Occam's Razor and Program Proof by Test

by

Peter Schorer

(Hewlett-Packard Laboratories, Palo Alto, CA (ret.))
2538 Milvia St.
Berkeley, CA 94704-2611
Email: peteschorer@gmail.com
Phone: (510) 548-3827

Feb. 29, 2016

Key words: Occam's Razor, Ockam's Razor, Ockham's Razor, program testing, program proving, algorithmic information theory

Introduction

Motivation

This paper was motivated by the attempt to answer two questions:

(1) Is there any theoretical basis for programmers' confidence that testing will reveal significant bugs (errors) in a program?

We have often felt that the surprising thing about programs is not that they often have bugs, but that they have so few bugs, given their complexity — given the number of machine instructions they represent. On the other hand, in a well-known statement, Dijkstra asserted, "Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence" [8]. This statement is true in the general case, as is demonstrated below in the section, ""Elusive" Errors" on page 8. It is not true of finite-state machines since, in principle, the absence of bugs in any finite-state machine can be determined through exhaustive testing. Neither is it true of the class of program described in [16.8], a class that is larger than that of finite-state machines, but one in which depth of nesting of loops is limited. In this paper, we define a class of program larger than that of finite-state machines, and with arbitrarily deep nesting of loops, such that the absence of bugs can be determined in a finite number of tests. The class is defined in "(3) Definition of the Class by Requring Every Instruction Be Executed Over a Certain Finite Range of Inputs" on page 17. So far as we know, such a class of program has not previously been defined.

To forestall any confusion: **this paper is not concerned with practical programming techniques for avoiding errors or for detecting errors. It is not concerned with the psychology of programming.**

The second question that motivated this paper was:

(2) Can the validity of Occam's Razor be proved in a program context?

Occam's Razor is the principle that says, informally, "simplest is best". The principle is named after the medieval philosopher William of Ockham (1300?-1349?), who wrote that "entities shall not be multiplied without necessity".

As applied to the program testing to be described in this paper, the principle can be interpreted as asserting that, if two programs, p, p', consisting of x, y, instructions respectively, where $1 \leq x < y$, both produce correct results after n tests, where $n \geq 1$, then program p will most likely be correct for all inputs.

In the hard sciences, the principle is usually expressed along the lines of: given two scientific theories, one long, and one short, where "long" and "short" refer to the length of the theory when written down, then if both theories seem to explain the same set of phenomena, we should prefer the shorter theory as being more likely to be correct in the long run. Thus the fact that the Copernican heliocentric theory is shorter than the Ptolemaic geocentric theory was one argument used to argue for the truth of the former.

We shall refer to the above two questions as the Basic Questions.

Examples to Motivate Interest in the Basic Questions

Example 1

Consider the following recursive program for ordinary multiplication of two non-negative integers:

[1] $mult(x, y) == \{\textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } y + mult(x - 1, y)\}$

It is well known that addition can be performed by a finite-state machine, but multiplication cannot, because of the need to store the intermediate sums, which for arbitrarily large x or y requires arbitrarily large memory. Therefore, in principle, it would seem that we can never hope to determine if a program "like" *mult* is correct merely by subjecting the program to a finite set of tests. Yet, at the same time, it seems hard to believe that at least certain types of error in *mult* could not be detected by testing. For example, suppose that the **if** clause were, "**if** x = 0 **then** 1 **else**...". This error would appear via a test of, e.g., *mult*(0, 3). The output would be 1 (incorrect) instead of 0 (correct). Or suppose that the recursive portion were "(y + 1) + *mult*(x – 1, y)". This error would appear via a test of, e.g., *mult*(2, 1). The output would be 3 (incorrect) instead of 2 (correct).

Example 2

Consider the following recursive program for the factorial function, $f(n) = n! = n(n - 1)(n - 2)...1$:

[2] $fact(n) == \{\textbf{if } n = 1 \textbf{ then } 1 \textbf{ else } n \times fact(n - 1)\}$.

Such a program cannot be a finite-state machine because it must do multiplication. Therefore, in principle, it would seem that we can never hope to determine if a program "like" *fact* is correct merely by subjecting the program to a finite set of tests. Yet, as with *mult*, it seems hard to believe that at least certain types of error in *fact* could not be detected by testing, e.g., an error in the **if** clause such as "**if** n = 1 then 2" or an error in the recursive part such as "(n + 1) × *fact*(n - 1)".

Example 3

Suppose that someone has chosen two real numbers, $x, y, \in [0, 1)$, i.e.,

$u = 0.U_1U_2U_3...$,

$v = 0.V_1V_2V_3...$,

where: U_i, V_i, $i \geq 1$ are decimal digits.

This person will reveal the numbers to us one pair of digits at a time (up to some finite number of pairs), progressing from left to right. That is, he or she will reveal U_1, V_1, then U_2, V_2, etc. Following his or her revealing each pair of digits, we are to make one of three replies:

(1) "$u \neq v$."

(2) "I am undecided whether $u = v$."

(3) "$u = v$."

Obviously, if, for some i, $U_i \neq V_i$, we will make reply (1). However, in general, we can never make reply (3) since, no matter how large the i such that $U_j = V_j$, $1 \leq j \leq i$, it is always possible that $U_{i+1} \neq V_{i+1}$.

Suppose, however, before showing us any digits, the person tells us that u was generated by a Turing machine of x instructions and that v was generated by a Turing machine of y instructions. Furthermore, the person tells us (truthfully) that the Turing machines belong to a class of machines that are capable of expressing all computable functions, hence all computable numbers.

Would there be a sufficiently large i such that, if $U_j = V_j$, $1 \leq j \leq i$, then we would be prepared to make reply (3)?

Probably not. (Consider that there exist finite descriptions of irrational numbers, e.g., programs to generate all the digits of π.)

The question, then, is: what further restrictions would we have to impose on u and v — hence, on their descriptions (i.e., on programs to generate them) — such that, knowing only x and y in advance, we would be able to compute an i such that if $U_j = V_j$, $1 \leq j \leq i$, then we could correctly make reply (3)? In particular, would these further restrictions still permit u, v to be irrational?

Example 4

This example is obviously related to Example 3. Suppose A writes a program to generate a number having an infinite number of digits. Suppose, further, A provides a valid proof that the program in fact generates precisely that number. A shows B the program, describes the number, and shows B the proof. A now challenges B to change the number in a way that A cannot detect with a program of A's own creation. However, A is allowed to stipulate the maximum number, d, of additional instructions B is allowed to add to A's original program to accomplish this. It may be possible for B to win, namely, by adding to the program a sub-program which in effect duplicates one from the literature that is known to compute a very large number, n, and then adding further statements to A's original program which say, in effect, "**If** the input = n **then** return the following erroneous value... and terminate, **else**..." However, if A further stipulates that B's change must be of the form, "**If** the input = n..." where n must be given explicitly, digit by digit, in the program, and not computed, then perhaps A can always win, namely, by making his program check all inputs of length 1 through the length of the modified program, since B must represent n with at most d digits.

Example 5

Consider any of the now-familiar fractal drawings of trees, leaves, and other similar structures. Suppose that one of the elements in one of these drawings were changed slightly: say, a tick mark were added to one of the lines that are reproduced recursively. It seems immediately obvious that this modification would appear "everywhere": there would be no doubt that a change had been introduced into the drawing. We might be inclined to say something like, "No matter where you would look, no matter at what scale, the change would be evident." If we regard the change as an "error", then it is clearly a different kind of error than that introduced by the **if** statement in Example 4 when n is allowed to be computed, and not specified digit by digit.

Other Ways of Stating the Basic Questions

The examples suggest that what we are looking for is a class of program (larger than that of finite-state machines) in which *inductive proofs of correctness can be performed by testing*. We are looking for a class of program about which we can say, informally, "If a program in the class is correct over a few tests, then it is correct over all tests." Or, "If you know a little about the behavior of a program in the class, then you know a lot about the behavior, in fact, you know everything." We call this class of program, the *Desired Class* of program.

Stated more abstractly, our question comes under the heading: what are some of the consequences of finite descriptions of infinite sets? The finite descriptions here are programs, the infinite sets are the sets of input/output pairs representing the functions that the programs compute.

Possible Benefits of This Research

In the late sixties, Dijkstra argued that it was advisable to get rid of GoTo statements in programming. Dijkstra was particularly concerned with GoTo statements inside of loops, which under certain specified conditions cause an immediate exit from the loop, instead of allowing the loop to terminate normally. GoTo statements are therefore like the **if** statements in "Example 4" on page 4.

If we can prove that the Desired Class of program exists, and contains "useful" programs — i.e., at the very least, programs to perform the basic arithmetic functions of addition, subtraction, multiplication, division, exponentiation, and extraction of roots — then we may get an insight into why Dijkstra's advice was well-founded.

Another possible benefit from this research is that it will give us further insight into one of the basic ideas of algorithmic information theory. (See, e.g., [4], [5], [6].) This theory was initiated by Greg Chaitin in the late sixties. One of several remarkable results of this theory is the most general definition of randomness now known: informally, a number (e.g., a string of binary digits) is random if its shortest (Turing machine) description is essentially as long as the number itself. For example, a binary string arrived at by flipping a coin is typically random according to this definition. A number is non-random, or orderly, or has a pattern, if its shortest description is considerably shorter than the number itself. For example, "10" repeated a million times is non-random (it has a very short description, e.g., the one just given).

But even though the number of bits in a program can be shown to be equivalent to the number of instructions in the program (for our purposes here), Chaitin's theory, as far as we know, does not deal with one of the central questions addressed in this paper, namely, how does the number of instructions in a program (in other words, the program's length) affect the degree of repetition present in the output of the program? Or, as we put it above, What are some of the consequences of finite descriptions of infinite sets?

A third possible benefit of this research is that it might shed light on Popper's Criterion of Falsifiability, which asserts that, of two scientific theories that seem to explain the same set of phenomena, that one is to be preferred which is more easily falsifiable.

Preliminaries

What Is An Error?

We must become aware of a hidden assumption we have been making all along, and that is, that there are programs, e.g., those for multiplication and the factorial function, from which other programs, e.g., those containing elusive errors, "differ". In other words, we have been assuming that programs can be divided into two classes: correct ones and ones that are incorrect relative to the correct ones.

But "correct" and "incorrect" are notions imposed by us. It is just as possible to consider a program with errors as being "correct" and the others as being "incorrect".

Let P be a class of program. Then P defines a class of functions, F(P), namely, the class of functions computed by the programs in P. In restricting P to be the Desired Class of program, we are restricting the class of functions that we want to consider. We are saying, in effect, "the functions we attempt to compute will be all and only the the functions defined by the Desired Class." Then an error in a program *p* in the Desired Class will always mean a difference between the function computed by *p*, and the function computed by one or more other programs in the Desired Class.

Therefore, before we begin testing a program *p* in the Desired Class to see if it computes a certain function *f*, we must prove that *f* is a function computed by at least one program in the Desired Class. We will have more to say about this under "(3) Definition of the Class by Requring Every Instruction Be Executed Over a Certain Finite Range of Inputs" on page 17.

Definitions

We assume familiarity with the Turing machine and finite-state machine concepts, and with elementary computer programming. We will here repeat briefly some of the definitions of terms pertaining to these concepts. Definitions are in alphabetical order by term. In some cases, we use terms from commercial programming instead of from the formal literature, e.g., we call the *finite-control* of a Turing machine, its *program*.

computation — the sequence of *step*s from the first, in which the tape head is on the leftmost cell of the *input*, to the last, in which the tape holds the *output* and the tape head is on the last written cell.

execute — see under *instruction*.

finite-control state — the state that the *program* itself is in. Same as *program state*.

finite-state machine — a Turing machine having only a finite amount of memory (tape cells) available for computation and storage of intermediate results. Addition of two integers can be performed by a finite-state machine because the machine only needs to remember the carry after adding each successive pair of digits. But multiplication cannot be performed by a finite-state machine because the arbitrarily large set of intermediate sums must be remembered.

The function that a finite-state machine computes can be determined in a finite number of tests related to the number of its instructions. The reason is that, in computing all inputs of length 1, then of length 2, then of length 3, ..., eventually the behavior of the machine must repeat because there is only a finite amount of memory available for computation and storage of intermediate results.

An informal argument that Occam's Razor does, in fact, apply to finite-state machines is the following:

Since, in a finite-state machine there is only a finite number of possible machine states (due to the finite memory), if we start testing all input strings in lexicographical order — 0, 1, 00, 01, 10, 11, 000, 001, 010, ... — *eventually* we will go through all possible machine states and then start repeating them. So we can determine the function computed by a finite-state machine over *all* inputs (an infinite number) by making only a finite number of tests.

An upper bound on the number of inputs we have to test is directly related to the number of *instruction*s in the finite-control. For example, if we test all inputs of length 1, 2, 3, ..., $x + 1$, where x is the number of *instruction*s in the finite-control, then we know that the execution of at least one *instruction* must be repeated. The point is that it is easy to create a Turing machine T that determines, for any finite-state machine M, the sequence of inputs, in lexicographical order, that must be tested in order for M to go through all of the machine states that can occur in any computation.Suppose, now, we have two finite-state machines, one with x *instruction*s, the other with y, where $x < y$. Assume that after testing the same inputs of length 1 through k on each machine, each machine produces the correct output, where $k \leq x$. Then clearly, the maximum

number of inputs that must yet be tested is smaller for the *x-instruction* machine than for the *y*. When all inputs of length 1 through $x + 1$ have been tested, we know the complete behavior of the *x-instruction* machine, but we don't yet know the behavior of the *y-instruction* machine.

In other words, in the case of finite-state machines, the smaller machine (as measured by number of *instruction*s) reveals its errors sooner than the larger (sooner in terms of the lexicographical order of inputs. And this is what Occam's Razor asserts. For a very simple model of a finite-state machine, see "The Wheel Model of the Desired Class" on page 14.

input — In a *Turing machine*, one tape contains the input string. The tape-head is initially positioned at the first (leftmost) digit of the string. A single input string can be an encoding of any finite set of numbers, e.g., two in the case of multiplication, where the two numbers are the ones to be multiplied. See also *length of an input*.

instruction — a standard <<current state>, <next state>>-pair in a Turing machine definition. Each such pair says, in effect, "If the program is in *finite-state control state u*, and the symbol *s* is under the tape-head, then the tape-head replaces that symbol with the symbol *t*, moves one cell {to the left, to the right} and the machine enters *finite-state control state v.*"

We sometimes refer to the (finite) set of <<current state>, <next state>> pairs in the finite-control as the machine's *program*.

We say that an instruction is "executed" when the <current state> - <next state> transition occurs. The execution of an instruction is sometimes called a *step*.

length of an input — the number of digits in the *input*, i.e., the number of tape cells the *input* occupies initially.

machine state — the machine's *program state*, the contents of the tape, and the position of the tape-head on the tape.

output — the result of a *computation*; the value that the *program* computes for a given *input*.

program — the content of — i.e., the (finite) set of *instructions* in — the *finite-control* of a *Turing machine*.

program state — a set of one or more *instructions*. There are two distinguished states: a start state, in which every computation starts, and a halt state, in which every *program* terminates computation. (In other words, **all *programs* in the class of programs we are attempting to define in this paper are algorithms**.)

statement — a certain sequence of instructions. In higher-level programming languages, examples are assignment statements, **if**...**then**... statements, loop statements.

step — the execution of an instruction, although sometimes the term *step* will mean the execution of a statement in a higher-level program.

Turing machine — an idealized computer consisting of:

(1) a finite number of finitely- or infinitely-long tapes each of which has been divided into

equal-sized cells. Each cell may contain one of a finite alphabet of symbols, or be empty. Without loss of generality, we assume that the alphabet throughout this paper is {0, 1, blank}.

(2) a tape-head positioned on one cell. (Initially, one tape contains the input string and the tape-head is positioned at the first (leftmost) digit of the string.) The behavior of the tape-head is governed by a *program* contained in the *finite-control* of the machine. Every computable function can be computed by such a machine, i.e., by programs in such a machine.

Each Turing machine computes exactly one function. For some inputs, the value of the output may be undefined, i.e., the program may go into an infinite loop. This is assumed never to happen for any machine in the Class of machine we are attempting to define in this paper. That is, we assume that all machines in the Class are algorithms.

Contrary to standard practice in the literature will frequently use *program* to refer to a Turing machine.

"Elusive" Errors

Definition of "Elusive" Error

When the possibility of proving the correctness of a program by testing is mentioned, people often say, "You can't know by testing if a program is correct because it might happen that...", and then they usually give a description of what I will call an *elusive error*. One example of an "elusive" error is the **if** statement in "Example 4" on page 4, when n is allowed to be computed and d is large. Intuitively, an elusive error is one that does not occur "often" or, if it does occur often, then its first appearance, relative to the lexicographical ordering of inputs, 0, 1, 00, 01, 10, 11, ..., does not occur until a large input. An error that occurs only once over the infinite set of all inputs, is elusive if it does not occur for a "short" input, e.g., for an input like 0, 1, 00, 01, 10, 11 ..., up to, say, an input whose length, in number of digits, is less than the total number of instructions in the program. The reason for this length-of-input criterion, which does not exclude all elusive errors, will become clear as we proceed.

For another example of an elusive error, consider any program containing the following statement (the program is here written in a pseudo-Pascal language}. We assume the program takes a single integer input, n. k is a fixed integer ≥ 0.

.
.
.

IF the n'th, (n+1)'th, (n+2)'th, ..., (n+k)'th digits in the decimal expansion of π each = 5
THEN Output an erroneous value and halt ELSE
Proceed as normally.

.
.
.

Such an error we call "elusive" because its occurrence is not predictable for all n. If there exists an n such that, for all digits beyond the n'th in the decimal expansion of π, k+1 successive 5's do not occur, then the above error occurs only a finite number of times over all inputs to the above program. Otherwise, the error occurs an infinite number of times. However, no amount of

testing can reveal which is the case, or for which untested inputs an error will occur (the existence of published decimal expansions of π does not, of course, invalidate this statement in the general case). Our goal, then, is, at the least, to define a class of program in which such errors are not possible.

Types of Elusive Error

The following are several types of elusive error:

(1) Elusive error resulting from an **if** statement or **if** clause, in which the input whose output is to be erroneous, is explicitly specified digit by digit, as in "Example 4" on page 4;

(2) Elusive error resulting from an **if** statement of the type described under "Definition of "Elusive" Error" on page 8;

(3) Elusive error resulting from an **if** statement of the following type:
if the length of the input is $\leq x$, where x is the total number of instructions in the program,
then proceed so as to produce a correct output, **else** output the following incorrect value...
and terminate.

The Busy Beaver Problem

A problem that must be taken into account in our discussion of elusive errors is the the so-called Busy Beaver Problem. It has been expressed in several different forms. The one that we will use asks: does there exist an algorithm to decide, for any Turing machine, what the largest number is that the Turing machine can write on its tape and then halt? Clearly, the machine must contain any "input" it uses in its computation, for otherwise we can in effect create a countably infinite class of machines, each of which does nothing but write its input to its output and halt. Since the size of inputs is unlimited, so can the size of outputs be.

The solution to the Problem is: no, there does not exist such an algorithm. Research has shown that, even machines with only a few states are capable of writing very large numbers. Consider, for example, a machine suggested by Graham's Number (see Wells, David, *The Penguin Dictionary of Curious and Interesting Numbers*, Penguin Books, London, 1987, pp.209-210). The machine first writes a number, say, 3, on its work tape. At stage 2, it writes 3^3 on its work tape. At stage 3, it writes 3^3 raised to the 3^3 raised to the 3^3 raised to the ... raised to the 3^3 on its work tape, where the number of 3^3 's in the exponent tower is 3^3. At stage 4, it repeats this process, taking the number produced at stage 3 as its basis, and creating a tower of exponents of length that number. The process continues for some number of stages specified in the program, say, 64. Call the final number, Big Number. Then a truly devilish elusive error could be introduced into a program by the following program segment:

Compute Big Number;
If n = Big Number **then** output the following erroneous value ... and terminate **else** proceed as usual.

Here, n is the input to the program.
The memory required to store the instructions for computing Big Number would be very

small relative to the memory required to store Big Number after it had been computed. So there would be no way to know from the total memory required to store the program as a whole, that it contained an encoding of such a big number.

The Two Extremes: Programs as Tables and Programs as General Rules

Clearly, over a finite set of inputs, any program can be written as a sequence of **if** clauses. For example:

fact'(*n*) == {
if *n* = 1 **then** return 1 **else**
if *n* = 2 **then** return 2 × 1 **else**
if *n* = 3 **then** return 3 × 2 × 1 **else**
if *n* = 4 **then** return 4 × 3 × 2 × 1 **else** ...
if *n* = *c* **then** return *c* × (*c*-1) × (*c*-2) × ... × 1
}

Such a program — which we might call a program in "table form" — is easily changed. We can change any one output by simply changing a single **if** clause. That change produces no changes in the computation of other inputs. Of course, since a program can consist of only a finite number of such clauses, we can only define outputs for a finite number of inputs using exclusively **if** clauses as in the above example. Outputs for an infinity of inputs will then be undefined unless we resort to a "closed form" sub-program, e.g., of the recursive type normally used to describe the factorial function.

On the other hand, we can write a program that computes, not merely a finite, but an infinite number of outputs via a much shorter statement, e.g.,

(5) *fact*(*n*) = { **if** input = 1 **then** 1 else *n* × *fact*(*n* - 1)}

But here it seems difficult if not impossible to change just one output without introducing additional **if** clauses and/or statements. It seems that, to change just one output we must increase the size of the program, i.e., the number of instructions, namely, by adding an **if** clause. A programmer might say that one or a few exceptions are *expensive* in terms of memory usage.

The Last-Pass Problem

People sometimes argue, "But the program can always make a last pass over the entire output tape and introduce unpredictable changes." On the other hand, at least some programs that are not finite-state machines, can produce final output "as they go", e.g., in the case of multiplication, the least significant digits of the product can be written as soon as the least significant digits of the summands have been determined.

Expressing this another way: suppose someone reveals the least significant digits of two long numbers to be multiplied, and the least significant digit of the product. The person asks us if we can tell how big the numbers are. Of course we can't. The person now reveals the next-least-significant digits of the numbers, and the next-least-significant digit of the product. Again the person asks the same question, and again, of course, we can't answer. In fact we can't answer until we come to the most significant digit of the larger of the two numbers.

Reasons for Believing the Desired Class of Program Doesn't Exist

In this section, we list some of the reasons for believing that the desired class of program doesn't exist. In the next section, we will reply to each of these reasons.

Reason 1 — Non-existence of machine to tell if two programs produce the same number

This Reason addresses Examples 3 and 4 under "Examples to Motivate Interest in the Basic Questions" on page 2. We use two theorems and a plausible assertion to state the case for this Reason.

Theorem 1: There does not exist a Turing machine *M* which, given any two machines M_1, M_2 as input, will return 1 if the two machines compute the same number, and 0 if not. The number may have a finite or infinite number of digits.

Proof of Theorem 1: (supplied by a friend)

Assume there exists such a machine *M*. We can now build another machine, *N*, which solves the Halting Problem[1] as follows:

N takes as input any program *P* and any input *I* to *P*.

N contains a program *U* which generates all the digits of π.

N also contains two programs *W* and *W*′. *W* simply contains a copy of *U*. *W*′ also contains a copy of *U*. *W*hen it is given *P* and *I*, *W* initiates the computation of *P* on *I*. Every time *P* executes an instruction, the next instruction in the generation of the digits of π by the *U* contained in *W*′ is executed.

We now use, in *N*, the assumed machine *M* to determine if *W* and *W*′ both compute π. If the answer is yes (i.e., 1), then we know that the machine *P* computing the input *I* never halts. If the answer is no (i.e., 0), then we know that the machine *P* computing the input *I* does halt (leaving an infinite number of blank cells on its tape).

Thus the assumption of the existence of the machine *M* implies that the Halting Problem is solvable, which is false. Hence no such machine *M* exists. □

Theorem 2: There does not exist a Turing machine *M* which, given any two machines M_1, M_2 as input, and the number of instructions in each machine, will return 1 if the two machines compute the same number, and 0 if not. The number may have a finite or infinite number of digits.

Proof of Theorem 2: The existence of such an *M* implies the solvability of the Halting Problem by a proof similar to that for Theorem 1. In fact, it is not necessary to give *M* the number of instructions in M_1 and M_2, since that is easily determinable by a machine. □

Plausible Assertion: It is not necessarily the case that one or more human beings, given the complete text of two programs, will be able to prove, with or without the aid of computers, within a pre-specified time, whether or not the two programs compute the same number. The number may

1. The Halting Problem asks if there exists a Turing machine U which, given as input any machine M and any input S to M, will return 1 if M eventually halts in computing S, and 0 if not. Alan Turing (after whom Turing machines are named) proved around 1936 that no such machine exists.

have a finite or an infinite number of digits.

Argument for Plausible Assertion:

Let *P* be a program which returns 1 if any given conjecture, e.g., the Riemann Conjecture, is true, and 0 otherwise. Let *P′* be a program which always returns 1.

For each conjecture, *P* is given:

- a representation of all the axioms considered sufficient to prove the conjecture;
- a representation of first-order predicate logic;
- whatever additional logical rules are considered sufficient to prove the conjecture.

Considering these representations as a formal grammar, *P* then generates all strings that can be generated in one step (replacement of variable in the grammar by another string as specified by the grammar), then all possible strings that can be generated in two steps, then all possible strings that can be generated in three steps, etc. After each step, *P* checks to see if the result is a representation of the specified conjecture. If so, then *P* halts with 1. If the result is a negation of the conjecture, then *P* halts with 0.

Now since neither the conjecture nor its negation may be provable, clearly there does not exist an algorithm to determine if the number computed by *P* equals the number computed by *P′*. (We assume that at the time the programs are run, it has not been proved that the conjecture or its negation is provable. If so, we can use any other conjecture for which this has not been proved.)

(End of argument for plausible assertion)

Reason 2 — Unpredictability of outputs for large inputs

This argument is often expressed along the lines of, "Even if your program tests correct for all 'small' inputs, you can never know, through testing, what the program will do for large inputs."

Reason 3 — A paper by Tsichritzis

Perhaps the most serious reason for doubting that the Desired Class of program contains many programs of interest is a 1970 paper [16.8] by Tsichritzis. That paper says, in essence, that a subclass of the class of program called "Loop programs" that compute all and only the primitive recursive functions, can only have the desired testing property if the programs in the subclass have at most one loop.Here is the definition of a Loop program from Meyer and Ritchie's original paper [13.05]. It will be referred to under "Reply to Reason 3 — A paper by Tsichritzis" on page 13:

"A Loop program is a finite sequence of instructions for changing non-negative integers stored in registers. There is no limit to the size of the integer which may be stored in a register, nor any limit to the number of registers to which a program may refer, although any given program will refer to only a fixed number of registers.

"Instructions are of five types: (1) X = Y, (2) X = X + 1, (3) X = 0, (4) LOOP X, (5) END, where 'X' and 'Y' may be replaced by any names for registers.

"The first three types of instructions have the same interpretation as in several common languages for programming digital computers. 'X = Y' means that the integer contained in Y is to be copied into X; previous contents of X disappear, but the contents of Y remain unchanged. 'X = X + 1' means that the integer in X is to be incremented by one. 'X = 0' means that the contents of X are to be set to zero. These are the only instructions which affect the registers.

"A sequence of instructions is a Loop program providing that type (4) and type (5) instructions are matched like left and right parentheses. The instructions in a Loop program are normally

executed sequentially in the order in which they occur in the program. Type (4) and (5) instructions affect the normal order by indicating that a block of instructions is to be repeated. Specifically if **P** is a Loop program, and the integer in X is x, then 'LOOP X, P, END' means that **P** is to be performed x times in succession before the next instruction, if any, after the END is executed; changes in the contents of X while **P** is being repeated to not alter the number of times **P** is to be repeated. The final clause is needed to ensure that executions of Loop programs always terminate. For example, the program

LOOP X
X=X + 1
END

is a program for doubling the contents of X, rather than an infinite loop. Note that when X initially contains zero, the second instruction is not executed." .

Reasons for Believing the Desired Class Does, in Fact, Exist

Reply to Reason 1 — Non-existence of machine...

The theorems and the plausible argument in no way exclude "useful" programs from being in the Desired Class, if it exists. It may be possible to show that, for any two programs *in the Class* that each generate a number, it can be determined by testing whether or not they generate the same number.

Reply to Reason 2 — Unpredictability of outputs...

The argument is true in the general case if for no other reason than that we can never know — or, rather, we can never write a program to determine — if the program even halts for all inputs. The question here is, are the outputs unpredictable if the program has no **if** statements of the type discussed under ""Elusive" Errors" on page 8?

Reply to Reason 3 — A paper by Tsichritzis

What is not clear from the definition of Loop programs, but what can be easily shown (see definition of Loop programs under "Reason 3 — A paper by Tsichritzis" on page 12), is that recursive functions, hence Loop programs, *allow elusive errors!* In other words, recursive functions include functions with **if** statements of the type described under ""Elusive" Errors" on page 8.

A Trivial Example of the Desired Class

Consider the following class F of programs:

Program f_0 in F computes the factorial function via the recursive program $f_0(n)$ = {**if** $n = 1$ **then** 1 **else** $n \times f_0(n\text{-}1)$}. (This function cannot be computed by a finite-state machine).

Program f_1 in F is exactly the same as f_0 except that in addition it contains the statement, "**If** input = 1 **then** output 0 and terminate execution".

Program f_2 in F is exactly the same as f_0 except that in addition it contains the statement, "**If** input = 2 **then** output 0 and terminate execution".

...
etc.

Clearly, the function computed by each f_k, $k > 0$, differs from the factorial function in exactly one output.

Now if we are given any two programs, f, f', in the class F, then an upper bound on the size of the input we need to test is given by the number of bits in the larger of the two programs because the exception input specified by the **if** statement (if it is present) must have fewer bits than this total number of bits.

Attempts at a Definition of the Desired Class

The Fundamental Importance of Program Length, i.e., Number of Instructions

The insight that we are convinced is essential for any progress in answering the Basic Questions is that, if a program has x instructions, then over any successive $x + 1$ instruction executions, or *steps*, the execution of at least one instruction must occur twice. Informally, "necessary repetition of instructions implies lack of arbitrariness of behavior in a program." This necessary repetition of instructions over a sufficiently long sequence of steps also seems to us to be a link to algorithmic information theory (see, e.g., works by Chaitin under "References and Bibliography" on page 22).

The Wheel Model of the Desired Class

We can model the above insight as follows. (This sub-section is excerpted from [16.7].)

Consider a class of wheel such that each wheel has a strip of evenly-spaced characters wrapped around its circumference. If the wheel is rolled in a straight line across a piece of paper, the sequence of characters will be printed. The distance between characters is the same on all wheels. A starting point, i.e., a starting character, as well as a direction of rotation, is indicated on each wheel. We shall speak of the ith character in a printed sequence, $i \geq 1$.

Assume a wheel has $k \geq 1$ characters on its printing strip. We make the following observations.

- If we do not know what the actual characters on the printing-strip are, we must wait until the wheel has made one revolution across the paper before we can predict what the ith character in a printed sequence will be.
- In particular, if, during the printing of the first i characters, $1 \leq i < k$, we observe a pattern, e.g., "...0 1 0 1 0 1" we cannot assume that the pattern will persist through any arbitrarily long sequence until k characters have been printed.
- In the case of two wheels of different diameters such that the smaller wheel has a printing-strip of length k characters and the larger wheel a strip of length m characters, $k < m$, the larger wheel must print more characters (namely, m) before we can predict with certainty what the ith character of any sequence printed by that wheel may be, $i \geq 1$.
- Assume we are testing two wheels of different diameters to see if they print the same sequence as that of a third wheel. Assume each of the two wheels contains a single error, i.e., erroneous character relative to the test wheel's sequence. Then the smaller wheel's error will appear more often in any sufficiently long sequence than will the error in the larger wheel's. Hence if we examine the ith character in the sequences printed by the two wheels, where $i \geq 1$ is chosen at random from a sufficiently large, finite, range of i's, this character is more likely to be the erroneous character in the case of the smaller wheel's sequence.

All Programs In the Class Must Be "Like" Those for Multiplication and Factorial

In studying the following attempts at a definition of the desired class, the reader should keep

in mind that our goal is a class of programs that are "like" the standard recursive programs for multiplication and for the factorial function. These programs can be written without explicit **if** statements. For example, the factorial function can be written:

$fact(n) = \{fact(1) = 1; fact\ n = n \times fact(n\text{-}1)\}$

Furthermore such programs have a property that can be expressed informally as: computations of large numbers repeat computations of smaller numbers. Thus, for example, the last 10 recursions of fact (12) are the same as the last 10 recursions of fact (100).

Attempts at a Definition of the Desired Class

A number of definitions have presented themselves in the course of our attempts to define the desired class of program. We review the most significant ones here.

(1) Definition of the Class by Pairs of Programs Having a Certain Property

A seriously flawed version of this definition appeared in [16.5] and [16.7]. The class of program so defined was there called O1-Class, "O" denoting Occam's Razor, and 1 denoting that this was the first of a possible sequence of classes, each more general than the previous. We give a brief, improved version of the definition here.

Let p, p', be programs that implement algorithms, i.e., programs that halt for all inputs. Let the number of instructions in p, p', be x, y, respectively, $x, y \geq 1$. Without loss of generality, we assume that p and p' always conclude a computation with the execution of the same instruction, or, more precisely, if $inp(j)$, $inp(h)$, $j, h \geq 1$, are different inputs, then the last instruction $inst(j,last)$ executed by p in the computation of $inp(j)$ = the last instruction $inst(h, last)$ executed by p in the computation of $inp(h)$. And similarly for p'.

We now define the process for determining if p, p' constitute an *O1-Class pair*. A pair of programs must be an O1-Class pair before they can be tested to see if they both compute the same function. Informally, each program in an O1-Class pair executes at least once in the computation of at least one input in the set of inputs of length 1, 2, 3, ..., $(x + 1) \bullet (y + 1)$, every instruction it "can" execute in the computation of any input.

The definition of an O1-Class pair is as follows.

Let $inp(j)$ denote the jth input, $j \geq 1$, in the lexicographical ordering of inputs, 0, 1, 00, 01, 10, 11, ...

Now for each input in the set $\{inp(j) \mid 1 \leq \mid inp(j) \mid \leq (x + 1) \bullet (y + 1)\}$, form the following list of ordered pairs $<inst(p, j, k>, inst(p', j, k)>$, $k \geq 0$:

$inst(p, j, 0)$ is the instruction in p that is executed last in the computation of $inp(j)$;
$inst(p, j, 1)$ is the instruction in p that is executed next to last in the computation of $inp(j)$;
$inst(p, j, 2)$ is the instruction in p that is executed third from last in the computation of $inp(j)$;
...
and similarly for $inst(p', j, k)$;

Each list terminates when k represents the first instruction executed by either or both programs in the computation of $inp(j)$.

Call the set of the above lists for all inputs in the set $\{inp(j) \mid 1 \leq \mid inp(j) \mid \leq (x + 1) \bullet (y + 1)\}$, AP ("Actual Pairs").

Now since p, p' are assumed to implement algorithms, every possible sequence of instruction executions by p (and similarly for p') can be represented as a tree, in which the root is the last instruction executed in a computation. (This instruction is always the same, as stipulated above.) Denote this tree T(p) in the case of p, and T(p') in the case of p'.

We can now form ordered pairs of instructions as follows.

$<inst(p, 0), inst(p', 0)>$ is the ordered pair of instructions that are the last instructions executed in every computation by p, p', respectively. We say that this pair is at the 0, or root, level.

We now proceed recursively. For each ordered pair of instructions at level k, $k \geq 0$, form every ordered pair of instructions $<inst(p,(k + 1)), inst(p', (k + 1))>$ such that:

$inst(p,(k + 1))$ can immediately precede an instruction $inst(p,(k))$ in an ordered pair $<inst(p,(k)), inst(p', (k))>$,
and such that
$inst(p', (k + 1))$ can immediately precede the instruction $inst(p', (k))>$ in the same ordered pair,
where "can immediately precede" means on the basis of the program of p (or p').

We halt this recursive process when every ordered pair at some level has already appeared at an earlier level. We are guaranteed that this will occur because (1) there are only a finite number of instructions in the programs of p and p', and (2) p, p' are both assumed to be algorithms, hence contain no infinite loops.

Call the resulting set of ordered pairs (at all levels), PP ("Actual Pairs").

Then if the set of ordered pairs in AP = the set of ordered pairs in PP, the pair of programs p, p' is an O1-Class pair.

It is easy to see that if p, p' form an O1-Class pair, then we can tell directly from AP if the two programs compute the same function.

(2) Definition of the Class by Nested Finite-State Machines

We can define the Class by having each program consist of nested finite state machines. For example:

```
for i1 := 1 to input1 do
    begin
    <finite state machine 1>
     for i2 := 1 to input2  do
          begin
          <finite state machine 2>
          for i3 := 1 to input3 do
               begin
```

```
            <finite state machine 3>
            for  i4 := 1 to input4 do
                begin
                <finite state machine 4>
               end
           end
      end
 end
```

In this program, the $input_i$, $1 \le i \le 4$, are the inputs; <...> denotes a comment, i.e., text that is not executed. But regardless of the number of nested finite-state machines, the program is nevertheless a finite-state machine, hence does not meet the requirements of the Desired Class.

(3) Definition of the Class by Requring Every Instruction Be Executed Over a Certain Finite Range of Inputs

Our first attempt along these lines was as follows: Let p be a program in the Class, and let p consists of x instructions, $x \ge 1$. Let I denote the set $\{inp(j) \mid inp(j)$ is the jth input, $j \ge 1$, in the lexicographical ordering of inputs 0, 1, 00, 01, 10, 11, 000, 001, ..., and $1 \le |inp(j)| \le x + 1$, where $|inp(j)|$ denotes the length, i.e., number of digits, in $inp(j)\}$. Then for each instruction *inst* in p, *inst* is executed at least once in the computation of the inputs in I.

This definition is not satisfactory because it permits programs to be in the Class that consist merely of tables (see"The Two Extremes: Programs as Tables and Programs as General Rules" on page 10), and hence are undefined for an infinity of inputs.

Our current attempt is as follows: Let p be a program in the Class and let p consists of x instructions, $x \ge 1$. Then:

each instruction is executed at least once in the computation of all strings of length $x + 1$, and
each instruction is executed at least once in the computation of all strings of length $x + 2$, and
each instruction is executed at least once in the computation of all strings of length $x + 3$, and
...

Plausibility Argument that the Class Contains No Elusive Errors:

Assume the contrary, namely, that there exists a program of length x that does not reveal an error until an input n of length $x + k$ is tested, where $k >> 1$. Without loss of generality, we can require that n be the (numerically) smallest such input.

But this assumption implies that at least one instruction in the program is wrong, i.e., writes an erroneous symbol on the machine's tape. But then if that instruction is wrong in the test of the input n of length $x + k$, it must have been wrong in the computation of an input of length $< x + k$. But this contradicts our assumption that the program does not reveal an error until an input n of length $x + k$ is tested, where $k >> 1$ (End of plausibility argument)

Remark: Observe that our Definition prevents the argument being made that one or more instructions might be executed for the first time in the set of computations of all inputs of length $x + k$, $k >> 1$, and that if these instructions were changed appropriately, then the program would be correct. Perhaps it would be, but such a program would clearly allow the presence of elusive errors.

Does the Class Defined in (3) Answer the Basic Questions?

Let us see if the Class we have just defined, answers the Basic Questions we set forth under "Motivation" on page 2. These questions are:

(1) Is there any theoretical basis for programmers' confidence that testing will reveal significant bugs (errors) in a program?

and

(2) Can the validity of Occam's Razor be proved in a program context?

Naively, it would appear that the answer to both these questions is yes. For, in reply to Question (1), we might say, "For any program p in the Class, after all inputs of length $x + 1$ have been tested, where x is the number of instructions in the program, we will know if p is correct or not." And in reply to Question (2), we might say, "Clearly, if program p has x instructions, and program p' has y, and $x < y$, then if p, p' both contain an error we will know that this is so for p in fewer tests (namely, in tests of at most all inputs of length 1, 2, 3, ..., $x + 1$) than we will for p' (which may require tests of all inputs of length 1, 2, 3, ..., $y + 1$).

But we are neglecting what we said under "What Is An Error?" on page 5, namely, that we assume that the functions that programs in the Desired Class will be tested against are limited to functions computed by programs in the Class. So we must ask what the length z of the minimal length program p'' is that p and p' are being tested against.

Clearly, further investigation is necessary to determine exactly what the requirements are of the function — hence the programs computing it — that one or more programs in the Desired Class are being tested against.

The "Usefulness" of the Desired Class as Defined in Definition (3)

As stated under "Possible Benefits of This Research" on page 5, by the Desired Class being "useful" we mean that the Class contains, at the very least, programs to do addition, subtraction, multiplication, division, exponentiation, and extraction of roots. But we hope that the Class includes many other programs besides, e.g., programs that compute functions that are constructed from programs that compute these basic arithmetic functions. In this section, we consider a few possibilities.

The Desired Class and Traditional Categories of Computable Functions

Clearly, a question that must be answered is: what is the relationship between the Desired Class and primitive recursive, general recursive and partial recursive functions?

The Desired Class and Fermat's Last Theorem

Fermat's Last Theorem asserts that, for all integers $x, y, z \geq 1$ and for all integers $n > 2$:

[1] $x^n + y^n \neq z^n$

The Theorem was finally proved by Andrew Wiles in the mid-1990s, some 300 years after Fermat first set forth the conjecture that it was true (and claimed to have a proof of that fact, although none was ever found among his papers). Interest remains, in a small segment of the mathematical community, as to whether Fermat really did have a proof, and, whether he did or did not, if there is a "simple" proof of the Theorem.

It might be possible to prove the Theorem as follows. Create a program $p_1(x, y, z, n)$ that, for each x, y, z, n, computes the left-hand side and the right-hand side of [1] and then, if the two sides are unequal, returns a 1, otherwise returns a 0. Create another program $p_2(x, y, z, n)$ that returns 1 for all inputs x, y, z, n. Now if these programs are in the Desired Class, it should be possible, in a finite number of tests, to determine if the programs p_1 and p_2 both compute the same function, namely, the function which returns 1 for all inputs x, y, z, n. If the programs both compute this function, then we have another proof of Fermat's Last Theorem.

Additional Thoughts

Topology and Testability

In point-set topology, topological spaces are defined by what are known as "separation axioms". Very roughly speaking, these axioms establish how "separated" any two points in the space are — or, more precisely, how separated each point and its neighborhoods is separated from each neighboring point and its neighborhoods. Thus (very intuitively) the points in some topological spaces may be likened to droplets of mist ("hardly separable"), the points in other spaces to grains of sand ("somewhat separable"), and the points in other spaces to gravel stones ("clearly separable").

It is possible to define topological spaces each of whose points is a function, e.g., a computable function. The value of topology in the theory of computation became clear in the late sixties through the work of Dana Scott and Christopher Strachey on the mathematical semantics of programming languages. Scott's ideas included a definition of continuous programs — programs having the characteristic that, (informally) if the amount of information in the input increases, then the amount of information in the output must increase also.

We believe that this is the correct point of view from which to approach the question of testability. Basically, our goal is to find a space of computable functions (hence programs) which, on the one hand, will guarantee that we will always be able to determine in a finite number of tests which function a given program computes, while on the other hand will be general enough for us to accomplish the computation tasks at hand. More precisely, let P be a class of program and let p, p' be programs in P. We would like to be able to answer the question, "How *far* is p' from p?", or, at the least, given a third program p'' in P, we would like to be able to answer the question, "Which is farther from p: p' or p''?"

Some Ways that Programs Can Differ

In its greatest generality, our question can be informally stated as follows: "How different from each other are the computable functions?" Specifically, "Are there non-trivial subsets of the computable functions such that, if we test two programs that are known to compute functions in the set, we will know in 'very few' tests whether the programs compute the same function?"

Let us begin by simply listing some ways that programs can differ.

Two programs can differ:

- In their outputs (in other words, in the functions they compute). The problem is that knowing that p and p' differ in k outputs, $k \geq 1$, or in an infinity of outputs, does not in itself tell us how far apart the differing outputs occur over the sequence of lexicographically ordered inputs, nor does it tell us for which input the first differing outputs occur.
- In the number of their instructions. As the number of instructions increases, so

does the potential number of programs having a given number of instructions.
- In the computational complexity of the algorithms they embody.
- In the number of instructions in the programs of *minimal* length that compute the same functions as the programs in question.
- In accordance with the Scott topology.

Continuity of Syntax and Semantics

Suppose a programming language, hence a set of programs, existed which had the property that syntax and semantics were "continuous". By this we mean that, given a program p, and knowing the function $f(p)$ that p computes, and then given a "slightly different" program p', one could then be confident that the function f' computed by p' would also be only "slightly different" from f. An analogy would be a natural language in which the meanings of words in the dictionary varied in accordance with the words themselves. Thus, for example, the meanings of all the words beginning with "b" and differing in only a letter or two, would be very similar. An example of — not a programming language per se, but a formal grammar — in which syntax and semantics can be said to be "continuous" is the set of sequences of finite binary strings in the symbols 0, 1. Here, each such string can be interpreted as defining a path through an infinite binary tree, where 0 denotes "take the right-hand branch from the current node" and 1 denotes "take the left-hand branch from the current node". Thus the string 01101 denotes the path beginning at the root and then proceeding down the right-hand branch to the next node, then down the left-hand branch to the next node, then down the left-hand branch ... etc. Given two strings of length n that differ in, say, only the i'th symbol, $1 \leq i \leq n$, then it may be reasonable to say that the paths they define are only "slightly different". But if we view the semantics of the strings as being binary numbers, then it is *not* reasonable to say that two strings of length n that differ in, say, only the i'th symbol, $1 \leq i \leq n$, define numbers that are only "slightly different", as the reader can see by considering, e.g., 01101 and 01100 (the corresponding numbers differ by 1) vs. 01101 and 11101 (the corresponding numbers differ by 32).

"Prickly" Spaces

If a function known as a homeomorphism exists between two objects in a topological space, then, topologically speaking, the objects are equivalent. Either one of the objects can be thought of as being continuously deformable into the other (i.e., without tearing or gluing). To use a popular example, a typical coffee cup is topologically equivalent to a donut because a coffee cup can be continuously deformed into a donut shape. But for testing purposes, we want function spaces in which, *if functions differ at all, they differ a lot*. We want to do as little testing as possible to determine if two programs compute the same function. We can call such topological spaces, "prickly". To use our table analogy: if all input-output pairs in a program are implemented as separate rows in a table, then we can infer nothing about other rows in the table if we know a single row in the table. All are independent. Such programs define a topological space which is not prickly at all. Two programs can differ "as little as possible", namely, by just one input-output pair (over the finite domain of the program's inputs). At the other extreme are programs in which, say, every instruction is executed in the computation of every input, over all inputs 0, 1, 00, 01, 10, 11, 000, 001, ... Such programs are prickly indeed, because if two programs differ at all, they differ "a lot"! We believe that the class of program described under "(3) Definition of the Class by Requring Every Instruction Be Executed Over a Certain Finite Range of Inputs" on page 17, is sufficient prickly to accomplish the basic testing goal, while yet still permitting useful programs

— e.g., those that do multiplication, division, exponentiation, finding of roots — to be in the Class. But it must be pointed out that we have been making an assumption here, namely, that the "prickly" property we desire can, in fact, be established by a topology. If anything, we want just the opposite of what a homeomorphism guarantees. We do *not* want it to be possible to continuously deform one function into another. We do not want functions to be that "close", that similar! So research is needed to determine if a topology is the structure we are seeking.

References and Bibliography

[1] W. Richard Adrion, Martha A. Branstad, and John C. Cherniavsky, "Validation, Verification, and Testing of Computer Software", *National Bureau of Standards Special Publication 500-75*, U.S. Government Printing Office, Washington, D.C., (1980).

[2] Timothy A. Budd, Richard J. Lipton, Frederick G. Sayward, "The Design of a Prototype Mutation System for Program Testing", *Proceedings of 1978 National Computer Conference*, (1978).

[3] Arthur W. Burks, *Chance, Cause, Reason*, Univ. of Chicago Press, Chicago, Ill., (1977).

[4] Gregory Chaitin, "On the Difficulty of Computations", *IEEE Transactions on Information Theory*, IT-16, No. 1, 5-9, (1970).

[5] Gregory Chaitin, "Information-Theoretic Limitations of Formal Systems" JACM, 21, 403-424, (1974).

[6] Gregory Chaitin, *Algorithmic Information Theory*, IBM J. Res. Develop., 350-359, (July 1977).

[7] Richard A. De Millo, Richard J. Lipton, Alan J. Perlis, "Social Processes and Proofs of Theorems and Programs", *CACM*, 22, 271-280, (1979).

[8] Edsger W. Dijkstra, "The Humble Programmer", *CACM*, 15, 859-866, (1972).

[9] William Feller, *An Introduction to Probability Theory and its Applications*, 1, John Wiley and Sons, New York, (1968).

[10] A. Fremantle, *The Age of Belief*, New American Library, New York, (1954).

[11] John E. Hopcroft, Jeffrey D. Ullman, *Formal Languages and their Relation to Automata*, Addison-Wesley Publishing Co., Menlo Park, California, (1969).

[12] J. C. Huang, "An Approach to Program Testing", ACM Computing Surveys, 7, 113-128, (1975).

[13] Ernst Mach, "The Economy of Science", in The World of Mathematics, ed. James R. Newman, 3, Simon and Schuster, New York, 1787-1795, (1956).

[13.05] Meyer, Albert R., and Ritchie, Dennis M., "The complexity of loop programs", *Proceedings A.C.M. National Meeting, 1967*

[13.1] Gerald Oster and Yasunori Nishijima, "Moire Patterns", Scientific American, May, 1963, pp. 54-63.

[14] Karl R. Popper, *The Logic of Scientific Discovery*, Harper and Row, New York, (1968).

[15] Hartley Rogers, *Theory of Recursive Functions and Effective Computation*, McGraw-Hill, New York, (1967).

[16] Bertrand Russell, *A History of Western Philosophy*, Simon and Schuster, New York, (1972), p. 472.

[16.5] Peter Schorer, "On 'O1-Class' Computer Programs", *International Journal of Computers & Software Engineering*, Vol. 11, No. 2/3, 1984, pp. 109-114.

[16.7] _____, *Shaving With Occam's Razor*, Occam Press, San Jose, Calif., 1985.

[16.8] D. Tsichritzis, "The Equivalence Problem of Simple Programs", Journal of the Association for Computing Machinery, Vol. 17, No. 4, Oct. 1970, pp. 729-738.

[17] Steven J. Zeil, Lee J. White, "Sufficient Test Sets for Path Analysis Testing Strategies", *Proceedings of 5th International Conference on Software Engineering*, (1981).

Index