more reliable.  Even if you use powerful tools that do everything that can be done with graphs, and furthermore, enable you to do it graphically, it‘s still a good idea to know how to do it by hand.  *Tool Building:*  If you build test tools or want to know how they work, sooner or later you‘ll be implementing or investigating analysis routines based on these methods—or you should be. Think about how a naive tool builder would go about finding a property of all paths (a possibly infinite number) versus how one might do it. The properties of graph matrices are fundamental to test tool building.  *Doing and Understanding Testing Theory:*  Without the conceptual apparatus of graph matrices, you‘ll be blind to much of testing theory,  especially those parts that lead to useful algorithms.  *The Basic Algorithms:*  The basic toolkit consists of:  1. Matrix multiplication, which is used to get the path expression from every node to every other node.  2. A partitioning algorithm for converting graphs with loops into loop-free graphs of equivalence classes.  3. A collapsing process (analogous to the determinant of a matrix), which gets the path expression from  any node to any other node. |
| THE MATRIX OF A GRAPH | *Basic Principles:*  A graph matrix is a square array with one row and one column for every node in the graph. Each row-column combination corresponds to a relation between the node corresponding to the row and the node corresponding to the column. The relation, for example, could be as simple as the link name, if there is a link between the nodes.  Observe the following in matrices:  1. The size of the matrix (i.e., the number of rows and columns) equals the number of nodes.  2. There is a place to put every possible direct connection or link between any node and any other node.  3. The entry at a row and column intersection is the link weight of the link (if any) that connects the two  nodes in that direction.  4. A connection from node *i* to node *j* does not imply a connection from node *j* to node *i*. Note that in  Figure 12.1h the (5,6) entry is *m*, but the (6,5) entry is *c*.  5. If there are several links between two nodes, then the entry is a sum; the ―+‖ sign denotes parallel links  as usual.  *A Simple Weight:*  The simplest weight we can use is to note that there is or isn‘t a connection. Let ―1‖ mean that there is a connection  and ―0‖ that there isn‘t. The arithmetic rules are:  1 + 1 = 1, 1 + 0 = 1, 0 + 0 = 0,  1 × 1 = 1, 1 × 0 = 0, 0 × 0 = 0.  A matrix with weights defined like this is called a connection matrix.  Each row of a matrix (whatever the weights) denotes the outlinks of the node corresponding to that row, and each column denotes the inlinks corresponding to that node. A branch node is a node with more  than one nonzero entry in its row. A junction node is a node with more than one nonzero entry in its column.  A self-loop is an entry along the diagonal. Using the principle that a case statement is equivalent to *n* – 1 binary decisions, by subtracting 1 from the total number of entries in each row and ignoring rows with no entries (such as node 2), we obtain the equivalent number of decisions for each row. Adding these values and then adding I to the sum yields the graph‘s cyclomatic complexity.  *Further Notation:*  Talking about the ―entry at row 6, column 7‖ is wordy. To compact things, the entry corresponding to node *i* and column*j*, which is to say the link weights between nodes *i* and *j*, is denoted by *aij*. A self-loop about node *i* is denoted by *aii*, while the link weight for the link between nodes *j* and *i* is denoted by *aji*.  The expression ―*aijajjajm*‖ denotes a path from node i to j, with a self-loop at *j* and then a link from node *j* to node *m*.  The expression ―*aijajkakmami*‖ denotes a path from node *i* back to node *i* via nodes *j*, *k*, and *m*. An expression such as ―*aikakmamj*+ *ainanpapj*‖ denotes a pair of paths between nodes *i* and *j*, one going via nodes *k* and *m* and the other via nodes *n* and *p*.  This notation may seem cumbersome, but it‘s not intended for working with the matrix of a graph but for  expressing operations on the matrix. It‘s a very compact notation.  For example, denotes the set of all possible paths between nodes *i* and *j* via one intermediate node. But because ―*i* and ―*j* denote any node, this expression is the set of all possible paths between any two nodes via one intermediate node.  The transpose of a matrix is the matrix with rows and columns interchanged. It is denoted by a superscript letter ―T, as in AT. If C = AT then *cij* = *aji*. The intersection of two matrices of the same size, denoted by A#B is a matrix obtained by an element-by-element multiplication operation on the entries.  For example, C = A#B means *cij* = *aij*#*bij*. The multiplication operation is usually boolean AND or set intersection. Similarly, the union of two matrices is defined as the element-by-element addition operation such as a boolean OR or set union. |
| RELATIONS | *General:*  A relation is a property that exists between two (usually) objects of interest. We‘ve had many examples of relations in this book. Here‘s a sample, where *a* and *b* denote objects and R is used to denote that a has the relation R to *b*:  1. ―Node *a is connected* to node *b*‖ or *a*R*b* where ―R‖ means ―is connected to.‖  2. ―*a* >= *b*‖ or *a*R*b* where ―R‖ means ―greater than or equal.‖  3. ―*a is a subset* of *b*‖ where the relation is ―is a subset of.‖  4. ―It takes 20 microseconds of processing time to get from node *a* to node *b*.‖ The relation is expressed  by the number 20.  5. ―Data object X is defined at program node *a* and used at program node *b*.‖ The relation between nodes  *a* and *b* is that there is a *du* chain between them.  Let‘s now redefine what we mean by a graph. graph consists of a set of abstract objects called nodes and a relation R between the nodes.  If *a*R*b*, which is to say that a has the relation R to *b*, it is denoted by a link from *a* to *b*. In addition to the fact that the relation exists, for some relations we can associate one or more properties. These are called link weights. A link weight can be numerical, logical, illogical, objective, subjective, or whatever. Furthermore, there is no limit to the number and type of link weights that one may associate with a relation.  ―Is connected to‖ is just about the simplest relation there is: it is denoted by an unweighted link. Graphs defined over ―is connected to‖ are called, as we said before, connection matrices.\* For more general relations, the matrix is called a relation matrix.  *Properties of Relations:*  Any given relation may or may not have these properties, in almost any combination.  Transitive Relations:  A relation R is transitive if *a*R*b* and *b*R*c* implies *a*R*c*. Most relations used in testing are transitive. Examples of transitive relations include: is connected to, is greater than or equal to, is less than or equal to, is a relative of, is faster than, is slower than, takes more time than, is a subset of, includes, shadows, is the boss of. Examples of intransitive relations include: is acquainted with, is a friend of, is a neighbor of, is lied to, has a du chain between.  Reflexive Relations:  A relation R is reflexive if, for every *a*, *a*R*a*. A reflexive relation is equivalent to a self-loop at every node.  Examples of reflexive relations include: equals, is acquainted with (except, perhaps, for amnesiacs), is a relative of.  Examples of irreflexive relations include: not equals, is a friend of (unfortunately), is on top of, is under.  Symmetric Relations:  A relation R is symmetric if for every *a* and *b*, *a*R*b* implies *b*R*a*. A symmetric relation means that if there is a link from *a* to *b* then there is also a link from *b* to *a*; which furthermore means that we can do away with arrows and replace the pair of links with a single undirected link.  A graph whose relations are not symmetric is called a directed graph because we must use arrows to denote the relation‘s direction. A graph over a symmetric relation is called an undirected graph.\* The matrix of an undirected graph is symmetric (*aij* = *aji* for all *i*, *j*).  Strictly speaking, we should distinguish between undirected graphs (no arrows) and bidirected graphs (arrow in both directions); but in the context of testing applications, it doesn‘t matter.  Antisymmetric Relations:  A relation R is antisymmetric if for every *a* and *b*, if *a*R*b* and *b*R*a*, then *a* = *b*, or they are the same elements.  Examples of antisymmetric relations: is greater than or equal to, is a subset of, time. Examples of  nonantisymmetric relations: is connected to, can be reached from, is greater than, is a relative of, is a friend of.  *Equivalence Relations:*  An equivalence relation is a relation that satisfies the reflexive, transitive, and symmetric properties. Numerical equality is the most familiar example of an equivalence relation. If a set of objects satisfy an equivalence relation, we say that they form an equivalence class over that relation. The importance of equivalence classes and relations is that any member of the equivalence class is, with respect to the relation, equivalent to any other member of that class.  The idea behind partition-testing strategies such as domain testing and path testing, is that we can partition the input space into equivalence classes. If we can do that, then testing any member of the equivalence class is as effective as testing them all.  When we say in path testing that it is sufficient to test one set of input values for each member of a branch-covering set of paths, we are asserting that the set of all input values for each path (e.g., the path‘s domain) is an equivalence class with respect to the relation that defines branch-testing paths. If we furthermore (incorrectly) assert that a strategy such as branch testing is sufficient, we are asserting that satisfying  the branch-testing relation implies that all other possible equivalence relations will also be satisfied—that, of course, is nonsense.  *Partial Ordering Relations:*  A partial ordering relation satisfies the reflexive, transitive, and antisymmetric properties. Partial ordered graphs have several important properties: they are loop-free, there is at least one maximum element, there is at least one minimum element, and if you reverse all the arrows, the resulting graph is also partly ordered.  A maximum element *a* is one for which the relation *x*R*a* does not hold for any other element *x*. Similarly, a minimum element *a*, is one for which the relation *a*R*x* does not hold for any other element *x*. Trees are good examples of partial ordering. The importance of partial ordering is that while strict ordering (as for numbers) is rare with graphs, partial ordering is common.  Loop-free graphs are partly ordered. We have many examples of useful partly ordered graphs: call trees, most data structures, an integration plan. Also, whereas the general control-flow or data-flow graph is not  always partly ordered, we‘ve seen that by restricting our attention to partly ordered graphs we can sometimes get new, useful strategies. Also, it is often possible to remove the loops from a graph that isn‘t partly ordered to obtain another graph that is. |
| THE POWERS OF A MATRIX | *Principles:*  Let A be a matrix whose entries are *aij*. The set of all paths between any node *i* and any other node*j* (possibly*i*itself), via all possible intermediate nodes, is given by  As formidable as this expression might appear, it states nothing more than the following:  1. Consider the relation between every node and its neighbor.  2. Extend that relation by considering each neighbor as an intermediate node.  3. Extend further by considering each neighbor‘s neighbor as an intermediate node.  4. Continue until the longest possible nonrepeating path has been established.  5. Do this for every pair of nodes in the graph.  *Matrix Powers and Products:*  Given a matrix whose entries are *aij*, the square of that matrix is obtained by replacing every entry with. More generally, given two matrices A and B, with entries *aik* and *bkj*, respectively, their product is a new matrix C, whose entries are *cij*, where:  A2A = AA2; that is, matrix multiplication is associative (for most interesting relations) if the underlying  relation arithmetic is associative. Therefore, you can get A4 in any of the following ways: A2A2, (A2)2, A3A, AA3.  However, because multiplication is not necessarily commutative, you must remember to put the contribution of the left-hand matrix in front of the contribution of the right-hand matrix and not inadvertently reverse the order.  The loop terms are important. These are the terms that appear along the principal diagonal (the one that slants down to the right).  *The Set of All Paths:*  This is an eloquent, but practically useless, expression. Let I be an *n* by *n* matrix, where *n* is the number of  nodes. Let I‘s entries consist of multiplicative identity elements along the principal diagonal. For link names, this can be the number ―1. For other kinds of weights, it is the multiplicative identity for those weights. The above product can be re-phrased as:  A(I + A + A2 + A3 + A4 . . . A∞)  But often for relations, A + A = A, (A + I)2 = A2 + A +A + I A2 + A + I.  Furthermore, for any finite n,  (A + I)*n* = I + A + A2 + A3 . . . A*n*  Therefore, the original infinite sum can be replaced by  This is an improvement, because in the original expression we had both infinite products and infinite sums, and now we have only one infinite product to contend with. The above is valid whether or not there are loops. Finding the set of all such paths is somewhat easier because it is not necessary to do all the intermediate products explicitly. The following algorithm is effective:  1. Express *n* – 2 as a binary number.  2. Take successive squares of (A + I), leading to (A + I)2, (A + I)4, (A + 1)8, and so on.  3. Keep only those binary powers of (A + 1) that correspond to a 1 value in the binary representation of *n*  – 2.  4. The set of all paths of length *n* – 1 or less is obtained as the product of the matrices you got in step 3  with the original matrix.  As an example, let the graph have 16 nodes. We want the set of all paths of length less than or equal to 15. The binary representation of *n* – 2 (14) is 23 + 22 + 2.  A matrix for which A2 = A is said to be idempotent. A matrix whose successive powers eventually yields an idempotent matrix is called an idempotent generator—that is, a matrix for which there is a k such that A*k*+1 = A*k*.  The point about idempotent generator matrices is that we can get properties over all paths by successive squaring. A graph matrix of the form (A + I) over a transitive relation is an idempotent generator; therefore, anything of interest can be obtained by even simpler means than the binary method discussed above. The *n*th power of a matrix A + I over a transitive relation is called the transitive closure of the matrix.  *Loops:*  The way to handle loops is similar to what we did for regular expressions. Every loop shows up as a term in the diagonal of some power of the matrix—the power at which the loop finally closes—or, equivalently, the length of the loop. The impact of the loop can be obtained by preceding every element in the row of the node at which the loop occurs by the path expression of the loop term starred and then deleting the loop term. Applying this method of characterizing all possible paths is straightforward. The operations are interpreted in terms of the arithmetic appropriate to the weights used.  *Partitioning Algorithm:*  Consider any graph over a transitive relation. The graph may have loops. We would like to partition the graph by grouping nodes in such a way that every loop is contained within one group or another. Such a graph is partly ordered. There are many used for an algorithm that does that:  1. We might want to embed the loops within a subroutine so as to have a resulting graph which is loopfree  at the top level.  2. Many graphs with loops are easy to analyze if you know where to break the loops.  3. While you and I can recognize loops, it‘s much harder to program a tool to do it unless you have a solid  algorithm on which to base the tool.  *Breaking Loops And Applications:*  Consider the matrix of a strongly connected subgraph. If there are entries on the principal diagonal, then start by breaking the loop for those links. Now consider successive powers of the matrix. At some power or another, a loop is manifested as an entry on the principal diagonal. Furthermore, the regular expression over the link names that appears in the diagonal entry tells you all the places you can or must break the loop. Another way is to apply the node-reduction algorithm (see below), which will also display the loops and therefore the desired break points.  The divide-and-conquer, or rather partition-and-conquer, properties of the equivalence partitioning algorithm is a basis for implementing tools. The problem with most algorithms is that they are computationally intensive and require of the order of *n*2 or *n*3 arithmetic operations, where *n* is the number of nodes. Even with fast, cheap computers it‘s hard to keep up with such growth laws. The key to solving big problems (hundreds of nodes) is to partition them into a hierarchy of smaller problems. |