rectness of programs," *Computing Surveys*, vol. 8, pp. 331–353, Sept. 1976.
[6] H. E. Koenig, Y. Tokad, and H. K. Kesavan, *Analysis of Discrete Physical Systems*. New York: McGraw-Hill, 1967.
[7] U. R. Kodres, "Discrete systems and flowcharts," *IEEE Trans. Software Eng.*, to be published.
[8] ——, "Logic circuit layout," in *Proc. Joint Conf. on Mathematical and Computer Aids to Design*, Oct. 1969, pp. 165–191.
[9] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 308–320, Dec. 1976.
[10] B. Shneiderman, R. Mayer, J. McKay, and P. Heller, "Experimental investigation of the utility of detailed flowcharts in programming," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 373–381, June 1977.

Uno R. Kodres received the B. A. degree in 1954 from Wartbug College, Waverly, IA, and the M.A. and Ph.D. degrees from Iowa State University, Ames, in 1956 and 1958, respectively.

While at Iowa State University he had a teaching fellowship in mathematics. He joined IBM in 1958 on the Design Automation Project at Poughkeepsie, NY. In 1963 he joined the Naval Postgraduate School, Monterey, CA, where he is presently an Associate Professor in the Department of Computer Science. In 1967 he was a consultant to IBM in Menlo Park on the Advanced Computational Systems Project. Most recently his interest is in homogeneous, distributed computer architectures making use of large-scale integrated technology. He is a coauthor of a book on design automation, has lectured at the UCLA sponsored short courses, and was invited as a Fulbright Scholar to Yugoslavia to lecture on automated design.

Dr. Kodres is a member of the Association for Computing Machinery, the Society for Industrial and Applied Mathematics, the Mathematical Association of America, and Sigma Xi.

Testing Software Design Modeled by Finite-State Machines

TSUN S. CHOW

Abstract—We propose a method of testing the correctness of control structures that can be modeled by a finite-state machine. Test results derived from the design are evaluated against the specification. No "executable" prototype is required. The method is based on a result in automata theory and can be applied to software testing. Its error-detecting capability is compared with that of other approaches. Application experience is summarized.

Index Terms—Control structure, finite-state machines, reliability, software testing, test covers, validity.

INTRODUCTION

ONE of the most difficult problems in testing is finding a test data selection strategy that is both valid and reliable [4]. The general problem is not solvable [13]. However, by restricting ourselves to look at the "control structure" of software systems only, and only those that can be modeled by a finite-state machine, we do find a testing strategy that is both valid and reliable.

This paper presents a new testing strategy that we call "automata theoretic." It has the following characteristics:

1) only the control structure of the design is checked;
2) it does not require an "executable" specification;
3) test sequences are guaranteed to reveal any errors in the control structure, provided that some reasonable assumptions are satisfied.

As far as testing the correctness of the control structure in a design is concerned, our method is superior to those existing testing strategies that are based on the structure of the design. By means of examples, we will show that there are errors that are detected by our method, while going undetected by other testing strategies.

In addition, we will define a new hierarchy consisting of "*n*-switch set covers," a generalization of the modified "switch cover." Although we prefer the automata theoretic method to *n*-switch set covers, we are able to specify *analytically* the classes of errors that can be detected by an *n*-switch set cover. Up until now, the best one could do was to obtain *empirical* data concerning the reliability of a testing strategy [13]. Of course, we remind the reader that here we are only dealing with the verification of the control structures at the design level, and only those that can be modeled by finite-state machines.

Next, our discussion proceeds to an examination of the

Manuscript received December 7, 1977; revised February 3, 1978.
The author is with Bell Laboratories, Naperville, IL 60540.

assumptions behind our automata theoretic method and how some of those assumptions can be relaxed. We also discuss whether the remaining restrictions are reasonable or practical.

We conclude by reporting our experience of applying the automata-theoretic method (manually) to test the design of three nontrivial systems in the area of computer graphics, real-time process control, and telephone switching.

"Automata Theoretic" Testing Strategy

Background

For the software systems that we shall consider in this paper, there are two kinds of primitives: stimuli and operations. Stimuli are inputs from the world outside the software system; operations are events caused by the software system by activating operations as a response to some stimuli. While not every software system can be modeled in this way, there is a large class of software systems in the areas of lexical analysis [5], data processing [14], and real-time process control, that does fall into this category. The control structure of such a system at the design level deals with how operations are sequenced as a response to stimuli. We are not concerned with the modeling or implementation of operations. In other words, design at this level only specifies the global control structure of the system. Here, we assume that such a control structure may be modeled as a finite-state machine in the sense that not only are there a finite number of states, but also, the next operation and the next state depend solely on the current state and the input. In particular, this implies that there are no control variables or counters manipulated by operations, which influence the sequencing of operations. We further assume that the machine is 1) completely specified, 2) minimal, 3) starts with a fixed initial state, and 4) every state is reachable in practice. With additional manipulation, the method can be applied to designs which may violate these assumptions.

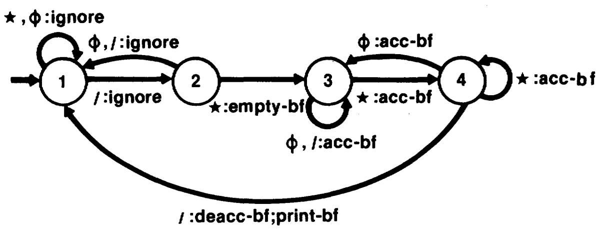
The specification against which the design shall be tested may be stated informally in the form of prose or formally as a set of axioms. Its purpose is to characterize the desired behavior of the system in terms of stimuli and operations. The important point is that we do not assume the existence of an "executable" specification, such as an operational model or a prototype. This is a salient feature of our method. In "model-referenced" testing [17], an alternate executable model must be available to test the system against. In practice, unfortunately, it is not always easy to come up with an equivalent and yet different executable model to serve the purpose of comparison.

As an illustration, throughout this discussion we shall consider a system that extracts comments from a character string. Its specification is given in Fig. 1. The corresponding design in the form of a minimal finite-state machine is shown in Fig. 2.

Comment Printer:

Specification: Input consists of characters ★, ϕ, /.
Print only comments. A comment is an input sub-sequence enclosed by /★ on the left and ★/ on the right. (It may contain other /★'s but not ★/'s.)

Fig. 1. Informal specification for a comment printing system.



X:Y - X is the input, Y is the output or operation.

Operations

ignore - nil action
empty-bf - buffer ⟵ empty
acc-bf - buffer ⟵ buffer concatenated with current input
deacc-bf - buffer ⟵ buffer with rightmost character truncated
print-bf - content of buffer printed

Note: The system variable buffer is of type character and its length is not bounded.

Fig. 2. Finite state design of comment printer.

The Method

The method consists of three major steps:

1) estimate the maximum number of states in the correct design;

2) generate test sequences based on the design (which may have errors);

3) verify the responses to the test sequences generated in step 2.

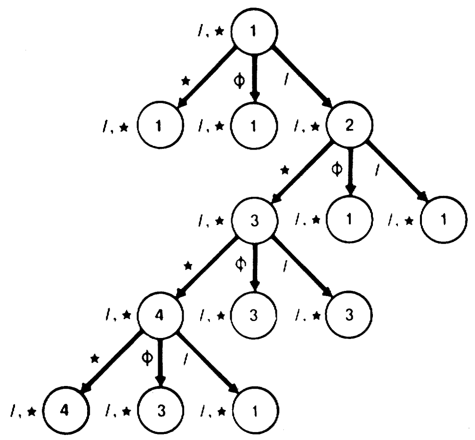
Estimation: We do not have access to the correct design. (Otherwise we would not undertake the process of designing the system in the first place.) Hence, human judgment must be used in estimating the maximum number of states in the correct design. The guess is usually based on the design at hand. We shall later comment on the effect of this step on the overall reliability.

Generation of Test Sequences: The set of test sequences required is the concatenation of two sets of sequences P and Z. In the following discussion, we shall give the "recipes" for constructing P and Z.

Construction of P: P is any set of input sequences such that for every transition from state Ai to state Aj on input x, there are input sequences p and px in P so that p forces the machine into state Ai from its initial state. One way to construct the set P is to first build a "testing tree" of A. All partial paths in the testing tree constitute the set P. A "partial path" is a sequence of consecutive branches. It starts at the root of the tree and ends at either a terminal or a nonterminal node. Each branch in the testing tree is labeled with an input symbol. Thus, P is a set of input sequences. The empty sequence is considered to be a member of P. A procedure of constructing testing trees is presented below.

	Response	
Input / State	/	★
1	ignore;	ignore
2	ignore;	empty-bf
3	acc-bf;	acc-bf
4	d-bf; p-bf	acc-bf

(a)



Test Sequences

$\{/,\star\}$
$\star \bullet \{/,\star\}$
$\phi \bullet \{/,\star\}$
$/ \bullet \{/,\star\}$
$/ \star \bullet \{/,\star\}$
$/ \phi \bullet \{/,\star\}$
$/ / \bullet \{/,\star\}$

$/ \star \star \bullet \{/,\star\}$
$/ \star \phi \bullet \{/,\star\}$
$/ \star / \bullet \{/,\star\}$
$/ \star \star \star \bullet \{/,\star\}$
$/ \star \star \phi \bullet \{/,\star\}$
$/ \star \star / \bullet \{/,\star\}$

(b)

Fig. 3. (a) State responses to the characterization set $\{/,\star\}$. (b) Testing tree and test sequences.

Given a finite-state machine A, a testing tree T is constructed as follows:

1) Label the root of T with the initial state of A. This is level 1 of T.

2) Suppose we have already built T to a level k. The $(k+1)$th level is built by examining nodes in the kth level from left to right. A node at the kth level is terminated if its label is the same as a nonterminal at some level $j, j \leqslant k$. Otherwise, let Ai denote its label. If on input x, machine A goes from state Ai to state Aj, we attach a branch and a successor node to the node labeled Ai in T. The branch and the successor node are labeled with x and Aj, respectively.

The above process always terminates, since there are only a finite number of states in A. Also, depending on the left to right order in which we place the successor nodes, a different tree may result.

Construction of Z: Let X be the input alphabet (a collection of all stimuli). Let the finite-state machine representing the design have n states in the minimal form. m is the maximum number of states that the correct version might have (see Estimation) $(m \geqslant n)$.

We use W to denote the "characterization set" of the design. Briefly, a characterization set consists of input sequences that can distinguish between the behaviors of every pair of states in a minimal automaton. For definitions and a method of constructing such sets, see [3]. The set shown in Fig. 3(a) is the characterization set for the machine in Fig. 2.

Z is defined to be the set $W U X \cdot W \cdots U X^{m-n} \cdot W$, where U is the union operator and $A \cdot B$ denotes the concatenation of the sets A and B; $A^0 = \{\text{null-string}\}, A^{i+1} = A \cdot A^i$.

Verification: There are two different ways the verification may be carried out:

1) *Test Mode*—Correct responses in the form of operation sequences are constructed first, based on the specification and the semantics of operations. This process should not be too difficult, since it only involves the system's behavior for specific input sequences. Responses based on the design are derived and compared with the correct version.

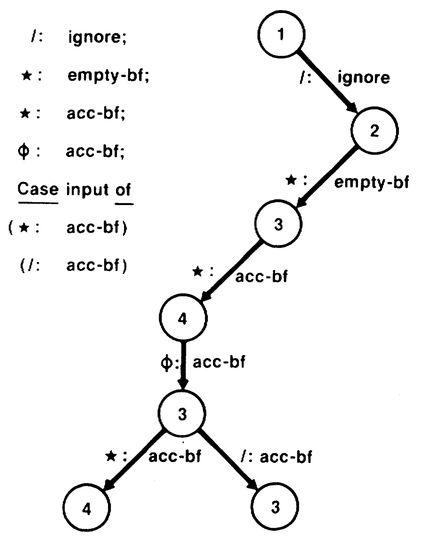


Fig. 4. Path program showing input: response pairs with respect to the test sequences /★ ★ φ · {/,★}.

2) *Walk-Through Mode*—Input sequences and their corresponding responses from the design may be represented as "path programs." The correctness of these programs may be established by a walk-through procedure based on the specification.

The reader should note that this step cannot be fully automated since we do not assume the specification to be executable. While this is not an easy step it should not be more difficult than the "walk-throughs" often used in design reviews. During this step, one may discover that there may be ambiguity or incompleteness in the specification, although the test sequences cannot reveal all possible errors in the specification.

Example: Consider the design given in Fig. 2. First, we may guess that the correct design should not have more states than the design given. Hence, $m = n = 4$. The set of test sequences is generated in two parts: set P and set Z. P consists of partial paths of a testing tree shown in the top half of Fig. 3(b). Z in this case is reduced to $W = \{/, *\}$ since $m = n$. The concatenation of P and Z is made explicit by labeling each node in the testing tree by Z. The complete set of testing sequences is shown in the bottom half of Fig. 3(b).

In the verification step, we have to decide whether or not a "path program" such as the one shown in Fig. 4 is correct. This may be accomplished by going back to the specification in Fig. 1 and the definitions of operations shown in Fig. 2.

Reliability

The testing process described in "The Method" section may be viewed as checking whether the design automation is $P \cdot Z$ equivalent to the automaton implied by the specification (the correct version). During the process, we are comparing the output sequences induced by not all input sequences but only those in the set $P \cdot Z$. According to a theorem proved in the Appendix, the two automata are equivalent, i.e., the design is error-free if and only if the two automata are $P \cdot Z$ equivalent, provided that the following assumptions are true:

1) the two automata have the same input alphabet;

2) the estimate of the maximum number of states in the automaton (correct version) implied by the specification is correct.

If we have spent adequate effort in analyzing the specification and constructing the design, it is not unreasonable to expect assumptions 1 and 2 to hold. Furthermore, Gill [3] has shown that without these assumptions there are *no* algorithms that can check whether or not the design automaton is equivalent to the automaton implied by the specification.

There are three important observations. First, the process that we have described is a test procedure, since we are only concerned with the correctness of operation sequencing with respect to a finite set of input sequences. This is easier than trying to prove the correctness of the operation sequencing in general. Second, for a given design, the set of test sequences generated by the procedure is not unique. Depending on the choice of P and Z, different sets of test sequences may be generated. Third, as long as the assumptions we made about the correct design are valid, the procedure is guaranteed to detect any sequencing errors in the design. Put another way, as a direct consequence of the theorem, one may show that the "automata theoretic" testing strategy is both "valid" and "reliable" in the sense defined in [4].

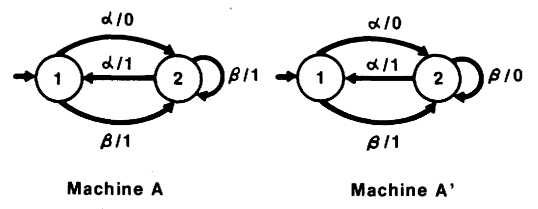
Complexity

We do not have the exact upperbound on the work required to generate test sequences. For a minimal finite-state machine with n states and k input symbols, the effort required in constructing P and W is roughly proportional to $n^2 \cdot k$. This can be seen as follows: P is obtained by first constructing a testing tree and then enumerating the partial paths in the tree. Since each arc appears exactly once in the testing tree, the complexity of the former is proportional to $n \cdot k$, the number of arcs in the machine. The complexity of the latter is also proportional to $n \cdot k$, because there are $n \cdot k + 1$ partial paths in the tree. The characterizing set W may be obtained from so-called P_k tables (see [3]). For a minimal machine, there are at most $n - 1$ such P_k tables. The amount of work required to construct a P_k table is proportional to $n \cdot k$, the number of entries in the table. Thus, the complexity of W is proportional to $(n - 1)\, n \cdot k$, or roughly $n^2 \cdot k$.

There are also simple techniques that one can use to minimize the number of test sequences. According to M. P. Vasilevskii [15], the minimal number of test sequences required is less than or equal to $n^2 \cdot k^{m-n+1}$ and the total length of all test sequences is not greater than $n^2 \cdot m \cdot k^{m-n+1}$, where n is the number of states in the design automaton, m is the maximum number of states in the automaton implied by the specification (the correct version), and k is the size of the input alphabet. In particular, for $m = n$, the respective bounds are $n^2 \cdot k$ and $n^3 \cdot k$. The reader should note that these are the best bounds in the worst case. In an average case, the number of test sequences can be much lower.

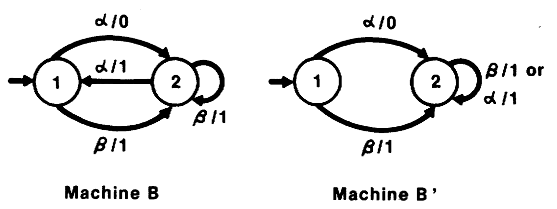
Comparison with Other Test Covers

For the purpose of comparing our testing method with other test coverages, it is convenient to divide errors in the



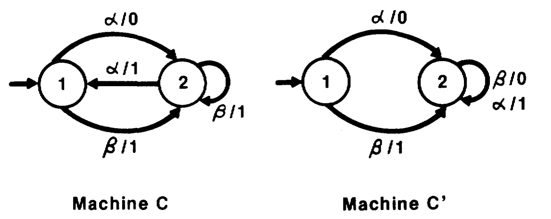
Machine A has an operation error at State 2 with input β. Such operation errors can be detected by a "branch cover" ααββ.

Fig. 5. Operation errors.



Machine B has a transfer error at State 2 with input α. This transfer error cannot be detected by a "branch cover" for A: ααββ.

Fig. 6. Transfer errors.



Machine C has a transfer error at State 2, with α as input, and an operation error at State 2, with β as input. The "branch cover," ααββ, can detect the operation error, but not the transfer error.

Fig. 7. Transfer and operation errors.

control structure into classes. A and A' are two minimal finite-state machines with the same input alphabet. A' is the "standard" or the correct version against which A is compared.

1) *Operation Errors*–A is said to have operation errors, if A is not equivalent to A' and A can be modified to be equivalent to A' by changing only the output function of A (without adding or deleting states in A). An example is shown in Fig. 5.

2) *Transfer Errors*–A is said to have transfer errors if A is not equivalent to A' and A can be modified to be equivalent to A' by changing only the next-state function of A (without adding or deleting states in A). An example is shown in Fig. 6. A combination of operation and transfer errors is shown in Fig. 7.

3) *Extra States (missing states)*–A is said to have extra (missing) states if in order to make A equivalent to A' the number of states in A must be reduced (increased). Since A and A' are minimal, an unequal number of states implies that A and A' are not equivalent.

We will refer to the above errors collectively as *sequencing errors*. In the following discussion we will use these error classes to compare the relative error-detecting capability of the automata theoretic method and other test covers. In each case, we shall indicate the class of errors that may not be detected by the other test cover, while we leave the reader to convince himself/herself that the same errors can be detected by the automata theoretic method. In all of the following examples, the primed machine is the correct version, while the unprimed machine represents the design. The primed machine is only used to simplify the verification step in the testing process. The generation and application of test sequences are restricted to the unprimed machine.

"Branch Cover"

The number of executable paths in a program with loops may be infinite. In these cases, it is not possible to test every path. A common practice [9] is to require that the test sequences form a "branch cover," i.e., every branch in the program is traversed by at least one test sequence. For a finite-state machine, a "branch cover" can detect all operation errors, but it cannot detect transfer errors. An example is shown in Fig. 6.

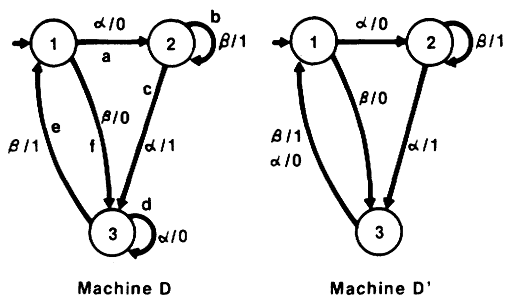
"Switch Cover"

A more stringent test coverage is called a "switch cover." A switch is a branch-to-branch pair [12]. A "switch cover" consists of test sequences so that every branch-to-branch pair in the program graph is traversed. Since, a "switch cover" is a "branch cover," it can detect all operation errors. However, even though it is more stringent than a "branch cover," a "switch cover" still cannot detect all transfer errors. The machine shown in Fig. 8(a) is 1-distinguishable[1] and yet a "switch cover" cannot detect its transfer errors.

"Boundary-Interior Cover"

Another test coverage, called "boundary-interior test cover," [8] is also used frequently in practice. Boundary-test paths refer to test paths that enter a loop without causing the loop to be iterated. An interior test path or sequence causes a loop to be entered and iterated at least once. These paths are divided into a finite number of classes. A "boundary-interior

[1]A machine is k-distinguishable if, for each pair of states, there is at least one input sequence of length k which, when applied to the pair, yields different output sequences.



Switches for Machine D:
ab, ac, bb, bc, cd, ce, dd, de, ea, ef, fd, fe

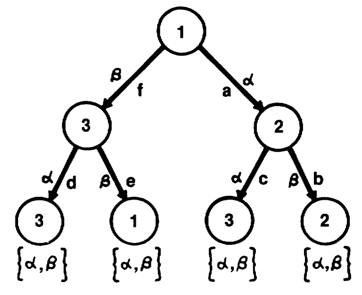
States in Machine D are 1- distinguishable.
However, Machines D and D' cannot be distinguished by the following "switch cover" for Machine D:

$\alpha\beta\beta\alpha\alpha\alpha\beta$ $\quad$ $\beta\alpha\alpha\beta$
$\alpha\alpha\beta\alpha$ $\quad$ $\beta\beta$
$\alpha\alpha\beta\beta$

(a)

1- Switch Sets for Machine D:

{ab, ac} {bb, bc} {cd, ce}
{dd, de} {ea, ef} {fd, fe}

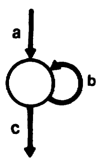


1- Switch Set Cover for Machine D:

$\beta\alpha\alpha$, $\beta\alpha\beta$
$\beta\beta\alpha$, $\beta\beta\beta$
$\alpha\alpha\alpha$, $\alpha\alpha\beta$
$\alpha\beta\alpha$, $\alpha\beta\beta$

(b)

Fig. 8. (a) Switch cover and transfer errors. (b) Switch set cover.

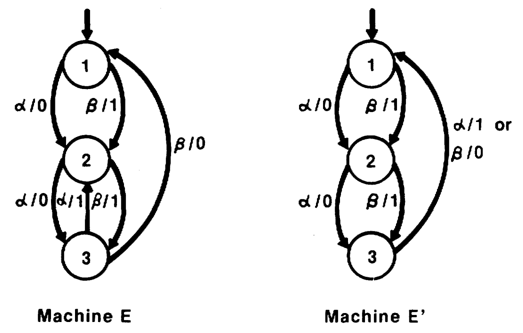


1- switches:

ac, ab
bc, bb

ac, abc form a boundary-interior cover for the figure shown. However, it is not even a 1-switch cover. In this case, a 1-switch cover is a boundary-interior cover for the loop shown.

Fig. 9. "Boundary-interior cover" versus "1-switch cover."



Machines E and E' can be distinguished by the input sequence $\alpha\alpha\alpha\alpha\alpha$.
However, they cannot be distinguished by the "H-Language" of E:

Q
Q • β • Q

$$\text{where } Q = \begin{Bmatrix} \alpha \\ \beta \end{Bmatrix} \bullet \begin{Bmatrix} \alpha \\ \beta \\ \alpha\alpha\alpha \\ \alpha\alpha\beta \\ \beta\alpha\alpha \\ \beta\alpha\beta \end{Bmatrix}$$

Fig. 10. Example shows error not detected by H-language.

cover" consists of a set of test sequences, in which at least one path from every class is traversed. Fig. 9 shows that sometimes a "boundary-interior cover" does not even form a "switch cover."

"H-Language"

A cover, similar to the boundary-interior cover,[2] called "H-language," [12] also deals with the testing of loops. An example is shown in Fig. 10. In this case, the test paths in the H-language of machine E, cannot distinguish machine E from machine E'.

It is clear from the above discussion that operation errors are easy to detect. However, existing test covers are not reliable in detecting transfer errors. We leave the reader to convince himself/herself that it is even harder to detect whether any state is extra or missing. As implied by the Section on "Reliability," the automata theoretic method can detect not only operation errors but also transfer errors, extra states, and missing states up to $m - n$ (where n is the number of states in the minimal machine that represents the design and m is the maximum number of states that the correct version might have).

Test Cover Hierarchy and Its Characterization

Test covers such as "branch cover" or "switch cover" are unsatisfactory, not only because they cannot catch all the errors, but also because there are no analytical methods to define precisely what kind of errors can be detected. In this

[2]There is a slight difference in the way a path class is defined in the H-language and a "boundary-interior cover."

section, we shall provide an analytical characterization of the classes of errors that can be detected by test covers, which we call "n-switch set covers." These covers do not involve automata theoretic concepts and are *not* proposed as an alternative to the automata theoretic method. Our purpose is to shed some light on the effect of increasing coverage on the error-detection capability.

We define an "n-switch" to be a sequence of consecutive branches or arcs in the program graph (transition diagram of a finite-state machine) of length $n + 1$. Hence, a branch is a 0-switch and Rault's switch [12] is a 1-switch.

For a fixed n, we can define for each branch b an "n-switch set" that consists of all n-switches, each having b as its prefix [see Fig. 8(b)]. We say that an n-switch set (associated with a particular branch) is covered, if there are test sequences in which all switches in the n-switch set are reached via the same path. An "n-switch set cover" is a set of test sequences in which all n-switch sets are covered. An "n-switch set cover" can be constructed quite easily from the testing tree of a machine. An example of a "1-switch set cover" is shown in Fig. 8(b).

To characterize the error-detecting power of an n-switch set cover, we need to apply the following theorem, which follows from the theorem proved in the Appendix.

Theorem: An n-switch set cover can detect all operation and transfer errors and extra states in a minimal finite-state machine which is n-distinguishable.

For our example in Fig. 8, a "1-switch set cover" will detect all sequencing errors, except missing states, since the states are 1-distinguishable.

Since an n-switch is the prefix of an $n + 1$-switch, "n-switch set covers" form a hierarchy, i.e., an $n + 1$-switch set cover is an n-switch set cover. The converse may not be true. Since the states are pairwise $n - 1$-distinguishable in any minimal finite-state machine with n states, the natural definition of a "full cover" is an "$n - 1$ switch set cover," where n is the number of states in the minimal machine.

Fig. 11 summarizes the relative error detecting power of various test coverages. Note that automata theoretic analysis is the most effective method of detecting sequencing errors. Although this method requires extensive analysis, in most cases the total length of sequences required is considerably shorter than that in other coverages.

Common Names	Test Covers	Detectable Error Classes
"branch cover"	"0-switch cover"	
	"0-switch set cover"	Operation Errors
	"1-switch cover"	Operation Errors Some Transfer Errors
	"1-switch set cover"	Operation Errors Transfer Errors among states which are 1-distinguishable
	"2-switch set cover"	
	⋮	⋮
"full cover"	"n-1-switch set cover"	Operation Errors Transfer Errors Extra States
	"automata theoretic"	Operation Errors Transfer Errors Extra States Missing States $\leq m - n$

Note: 1. n is the number of states in A.
2. The correct version is assumed to have $\leq$ m states.

Fig. 11. A hierarchy of test covers for a minimal automaton A.

Application Considerations

In this section, we shall examine the following questions.

1) How reasonable are the assumptions behind our automata theoretic method?

2) How may some of these assumptions be relaxed?

In only a very few real programs can one use a finite-state machine to model both the manipulation of data and the flow of control. There is, however, a large class of programs whose control structures can be modeled by finite-state machines in the form of an input/output transducer, [1], [2], [5], [11]. Our experience in telephone switching seems to support the same claim. Recently, for example, we took a look at the design of an on-going software project (40K words of object code completed). We found that finite-state machines were used in the design of the control structure in every module that was supposed to be machine independent. This amounted to 10 modules or 25 percent of the total object code.

The reader will agree that it is reasonable to require a machine to have no unreachable states. If the machine has more than one initial state, it may be considered to be the coincidence of multiple machines, each with a single initial state. There are standard techniques [3] to 1) minimize a machine if it is not already in a minimal form, 2) convert a nondeterministic machine to a deterministic one, and 3) complete an incompletely specified machine.

In some applications, we might find it convenient to specify the control structure in such a way that the operation and the next state depend not only on the current state and a given input, but also on the status of certain global variables. If the range of any of these variables is large, the control structure in question cannot be adequately modeled by a finite-state machine.

However, if the number of global variables and the ranges are small (binary for example), then the conditional state transfer and operation activation may be removed by a process called "unfolding" by Howard and Alexander [16].

In order to keep the amount of work involved in testing to a manageable size, it is crucial that the number of states and the input alphabet of each machine are small. It may be accomplished in two ways: 1) λ-transformation and 2) abstract inputs and outputs.

In λ-transformation, transitions involving less important inputs are changed to λ-transitions, which are eliminated by a process similar to that of removing λ-transitions from a transition graph [10]. The details may become complicated if the transformation has to be applied to a machine whose transition depends also on flag variables.

A better way of keeping the machine small is to design in terms of abstract inputs and outputs. An input in such a machine may be an abstraction of a number of related physical inputs. An output may be a sequence of operation invocations. In other words, a transition itself is an invocation of another finite-state machine. Thus, the whole design can be represented by a hierarchy of finite-state machines. Testing may be applied to one machine at a time.

Experience

In this section we shall report briefly the author's experience in applying the automata theoretic method to three systems in the area of computer graphics [1], real-time process control [11], and telephone switching. These applications, which were all worked out with pencil and paper, were only meant to demonstrate the practicality of the proposed testing method. In each of the following cases, we assumed that the correct version has the same number of states as that of the design.

A Graphics Command Language

The design and the specification were taken from [1]. The finite-state machine was already in the minimal form, with seven states and 18-input signals, although incompletely specified. It was quite easy to complete the unspecified outputs and next state. Approximately 400 test sequences are required, with an average length of four input symbols. Although the number of test sequences might seem quite large, many sequences had common prefixes and thus could be verified in groups. In this case, we found four transfer errors and four operation errors. Since each transfer error was associated with an operation error, a branch cover would also detect these errors.

A Process Control System

This example is taken from a paper [11] on real-time programming. The original design, too complex to be analyzed, was decomposed into three modules. The module that is analyzed has five states and four-input signals. It requires 48 test sequences, with an average length of three-input signals.

In this case, a very subtle sequencing error was found, which could not be detected by an arbitrary "branch cover" or "switch cover."

A System Related to Telephone Switching

This case is based on a design developed at Bell Laboratories. The original design has 23 states, 13 inputs, and 25 outputs or operations. In addition, seven binary global variables are also used to control the sequencing of operations. The design was decomposed into two modules, and only the larger module was analyzed. It has 17 states, 13 inputs, and three binary control variables. We had to apply both unfolding and λ-transformation to reduce the design to a version consisting only of 11 states, five inputs, and no control variables. It requires 135 test sequences, with an average length of four input signals.

Because of some nontechnical reasons, the verification was not completed. Only a small number of these 135 test sequences have been carefully verified. So far, no errors have been found.

Conclusion

Based on our experience, we feel that the method proposed in this paper can be a powerful testing tool for checking the correctness of the control structure at the design level of many software systems. Of course, if one is interested in testing the data-manipulation aspect of the system or the consistency between the design and the implementation, other testing approaches must be used [17]. As of now, the automata theoretic method is still under study. Eventually the process of test sequence generation will be automated. In the future such test sequences will be used to guide us in the walk-throughs during design reviews.

Appendix

Validity of Testing Strategy

We will justify the validity of the "automata theoretic" method. This is done by first proving five lemmas. Lemma 0 is needed in the proof of Lemma 3, only for the case when automaton B might have more states than automaton A. The theorem follows directly from Lemmas 1 through 4.

We use $Ai \xrightarrow{x/y} Aj$ to indicate that automaton A at state Ai responds with an output y when input x is applied and goes to state Aj. If output is not relevant, y is dropped. We use $Ai \xrightarrow{p} Aj$ to indicate that the automaton A is originally at state Ai and goes to state Aj when an input sequence p is applied. In the following, let A and B denote two automata having the same input alphabet.

Definition 1: Let Ai and Bj be two states in A and B. V is a set of input sequences. Ai is said to be *V-equivalent* to Bj, if A at Ai and B at Bj yield identical output sequences as a response to each input sequence in V. Otherwise, A and B are said to be *V-distinguishable*.

Definition 2: Let Ai and Bj be two states in A and B. Ai is said to be *equivalent* to Bj, if Ai is *V-equivalent* for any set of input sequences V.

Note that both *V-equivalence* and *equivalence* are equivalence relations. In particular, they are symmetric. There is no ambiguity when we say that states Ai and Bj are *V-equivalent* or equivalent.

Definition 3: Two *automata* A and B are said to be *V-equivalent* (*equivalent*) if, and only if, Ao and Bo are *V-equivalent* (*equivalent*). Ao and Bo are the initial states of A and B. V is a set of input sequences.

It is clear from the definitions that two automata A and B being equivalent implies A and B being *V-equivalent* for any set V consisting of input sequences.

Definition 4: An *isomorphism* from A to B is a function f which maps states in A to states in B, such that: a) f is one-one and onto b) if $Ai \xrightarrow{x/y} Aj$ is in A, then $f(Ai) \xrightarrow{x/y} f(Aj)$ is in B. A is said to be *isomorphic* to B.

Definition 5: A relation is said to be an *isomorphism* from A to B, if it is a graph of a function which is an isomorphism from A to B. If f is a function, its graph is a relation R, such that xRy if and only if $y = f(x)$.

Definition 6: If $Ai \xrightarrow{p} Aj$ and the number of input symbols in p is $l \geqslant 0$, then Aj is called the lth successor of Ai.

Definition 7: Let A be a minimal automaton and W a set of input sequences. W is said to be a characterization set if W can distinguish between every pair of states in A.

Standard techniques for constructing minimal characterization sets are available. For instance, see [3, algorithm 4.1] or [15]. However, the validity of the following lemmas does not rely on the minimality of W.

In the following, A is a minimal automaton given. A has n states with $n \geqslant 2$ and X as its input alphabet. W is a characterization set of A. B is *any* minimal automaton which also has X as its input alphabet. m is the maximum number of states that B might have, $m \geqslant n$.

We use Z to denote the *distinguishing set* Z, which is defined to be $W \cup X \cdot W \cdots \cup X^{m-n} \cdot W$. x is a member of X. P is a set of input sequences such that for every $Ai \xrightarrow{x} Aj$ in A, there are p and px in P with $Ao \xrightarrow{p} Ai$ and $Ao \xrightarrow{px} Aj$. Ao and Bo are the initial states of A and B.

Lemma 0: Suppose W-equivalence partitions the states in B into at least n classes. Z will distinguish every pair of states in B.

Proof:

Induction Hypothesis—$X^i \cdot W \cup \cdots \cup W$-equivalence partitions states in B into at least $i + n$ classes for $i = 0, \cdots, m - n$.

Induction Base—$i = 0$, true by hypothesis of the lemma.

Induction Step—Assume induction hypothesis true for $i \geqslant 0$. To show that it holds for $i + 1$. If $X^i \cdot W \cup \cdots \cup W$-equivalence has already partitioned states in B into $i + 1 + n$ classes, then it is obviously true. Otherwise, since B is minimal, there must be a pair of states in B, Bi, and Bj such that Bi and Bj are $X^k \cdot W$ distinguishable but $X^{k-1} \cdot W \cup X^{k-2} \cdot W \cdots \cup W$-equivalent, for some $k > i$. This implies that there must be a pair of states Bi' and Bj' such that Bi' and Bj' are the $(k - i - 1)$th successor of Bi and Bj and Bi' and Bj' are $X^{i+1} \cdot W$-distinguishable, but $X^i \cdot W \cup X^{i-1} \cdot W \cdots \cup W$-equivalent. Hence, $X^{i+1} \cdot W \cup X^i \cdot W \cdots \cup W$ partitions the states in B into $i + 1 + n$ classes. By induction, we have the lemma.

Lemma 1: A is equivalent to B if and only if A is isomorphic to B.

Proof: We omit the proof, since it is a well-known result.

Lemma 2: A is isomorphic to B if and only if V-equivalence is an isomorphism from A to B for some V.

Proof: The "if" part is obvious. The "only if" part is based on the observation that isomorphism implies equivalence, and equivalence implies V-equivalence.

Lemma 3: Z-equivalence is an isomorphism from A to B if and only if:

1) for every Ai in A, there is Bj in B such that Bj is Z-equivalent to Ai. In particular, Bo is Z-equivalent to Ao;

2) if $Ai \xrightarrow{x/y} Aj$, then there are B_k and B_l in B such that B_k, B_l are Z-equivalent to Ai and Aj, respectively, and $B_k \xrightarrow{x/y} B_l$.

Proof: The "only if" part is the definition of isomorphism.

"if part"

We first argue that Z-equivalence is a function. Because of 1), and $Z \supseteq W$, for every Ai in A, there is a Bj in B such that Bj is W-equivalent to Ai. Since A has n states, from the definition of W, it follows that W-equivalence has partitioned the states of B into at least n classes. By lemma 0, Z will distinguish every pair of states in B. Hence, there must be at most one state in B which is Z equivalent to a state in A.

From the definition of W and $Z \supseteq W$, Z-equivalence is one-one. Also, it is easy to see that the mapping is onto, because B has the same input set as A and, hence, according to 2), every reachable state in B is Z-equivalent to some state in A.

Lemma 4:

1) For every Ai in A, there is a Bj in B such that Bj is Z-equivalent to Ai. In particular, Bo is Z-equivalent to Ao.

2) If $Ai \xrightarrow{x/y} Aj$ then there are B_k and B_l in B such that B_k and B_l are Z-equivalent to Ai and Aj, respectively, and $B_k \xrightarrow{x/y} B_l$.

1) and 2) are true if and only if A and B are $P \cdot Z$-equivalent.

Proof:

"if part"

1) For every Ai in A, there is a p in P such that $A0 \xrightarrow{p} Ai$. Let Bj be the state reached by p in B. Since we cannot distinguish A and B with respect to $p \cdot Z$, Bj must be Z-equivalent to Ai.

2) If $Ai \xrightarrow{x/yA} Aj$, there is a p in P such that $A0 \xrightarrow{p} Ai$. We also have $p \cdot x$ in P.such that $A0 \xrightarrow{px} Aj$. Let Bk and Bl denote the states reached by p, px in B. Since we cannot distinguish A and B with respect to $p \cdot Z$ and $px \cdot Z$, Bk, Bl are Z-equivalent from Ai and Aj, respectively. Furthermore, $yB = yA$ where yB is the output of B at Bk with respect to input x.

The "only if part" follows from Lemma 3 and the fact that isomorphism implies equivalence and equivalence implies $P \cdot Z$-equivalence.

Theorem: A is equivalent to B if and only if A is $P \cdot Z$-equivalent to B.

Proof: It follows from Lemmas 1 through 4.

Acknowledgment

I wish to express my appreciation to the numerous people who have read and criticized this paper, especially R. C. Hansen, A. N. C. Kang, Y. H. Levendel, E. A. Mark, P. R. Menon, and M. K. Seagren.

References

[1] B. H. Barnes, "An automata theoretic approach to interactive computer graphics command languages," in *Applied Computation Theory: Analysis, Design, Modeling*, R. T. Yeh, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1976.

[2] D. M. Birke, "State-transition programming techniques and their use in producing teleprocessing device-control programs," *IEEE Trans. Commun.*, vol. COM-20, pp. 569-575, June 1972.

[3] A. Gill, *Introduction to the Theory of Finite-State Machines*. New York: McGraw-Hill, 1962.

[4] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," in *Int. Conf. Reliable Software*, Los Angeles, CA, 1975.

[5] D. Gries, *Compiler Construction for Digital Computers*. New York: Wiley, 1971.

[6] M. A. Harrison, *Introduction to Switching and Automata Theory*. New York: McGraw-Hill, 1965.

[7] F. C. Hennie, "Fault detecting experiments for sequential circuits," in *Proc. 5th Annu. Symp. Switching Circuit Theory and Logic Design*, Princeton, NJ, Nov. 1964, pp. 95-110.

[8] W. E. Howden, "Methodology for the generation of program test data," *IEEE Trans. Comput.*, vol. C-24, pp. 554-560, May 1975.
[9] J. C. Huang, "An approach to program testing," *ACM Comput. Survey*, vol. 7, no. 3, Sept. 1975.
[10] Z. Kohavi, *Switching and Finite Automata Theory*. New York: McGraw-Hill, 1970.
[11] H. G. Mendelbaum and F. Madaule, "Automata as structured tools for real-time programming," in *Proc. 1975 IFAC-IFIP Workshop Real-Time Programming.*
[12] S. Pimont and J. C. Rault, "A software reliability assessment based on a structural behavioral analysis of programs," in *Proc. 2nd Int. Conf. Software Eng.*, San Francisco, CA, Oct. 1976.
[13] W. E. Howden, "Reliability of the path analysis testing strategy," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 208-215, Sept. 1976.
[14] K. G. Salter, "A methodology for decomposing system requirements into data requirements," in *Proc. 2nd Int. Conf. Software Eng.*, San Francisco, CA, Oct. 1976.
[15] M. P. Vasilevskii, "Failure diagnosis of automata," *Kibernetika* (Transl.), no. 4, pp. 98-108, July-Aug. 1973.
[16] J. H. Howard and W. P. Alexander, "Analyzing sequences of operations performed by programs," in *Program Test Methods*, W. C. Hetzel, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1973.
[17] W. H. Jessop, J. R. Kane, S. Roy, and J. M. Scanlon, "ATLAS–An automated software testing system," in *Proc. 2nd Int. Conf. Software Eng.*, San Francisco, CA, Oct. 1976.

Tsun S. Chow received the B.S. degree in engineering from the University of California, Los Angeles, in 1968, and the Ph.D. degree in electrical engineering and computer science from the University of California, Berkeley, in 1972.

He is currently a member of Technical Staff at Bell Laboratories, Naperville, IL. He has previously held positions at California State Polytechnic University, San Luis Obispo, and the University of Missouri, Columbia. His research interests are in software reliability and design methodology.

Dr. Chow is a member of the Association for Computing Machinery.

Program Complexity and Programmer Productivity

EDWARD T. CHEN

Abstract—**This paper proposes a measure of program control complexity from an information theory viewpoint. A set of empirical data showing programmer productivity as a function of program control complexity is also presented. The data reveals a step-function-like contour to programmer productivity with increasing program control complexity.**

Index Terms—Control complexity, entropy, information theory, productivity.

INTRODUCTION

PROGRAMMER productivity is a controversial subject [6] which raises many questions: What does it mean? What should be measured? Does increased productivity imply reduced quality? [3], etc.

A satisfactory definition of programmer productivity can be given if an input-process-output picture of programming activity is recognized. In terms of this picture, the programmer is the processor, the input is the program specifications, and the output is a set of programs written in a good programming style [4] and in accordance with the specifications. Ideally, the finished programs will also be "proper" in that there are no infinite loops and dead code, and only one entry and one exit.

In the context of this input-process-output model for our discussion in this paper, we define a measure of programmers' productivity in terms of the number of valid source statements that are written by the programmers in a programming language during a unit period of busy time.

"Program complexity" is the least known factor in programming activity. It is not easily measured or described, and is often ignored during the system planning process [1].

The primary notion of program complexity germane to our discussion here is a notion of *quantified variety* manifested through the programs written by the programmers.

From a programmer's viewpoint, there are many factors which determine the complexity of a computer program. Among these factors are: 1) data representation and structure, 2) data volume and distribution, 3) data communication and management, 4) memory, 5) processor, 6) I/O device, 7) operating system, 8) programming language, 9) arithmetic and non-arithmetic logic, and 10) program control logic.

In a particular business environment and with a given computer hardware configuration and a specific data structure, the complexity factor which concerns us most is the program con-

Manuscript received December 7, 1977; revised January 31, 1978.
The author is with the Computer Science Division, The Travelers Insurance Companies, Hartford, CT 06115.