A Recursion Theoretic Approach to Program Testing

JOHN C. CHERNIAVSKY AND CARL H. SMITH

***Abstract*—Inductive inference, the automatic synthesis of programs, bears certain ostensible relationships with program testing. For inductive inference, one must take a finite sample of the desired input/output behavior of some program and produce (synthesize) an equivalent program. In the testing paradigm, one seeks a finite sample for a function such that any program (in a given set) which computes something other than the object function differs from the object function on the finite sample. In both cases, the finite sample embodies sufficient knowledge to isolate the desired program from all other possibilities. These relationships are investigated and general recursion theoretic properties of testable sets of functions are exposed.**

***Index Terms*—Inductive inference, program testing, recursion theory, white box testing.**

I. Introduction

THERE is a striking symmetry between the problems of program testing and inductive inference. The testing problem consists of finding a finite set of inputs and associated outputs that characterize the most likely errors in a given program. The inference problem consists of discovering a program given a finite set of inputs and associated outputs. This symmetry has been exploited before. Weyuker [24] developed a testing methodology based upon inferring a program from given inputs and outputs. If the inferred program was equivalent to the program to be tested, then the test data were deemed to be adequate. This connection was also noted, but not exploited, in [5].

The approach we take is best characterized as specification independent testing [10]. The program's correctness is to be determined solely from its internal structure (intensional information). This theory is general enough to describe both white box and black box testing methodologies [1]. The approach is recursion theoretic in nature and the sort of questions that we ask reflect this view. Our approach does not result in any particular testing methodology nor does it give practical hints for testing. What it does provide are precisely stated limitations as to the power of any testing methodology.

Manuscript received November 30, 1984; revised January 31, 1986. C. H. Smith was supported by the National Science Foundation under Grant MCS 8301536 and by the National Security Agency OCREAE under Grant MDA904-85-H-0002.

J. C. Cherniavsky is with the Department of Computer Science, Georgetown University, Washington, DC 20057 and the Institute for Computer Science and Technology, National Bureau of Standards.

C. H. Smith is with the Department of Computer Science and the Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742 and the Institute for Computer Science and Technology, National Bureau of Standards.

IEEE Log Number 8714557.

Our main results concern the relationship between testing and inductive inference (in the context of recursion theory). We show that for recursively enumerable sets of functions, inference is more difficult than testing. On the other hand, if the recursive enumerable condition is dropped, then testing and inference are incomparable. We also develop a connection between completely recursively enumerable sets and testable sets.

II. Notation

We are concerned with the testing and synthesis of programs. Consequently, the notion of *program* must first be formalized. We restrict our attention to programs computing functions that take a single natural number as input and produce another one as output. In the context of recursion theory, this restriction entails no loss of generality since any string of symbols can be uniquely encoded as single natural number. See [13] for an example coding scheme. As the results below are independent of any particular programming language, we will start with the language of the reader's choice. The choice of language must satisfy only two constraints. Firstly, one must be able to algorithmically determine if a string of symbols is a syntactically well formed program in the language of choice. Secondly, the language must possess an unambiguous semantics. Clearly, an implemented language with a compiler for some machine satisfies these constraints.

Programs in the language of choice will be formed from strings of symbols from some alphabet. For the sake of example, consider the ascii alphabet. Finite strings of symbols over a finite alphabet form a *recursively enumerable* (r.e.) set [12]. The enumeration starts with the null string, followed by all the strings with one symbol listed alphabetically, followed by all the strings with two symbols listed alphabetically, followed by $\cdots$. The function of one argument computed by the program represented by the ith string in the above-mentioned list of strings will be denoted by φ_i. If it turns out that the ith string is not a syntactically well formed program in the language of choice, then, by convention, φ_i is the everywhere undefined function. The list of programs, $\varphi_0, \varphi_1, \cdots$, is called an *acceptable programming system* [13] (originally called an *acceptable Gödel numbering* [17]). Consequently, the natural numbers (N) will name the programs. Furthermore, several properties are known to hold for any acceptable programming system. Some of these properties are used in the proofs below.

Functions can be represented as sets of ordered pairs. For example, some function f could be represented by

0098-5589/87/0700-0777$01.00 © 1987 IEEE

$\{(x, f(x)) | x \in N\}$. Finite functions will be denoted by lower case Greek letters. We will say $\sigma \subset f$ if and only if the set $\{(x, \sigma(x)) | x \in \text{domain } \sigma\}$ is precisely the set $\{(x, f(x)) | x \in \text{domain } \sigma\}$. Furthermore, σ is an *initial segment* of f if $\sigma \subset f$ and it holds that $\forall x$ and y if $x \in$ domain σ and $y \leq x$ then $y \in$ domain σ. The functions f and g are called *finite variants* of each other if the set $\{x | f(x) \neq g(x)\}$ is finite. A set S is *closed under finite variance* if whenever $f \in S$ and g is a finite variant of f, then $g \in S$. Our concern with finite functions also requires an effective, canonical list $D_0, D_1, \cdots$ of all the finite sets. Each finite set in the list can be enumerated and the end of the enumeration can be effectively determined. One way to form such a list is to say $y \in D_x$ if and only if the yth bit of the binary representation of x is 1 [18]. From the enumeration of finite sets it is possible to form an effective canonical enumeration, $\pi_0, \pi_1, \cdots$ of all and only the finite functions [13].

III. Notions of Inference and Testability

Inductive inference has been studied for centuries with the most recent activity by computer scientists [2]. The basic idea is to synthesize a program given a finite sample of its intended input/output behavior. An *inductive inference machine* is an algorithmic device which inputs a function, represented as a set, an ordered pair at a time *and* while doing so, occasionally outputs programs intended to compute the function being input. Although the above definition does not explicitly mention the *order* in which the inference machine sees its input, we can nonetheless pretend that our inference machines see only total functions in their natural increasing domain order $(0, f(0)), (1, f(1)), \cdots$ [4], [6]. If M is an inductive inference machine and σ is an initial segment of some function then $M(\sigma)$ denotes the last output made by M after seeing all the ordered pairs in σ but before receiving any additional input.

As a mathematically rigorous view of an inductive inference machine, consider a two tape Turing Machine with one of the tapes being read only. The value on the other tape is considered the current output of the machine. If an input segment σ, of size $n + 1$ is placed on the read only tape in order $(0, \sigma(0)), (1, \sigma(1)), \cdots (n, \sigma(n))$, then $M(\sigma)$ is formally defined to be the value on the output tape at the precise moment that the read head on the read only tape moves to the right of the string "$(n, \sigma(n))$" for the first time. An IIM M that stops reading input at some point is particularly uninteresting. We may (uniformly and effectively) modify each IIM to read an additional input every few instructions. This will not guarantee that the IIM will use the data in any meaningful way, only that the machine eventually reads each data point. Consequently, we may assume without any loss of generality that the quantity "$M(\sigma)$" is effectively computable for any M and any σ. The computability of $M(\sigma)$ for finite σ's is essential for the proofs below (and in the work in inductive inference referenced below).

There are several notions of what constitutes a successful inference by an inductive inference machine. The success criterion we will use is due to Gold [9]. An inductive inference machine M *converges* to i on input from f (written: $M(f) \downarrow i$) if there is a $\sigma \subset f$ such that $M(\sigma) = i$ and for each τ if $\sigma \subset \tau \subset f$ then $M(\tau) = i$. In the event that M gives up reading input, then we say the M has converged to the *last* program that it produced as output. M infers f if and only if there is an i such that $M(f) \downarrow i$ and $\varphi_i = f$. Each inductive inference machine M will infer a set of recursive functions denoted by $EX(M)$. The inference of partial recursive functions can also be discussed by saying that M infers a partial function ψ if and only if $M(\psi) \downarrow i$ and $\phi \subset \varphi_i$.

The object of testing is to produce a *test set*, or a finite sample of the input/output behavior of some function, say f, such that any program matching the behavior of f on the test set can be safely concluded to compute f. Of course, this goal is *a priori* impossible since any finite test set has infinitely many different recursive extensions. Consequently, testing can only be done within the confines of a predetermined set of recursive functions called S in the definition below. The idea here is to distinguish functions in S from each other in those cases when both the object function and the function computed by the program to be tested are in S. T is an *adequate* test for S if and only if for each $f \in S$ if $\varphi_i \in S$ then [$\varphi_i = f$ if and only if $T(i) \subset f$]. The testing agent T maps programs to finite functions. Given a function $f \in S$ and a program i computing some function in S, T is adequate to determine if $\varphi_i = f$. If a testing agent exists for a set S, then S is called a *testable* set.

For T to be a useful testing agent it must be at worst a partial recursive function with domain including $\{i | \varphi_i \in S\}$. Ideally, T would be recursive and easy to compute. We placed no restriction on T in the above definition because we wanted to investigate testing in its full generality. For the same reason, we have not been specific about *how* T computes. If the computation of T proceeds by outputting ordered pairs then we may not be able to tell when all of $T(i)$ has been output. On the other hand, T could produce a single integer which is to be interpreted as canonical index for a finite function. If it is the case that we can enumerate the ordered pairs of $T(i)$, for any i, and know when the list is complete, we will call the finite test sets generated by T canonical. For brevity, we will say that T is canonical in such cases. We say that T is *semicanonical* if T is partial recursive and when i is in the domain of T, we know when the enumeration of $T(i)$ is complete.

Note that our definition of adequacy differs from many of the published notions of adequacy. Our notion is essentially equivalent to the definition of program adequate given in [24]. As noted in [24], this notion can be viewed as a generalization of many other notions of adequacy, including those based upon white box testing and black box testing [1]. If black box testing is viewed as generating test data that distinguish functions from one another, then our definition of testing fits the black box testing par-

adigm. But as noted in [24] the notion of white box testing can also be incorporated within the definition of program adequate. White box testing is testing where knowledge of the program is used rather than knowledge based solely on the functional behavior of the program. White box testing methodologies view test data sets as being adequate when some "coverage" criteria is satisfied. This can generally be expressed as test data selection that forces particular paths in the program to be executed. This can be further expressed as a symbolic computation in which certain predicates are required to evaluate to true. In all cases of which we are aware, these coverage criteria can be expressed as distinguishing among a finite set of programs obtained from the program under test using simple modifications. For example, Gourlay [10] gives a general construction showing how systematic path coverage can be enforced by modification of the original program (the construction is part of a proof showing that $PATH_n$ testing is strictly weaker than $PATH_{n+1}$ testing). Weyucker and Davis give constructions that ensure path coverage in their more general notion of program-based test data adequacy [22], (Weyucker, private communication). DeMillo *et al.* [8] in their mutation analysis work rely on distinguishing between programs obtained by their simple mutation operators.

Incorporation of the notion of white box testing into our theory is now straightforward. Given a program i that is to be tested by a white box testing methodology, form the programs that are to be distinguished by the test data. That collection of programs then defines the class S of functions that are to be distinguished. Test data that distinguish these functions force the paths to be executed and thus satisfy the coverage criteria.

One small point has to be addressed and that is the assurance that the programs that are to be distinguished compute recursive functions. In practice, a timer, or counter, would be included in each program defining a function in S which, after a specified period of time has elapsed without the program halting, would stop the program's execution and output some special value. The following simple theorem then gives us the desired results. Hence, white box testing is included in our theory.

Theorem 1: If S is a finite set of recursive functions, then S is testable via a semicanonical testing agent.

Proof: Since S is finite, there is some n such that all the functions in S are distinguished by their values on the first n integers. $T(i)$ is just the ith program restricted to the first n integers. If $\varphi_i \in S$ then φ_i is recursive and defined on the first n arguments. Hence, T is semicanonical. If i computes a function in S, then $T(i)$ distinguishes that function from the rest of the functions in S. □

IV. The Testing Inference Relationship

The first relationship between testing and inductive inference that we wish to note is that there are sets S which are inferable but not testable. A particularly strong formal statement is proven below. First the particular set S will be chosen. The functions of *finite support* (S_*) are all those recursive functions that evaluate to 0 on all but finitely many arguments [14].

Theorem 2: S_* is inferable.

Proof: An inductive inference machine M is defined such that $S_* \subseteq EX(M)$. M will be able to accept inputs from S_* in any order. M starts out by initializing $\sigma = 0$ and outputting a program *zero* such that φ_{zero} computes the everywhere zero function. Then, M repeats the following instructions eternally:

> Input an ordered pair (x, y). If $y = 0$ then do nothing. If $y \neq 0$ then add (x, y) to σ, e.g., $\sigma = \sigma \cup \{(x, y)\}$ and output a program for ψ where
>
> $$\psi(x) = \begin{cases} \sigma(x), & \text{if } x \in \text{domain } \sigma; \\ 0, & \text{otherwise.} \end{cases}$$

Suppose $f \in S_*$. Since M outputs a new program only when a pair (x, y) with $y \neq 0$ is read and there are only finitely many such pairs contained in f, $M(f) \downarrow i$ for some program i. At the point when M outputs i, σ contains all the pairs (x, y) from f such that $y \neq 0$, as otherwise M would eventually output some program other than i. $\varphi_i = f$, as if $x \in$ domain σ then $\varphi_i(x) = \sigma(x) = f(x)$ and if $x \in$ domain σ then $\varphi_i(x) = 0 = f(x)$. Hence, $f \in EX(M)$. □

Theorem 3: S_* is not testable.

Proof: Assume that S_* is testable and let T be the testing agent. For this proof, it does not matter whether T is canonical, or even effectively computable. Given any program i computing a function of finite support, let $T(i)$ be the finite function produced (perhaps noneffectively) by the testing agent. There are an infinite number of functions of finite support extending $T(i)$ that are not equivalent to φ_i. Thus we have a function $f \in S_*$ such that $T(i) \subset f$ and $f \neq \varphi_i$. Hence, S_* is not testable. □

Not only is S_* untestable, every class of functions that is closed under finite variance is untestable. This follows since the *only* property of S_* used in the above proof is that it is closed under finite variance. The abundance of untestable sets may seem to be a serious problem. However, in many practical applications, the set S is finite and the apparent problem disappears. For eample, by Theorem 3, the class of functions described by finite automata is not testable. However, if the class is restricted in some natural way, say to the class of all functions described by automata with fewer than n states (for some value of n) then testing can be performed adequately.

As a strengthening to Theorem 3 above notice that S_* is r.e. To see this, an argument similar to the one above is used. Construct a recursive function g such that $\{\varphi_{g(i)} \mid i \in N\} = S_*$ as follows. For $x \in N$, by the Church–Turing Thesis, let

$$\varphi_{g(x)}(y) = \begin{cases} \pi_x(y) & \text{if } y \in \text{domain } \pi_x \\ 0, & \text{otherwise.} \end{cases}$$

Clearly, $S_* = \{\varphi_{g(x)} \mid x \in N\}$.

Now suppose that T is an adequate test for S. If S is an

r.e. class of recursive functions, then S is *extrapolable* [3]. Every extrapolable function is also inferable [6]. Consequently, all the testable sets of recursive functions which are r.e. are also inferable making testing a strictly weaker notion than inference for r.e. sets of programs. In fact, a slightly stronger result holds.

Theorem 4: Any testable r.e. set of partial recursive functions is inferable.

Proof: Suppose f is the recursive function witnessing that S is r.e., e.g. $S = \{ \varphi_{f(i)} \mid i \in N \}$. Let T be an effective (e.g., computable) testing agent for S. T may not be canonical. Let $T(i)^s$ denote the finite set of ordered pairs appearing within s steps after the computation of T on input i is started. An inductive inference machine M is defined by specifying $M(\sigma)$ for all finite functions σ.

> Suppose σ is input to M. Let c denote the cardinality of σ. If there is no value $i \leq c$ such that $T(f(i))^c \subseteq \sigma$ then M outputs nothing ($M(\sigma) = 0$). Otherwise choose $i \leq c$ least such that $T(f(i))^c \subseteq \sigma$ and output $f(i)$ ($M(\sigma) = f(i)$).

Suppose $\psi \in S$. Choose j least such that $\psi = \varphi_{f(j)}$. Let $\sigma_0 \subset \sigma_1 \subset \sigma_2 \cdots$ be an enumeration of the graph of ψ. For the enumeration to correspond to a valid presentation of ψ to M it must be that $(\forall n)[\sigma_n \subset \psi]$ and $(\forall x \in \text{domain } \psi)(\exists n)[x \in \text{domain } \sigma_n]$. Choose n large enough such that (for c = the cardinality of σ_n)

1) $T(f(j))^c = T(f(j))$,
2) $T(f(j)) \subseteq \sigma_n$, and
3) $(\forall i < j)[T(f(i))^c \not\subseteq \sigma_n]$.

Clearly, an n satisfying 1) and 2) exists. For such an n suppose by way of contradiction that there is an $i < j$ such that $T(f(i)) \subset \sigma_n$. Since $\psi, \varphi_{f(i)} \in S$ and T is an adequate test for S, $\varphi_{f(i)} = \psi$, contradicting the choice of j. □

However, if the constraint that S is r.e. is dropped, then there are testable sets of recursive functions which are not inferable, making the two notions incomparable.

Theorem 5: There is a non-r.e. set of recursive functions which is testable but not inferable. Furthermore, the testing agent is semicanonical.

Proof: The proof proceeds in two parts. A set of nonidentifiable functions is constructed. Then a non-r.e. set S is formed along with its adequate test to complete the proof. Since inductive inference machines are algorithmic devices, we can form an enumeration, $M_0, M_1 \cdots$ of all of them in a manner similar to the enumeration of all the Turing Machines. That is, start with an enumeration of all strings over some fixed but arbitrary alphabet with enough symbols to express the algorithm used by any inference procedure. If some string j does not correspond to a syntactically valid inductive inference machine, then we say that M_j identifies nothing. Program $p(i)$ below is designed to fool M_i. Unfortunately, for some i's, program $p(i)$ may compute a finite, and hence not a total, recursive function. The construction below will guarantee that if $\varphi_{p(i)}$ is recursive then it cannot be inferred by M_i. On the other hand, if $\varphi_{p(i)}$ is finite then M_i will fail to infer any recursive extension of $\varphi_{p(i)}$. The twisted argument is necessary since $\{ \varphi_{p(i)} \mid i \in N \}$ is r.e. and hence, inferable. The basic idea of the proof is to, for each inference machine, construct a function uniquely identified by its value at argument 0 which baffles the given inference machine.

Begin program $p(i)$. On input x, successively execute the stages $s \geq 0$ described below until (if ever) $\varphi_{p(i)}(x)$ is defined. $\varphi^s_{p(i)}$ denotes the finite amount of $\varphi_{p(i)}$ defined before stage s. At each stage we will look for a proper extension of $\varphi^s_{p(i)}$. By way of initialization, set $\varphi^0_{p(i)} = \{(0, i)\}$, thereby guaranteeing testability.

> *Begin stage s.* Simultaneously execute the two subprocedures below until one of them succeeds in defining $\varphi^{s+1}_{p(i)}$. Alternatively, dovetail the computations described below.
>
> 1) Look for an initial segment σ such that $\varphi^s_{p(i)} \subset \sigma$ and $M(\varphi_{p(i)}) \neq M(\sigma)$. When, and if, such a σ is found, set $\varphi^{s+1}_{p(i)} = \sigma$. (Then M_i is forced to change its mind.)
> 2) Look for an $x \notin$ domain $\varphi_{p(i)}$ such that $\varphi_{M_i(\sigma^s)}(x)\downarrow$. When, and if, such an x is found set $\varphi^{s+1}_{p(i)} = (\varphi^s_{p(i)} \cup \{(y, 1 \dot{-} \varphi_{M_i(\varphi^s_{p(i)})}(x)) \mid y \leq x, y \notin \text{domain } \varphi^s_{p(i)}\})$. (Then M_i's last conjecture is not a program for any extension of $\varphi^{s+1}_{p(i)}$.)
>
> End stage s.

End program $p(i)$.

The next step of the proof is to pick, for each i, a recursive function, $f_i \notin EX(M_i)$. The testable set S will be the union of the f_i's. For each i, f_i will either be $\varphi_{p(i)}$ or an extension thereof, depending on which of the two case below holds. Since we cannot determine which of the cases holds, S will *not* be r.e. [6].

Case 1: Each stage terminates. Hence, the search for $\varphi^{s+1}_{p(i)}$ always succeeds. Set $f_i = \varphi_{p(i)}$, a recursive function. Suppose that $M_i(f)\downarrow q$. Then, since M_i on f at some point stops changing its mind, $(\exists s)(\forall \sigma)[\varphi^s_{p(i)} \subset \sigma \subset f \Rightarrow M_{p(i)}(\varphi^s_{p(i)}) = M_{p(i)}(\sigma) = q]$. Hence, past stage s, all extensions to $\varphi_{p(i)}$ are made by clause 2). Therefore, $\varphi_q(x)$ converges $\neq f_i(x)$ for infinitely many numbers x found at stages $s' > s$. Hence $f_i \notin EX(M_i)$.

Case 2: Some stage s never terminates. Then $\varphi_{p(i)} = \varphi^s_{p(i)}$ a finite initial segment. Set $f_i = (\varphi_{p(i)} \cup \{(x, 0) \mid x \notin \text{domain } \varphi^s_{p(i)}\})$, a recursive function. Choose $\sigma \subset f_i$ such that $\varphi^s_{p(i)} \subseteq \sigma$. Then $M_i(\sigma) = M_i(\varphi^s_{p(i)})$ as otherwise clause 1) would have terminated stage s. Hence, $M_i(f_i) \downarrow = M(\varphi^s_{p(i)})$. Furthermore, $\varphi_{M_i(f_i)}(x)$ is undefined for any for any $x \notin$ domain $\varphi^s_{p(i)}$, as otherwise clause 2) would have terminated stage s. Hence $M_i(\varphi^s_{p(i)})$, M_i's final output on f_i, does not compute f_i. Therefore, $f_i \notin EX(M_i)$.

Continuing with the main proof, let $S = \{ f_i \mid i \in N \}$. S is not inferable as each inference procedure is baffled by at least one member of S. Define a partial recursive function T such that $T(i) = (0, \varphi_i(0))$. T is computable, but not recursive. Furthermore, T is semicanonical in the sense that if any ordered pair is produced by $T(i)$ then *all* of $T(i)$ has been produced. The proof is completed by

showing that T is an adequate test for S. Choose a member of S, say f_i, and a program to test, say j. If $\varphi_j \in S$, then $\varphi_j(0)$ is defined to a value which is unique among the members of S. Consequently, $[\varphi_j = f_i \text{ iff } T(j) \subset f_i]$, and T is an adequate test for S. □

With the r.e. constraint on S dropped we can even find a well known subclass of the inferable sets, all of which are testable. Following [6] we say that an inductive inference machine M EX_i infers a recursive function f if and only if M EX infers f after changing its output conjecture at most i times. $EX_i(M)$ denotes the set of recursive functions EX_i inferred by M.

Theorem 6: Every EX_0 inferable set of recursive functions is testable.

Proof: Suppose that M is an inductive inference machine. We may suppose without loss of generality that M makes at most one conjecture on any input. Let T be the test procedure defined as follows:

> On input i start up a subprocess to enumerate the set $\{(x, \varphi_i(x)) \mid x \in N\}$. The subprocess is terminated if and when $T(i)$ is defined. Let σ^s denote the finite amount of output generated by the subprocess after s computation steps. Set τ^s be the largest initial segment of σ^s. Then $T(i) = \tau^s$ where s is the least number such that $M(\tau^s)$ is defined.

Suppose now that f and φ_i are members of $EX_0(M)$. Then $T(i)$ is defined since M can EX_0 infer φ_i. Furthermore, $T(i)$ is an initial segment of the graph of φ_i. If $\varphi_i = f$ then $T(i) \subset \varphi_i = f$. If $T(i) \subset f$ then $M(f)\downarrow = M(\varphi_i) = M(T(i))$. Hence, $f = \varphi_{M(f)} = \varphi_{M(T(i))} = \varphi_i$. Therefore, T is an adequate test for $EX_0(M)$. □

Although the definition of EX_0 inference seems to be very restrictive, there are some extremely rich EX_0 inferable (and hence, testable) sets. The proof of this uses the notion of singleton variance. A function f is a *singleton variant* of a function g iff $f(x) = g(x)$ except for *one* value of $x \in N$. The proof also uses the *parametric* form of Kleene's original recursion theorem [11] (see [18]). Intuitively, the parametric recursion theorem allows one to construct sequences of programs $p_0, p_1, \cdots$ such that each program p_i knows its own description (p_i) and its position in the sequence (i). The theorem is stated below. See [20] for an intuitive discussion of this and other recursion theorems.

Lemma 7: Parametric Recursion Theorem—For any partial recursive function ψ there is a total recursive function p such that for all i and x, $\varphi_{p(i)}(x) = \psi(x, p(i), i)$.

Theorem 8: There is an r.e. set of functions containing a singleton variant of every partial recursive function. Furthermore, the same r.e. set is in EX_0.

Proof: By the parametric recursion theorem there is a recursive function p such that $\forall i, x$

$$\varphi_{p(i)}(x) = \begin{cases} p(i), & \text{if } x = 0; \\ \varphi_i(x); & \text{otherwise.} \end{cases}$$

Clearly, $\{\varphi_{p(i)} \mid i \in N\}$ is r.e. Let M_0 be an inductive inference machine that ignores all its input until it sees an ordered pair $(0, y)$ for some y. M_0 then outputs the value y and stops. Clearly, $\{\varphi_{p(i)} \mid i \in N\} \subset EX_0(M_0)$. □

The next larger class of inferable sets, EX_1, is not testable.

Theorem 9: There is a set in EX_1 which is not testable.

Proof: First, a set $\mathcal{S}$ which is EX_1 inferable is defined. Let

$$f_i(x) = \begin{cases} 0, & \text{if } x \le i; \\ 1, & \text{otherwise.} \end{cases}$$

Let $\mathcal{S} = \{f_i \mid i \in N\} \cup \lambda x[0]$. Clearly, $\mathcal{S}$ is EX_1 inferable by an inference machine that initially guesses a program for $\lambda x[0]$ and then outputs another conjecture only when it sees input $(x, 0)$, $(x + 1, 1)$. If such a pair of inputs is received, then the machine outputs a program for f_x. Suppose by way of contradiction that T is an adequate test for $\mathcal{S}$ and $\varphi_z = \lambda x[0]$, a member of $\mathcal{S}$. Then $T(z)$ is defined. Let x be the least number not in the domain of $T(z)$. Then $f_x \in \mathcal{S}$ and $T(z) \subset f_x$ but $f_x \ne \varphi_x \ne \varphi_z$, a contradiction. □

V. Relativized Testing and Inference

The above results demonstrate that testing and inference are incomparable. In this section we consider relativized testing and inference, that is testing and inference procedures that make use of an oracle (extra information). Following Rogers [18] let $K_0 = \{(x, y) \mid \varphi_x(y)\downarrow\}$. K_0 is Turing equivalent to the well known halting problem set $K = \{x \mid \varphi_x(x)\downarrow\}$. A general *oracular* procedure is an effective procedure except for calls to a subroutine computing the characteristic function of some nonrecursive set (the oracle). Another definition of oracle has been used in discussion testing issues [23]. This other definition defines the oracle to be the function putatively computed by the program being tested. We use the recursion theoretic notion of oraclar procedures [18]. Of course an oracle for the halting problem contains enough information to construct an inductive inference procedure identifying all the partial recursive functions.

Theorem 10: There is an inductive inference procedure, using an oracle for K_0, inferring all the partial recursive functions.

Proof: Let M be an inductive inference procedure that initially outputs 0 as its first conjecture and then executes the following steps eternally.

> Let *conj* be the most recent conjecture output by M. Request input. Let σ be the finite amount of input received so far. Using an oracle for K_0 determine if domain $\sigma \subset$ domain φ_{conj}. If so, then, by simulation, determine if $\varphi_{conj}(x) = \sigma(x)$ for all $x \in$ domain σ. If program *conj* fails any of these tests then output i where i is the least number $>$ *conj* passing both the above tests and set *conj* $= i$. If the input function is partial recursive then i is guaranteed to exist.

Now suppose that f is a partial recursive function fed to

M as input. Let i be the least number such that $f \subseteq \varphi_i$. If M outputs *conj* on input from f for *conj* $< i$ then it must output a new conjecture since, by the choice of i, either φ_{conj} is not defined on some point, x, in domain f or there is an x such that $\varphi_{conj}(x) \downarrow \neq f(x)$. In either case, M will reject *conj* as soon as $x \in$ domain σ. On the other hand, M will never reject i, since $f \subseteq \varphi_i$. □

Theorem 11: There is no oracular testing procedure adequate for S_*.

Proof: The proof is identical to the proof of Theorem 3, except that T is allowed the use of an oracle. □

Consequently, not even an oracle for program equivalence is powerful enough to yield adequate testing procedures for some sets. This indicates that testing is more difficult than inference. A preliminary speculation as to why testing is so different from inference is that inference procedures are limiting in nature, they can change their minds finitely often as to their answer while testing procedures cannot. We saw in Theorem 9 that inference procedures that changed their hypothesis once were more powerful than testing procedures. What if we allow testing procedures to converge in the limit to a finite test set? Even then, S_* is not testable. To see this, we appeal to the *limit lemma*, a folk theorem appearing in [19].

Lemma 12: A total function f is computable with an oracle for the halting problem if and only if there is a recursive function h such that for all x, $\lim_{s \to \infty} h(x, s) = f(x)$.

If there were a limiting test procedure T such that given $F \in S_*$ and $\varphi_i \in S_*$, $\varphi_i = f \Leftrightarrow \lim_{s \to \infty} T(i, s) \subset f$, then by the limit lemma, a function equivalent to the limit of T would be computable with an oracle for the halting problem, contradicting Theorem 11. Consequently, the difficulty of testing does *not* lie in the constraint that the testing procedure generate a test set uniformly and effectively from the program it is to test. The difficulty of testing then must be because it is impossible to distinguish infinitely many functions, all of which are finite variants of each other, based on a finite sample. Furthermore, testing is difficult (impossible) even when compared only with other unsolvable problems.

It may seem counterintuitive at first that an oracle for the program equivalence problem does not always help in the testing of programs. This is due to the functional nature of our notion of program testing. The program equivalence oracle tells when two *programs* compute the same *function*. It does not say if a given *program* computes a given *function*. The function to be implemented as a program may be only vaguely specified, or specified in a nonalgorithmic fashion.

VI. Testing and Completely R.E. Sets

The relationship between testing and inference depends on whether the set to be tested or inferred is r.e. In this section the relationships between testable sets and r.e. sets is examined further. We proceed by introducing the notion of a recursively enumerable index set. Suppose that P is a property of functions and C is the set of all functions possessing property P. Then Pr_C is the *index set* consisting of all the programs computing functions in C. In symbols, $Pr_C = \{i \mid \varphi_i \in C\}$. The term "index set" stems from the historical perspective that the natural numbers serve as indexes of functions in an acceptable programming system. The following result, from [16], characterizes precisely when an index set is r.e. As such, it is an extension of the famous "Rice's" theorem characterizing when index sets are recursive. The version stated below is from [13]. The set of functions computed by the programs in an r.e. index set is sometimes called a *completely* r.e. set of functions.

Lemma 13: Suppose C is a collection of functions. Then Pr_C is r.e. if and only if $Pr_C = 0$ or there is an r.e. set A such that for all $x\,[\varphi_z \in C$ if and only if $\pi_y \subset \varphi_x$ for some $y \in A]$.

The finite functions in A are collectively called a key array. Our next result says that any testable set is contained in some completely r.e. set.

Theorem 14: Suppose T is a semicanonical adequate test for S. Then there is a completely r.e. superset of S containing only functions extending the test sets generated by T.

Proof: By the canonical and effective way in which the list of finite functions was formed, and by the Church–Turing thesis, there is a partial recursive function k such that, for all i, $\pi_{k(i)} = T(i)$ is undefined, then $k(i)$ is undefined. Let A equal the range of k, an r.e. set. Set $\overline{S} = \{\psi \mid$ there is an $i \in A$ such that $\pi_{k(i)} \subseteq \psi\}$. By Lemma 13, $\overline{S}$ is completely r.e. Suppose $f \in S$ and $\varphi_i = f$. Then $T(i) \subseteq \varphi_i$. Therefore, $\pi_{k(i)} \subseteq \varphi_i$. Hence, $\varphi_i \in S$ implies $f \in \overline{S}$. □

As a companion result to the above Theorem we also have the following.

Theorem 15: Suppose S is a completely r.e. set. Then there is an r.e. semicanonical testable subset of S containing only recursive functions.

Before presenting the proof we pause for a technical Lemma.

Lemma 16: If S is a completely r.e. set with key array A generated by f, then there is a recursive function g generating another key array for S such that if $i < j$ then $\pi_{g(i)} \not\subseteq \pi_{g(j)}$.

Proof: Suppose A generated by f is the key array for S, a completely r.e. set. Define g as follows:

$$g(x) = \begin{cases} f(i), & \text{where } i \text{ is the least number} < x \text{ such that } \pi_{f(i)} \subseteq \pi_{f(i)}, \text{ if such an } i \text{ exists;} \\ f(x), & \text{otherwise.} \end{cases}$$

If $\varphi_i \in S$ then $\pi_{f(x)} \subset \varphi_i$ for some x. If $g(x) = f(x)$ then φ_i is also in the completely r.e. set with key array generated by g. If $g(x) \neq f(x)$ then, by the construction, $g(x) = f(j)$ for some $j < x$. In which case, $\pi_{f(j)} \subset \pi_{f(x)} \subset \varphi_i$. Since $g(x) = f(j)$ for some $j < x$ and, consequently, $\varphi_i \in S$. □

Proof (of Theorem 15): Let A be the r.e. set witnessing that S is completely r.e., let f be a recursive function

with range A. We may suppose without loss of generality, by Lemma 16, that if $i < j$ then $\pi_{f(i)} \not\subseteq \pi_{f(j)}$. The first step of the proof is to construct another key array generating a completely r.e. set $\bar{S} \subset S$. The key array for $\bar{S}$ will be enumerated by a recursive function g such that for all i and j either $\pi_{g(i)} = \pi_{g(j)}$ or there is an $x \in$ (domain $\pi_{g(i)} \cup$ domain $\pi_{g(j)}$) such that $\pi_{g(i)}(x) \neq \pi_{g(j)}(x)$. The testable set and the testing function will be built from g. g will be defined in effective stages of finite extension, with $g(s)$ being defined in stage s. The construction will employ counters, one for each number. Initially zero, counter c_i will be incremented so that its value is at least as big as the value of $\pi_{g(j)}(i)$ for any j such that $g(j)$ has already been determined and $i \in$ domain $\pi_{g(j)}$. To emphasize the effectiveness of the procedure, a set variable C is used to keep track of the active counters. At any point, only finitely many counters will be active. Initially, $C = 0$. Execute the following stages in order.

Begin stage s. If there is an $\bar{s} < s$ such that $f(\bar{s}) = f(s)$ then set $g(s) = g(\bar{s})$ and go to stage $s + 1$. Otherwise, set $ADD = \{i \in C \mid i \notin \text{domain } \pi_{f(s)}\}$. Choose k such that $\pi_k = \pi_{f(s)} \cup \{(i, c_i + 1) \mid i \in ADD\}$. Since C and ADD are finite and $\pi_{f(s)}$ is a finite function, the choice of k is effective. Set $g(s) = k$ and increment all counters c_i such that $i \in ADD$. For each $j \in$ domain $\pi_{f(s)}$ do the following:
If $j \in C$ then set c_j to the max of its current value and $\pi_{f(s)}(j)$ else put j in C and initialize $c_j = \pi_{f(s)}(j)$.
End stage s.

We claim that for each i and j either $\pi_{g(i)} = \pi_{g(j)}$ or $\pi_{g(i)}$ and $\pi_{g(j)}$ are inconsistent. Suppose by way of contradiction that $i < j$, $g(i) \neq g(j)$ and $(\forall x \in (\text{domain } \pi_{g(i)} \cap \text{domain } \pi_{g(j)}))[\pi_{g(i)}(x) = \pi_{g(i)}(x)]$. By Lemma 16, $\pi_{f(i)} \not\subseteq \pi_{f(j)}$. So there must be an $x \in$ (domain $\pi_{f(i)}$ − domain $\pi_{f(j)}$). Then counter c_x will have value at least $\pi_{g(i)}(x)$ at the beginning of stage $i + 1$. At stage j then, $g(j)$ is chosen such that $\pi_{g(j)}(x) > c_x \geq \pi_{g(i)}(x)$, which contradicts the choice of i and j.

Let $\bar{S}$ be the completely r.e. set generated using g as key array. If $\varphi_i \in \bar{S}$ then there is an x such that $\pi_{g(x)} \subset \varphi_i$. By the construction, $\pi_{f(x)} \subseteq \pi_{g(x)}$ so $\varphi_i \in S$.

The next step is to build a testable subset of $\bar{S}$. Let $\hat{S} = \{\psi_i \mid i \in N\}$ where ψ_i is defined as:

$$\psi_i(x) = \begin{cases} \pi_{g(i)}(x), & \text{if } x \in \text{domain } \pi_{g(i)}; \\ 0, & \text{otherwise.} \end{cases}$$

Clearly, $\hat{S}$ is an r.e. subset of the functions of finite support. Notice that there is only one member of $\hat{S}$ extending any of the $\pi_{g(i)}$'s. The proof is completed by finding an adequate test for $\hat{S}$. T is computed by the following informal algorithm:

On input i, use the key array g to enumerate $\bar{S}$ until i is produced as output. Then it must be that $\pi_{g(x)} \subset \varphi_i$ for some x. Set $T(i) = \pi_{g(x)}$. With a little more care and a lot more notation, the x can be effectively found. If i never appears then this procedure does not terminate and $T(i)$ is undefined. The semicanonicalness of T follows from the canonical nature of the π_j's.

Suppose f and φ_i are members of $\hat{S}$ and we wish to test if program i computes f. Since $\hat{S} \subset \bar{S}$, $T(i)$ is defined, say equal to $\pi_{g(x)}$. In which case, $\pi_{g(x)} \subset \varphi_i$. If $\pi_{g(x)} \subset f$ then both f and φ_i are in $\hat{S}$, both extend $\pi_{g(x)}$ and there is only one function in $\hat{S}$ extending $\pi_{g(x)}$. Hence, $f = \varphi_i$. If $\pi_{g(x)} \not\subset f$ then there is a y such that $f(y) \neq \pi_{g(x)}(y)$. By the choice of x, $\varphi_i(y) = \pi_{g(x)}(y)$, so $f \neq \varphi_i$. Therefore, T is an adequate test of $\hat{S}$. □

One of the first observations made in Section III was that certain restrictions of impossible testing problems are in fact testable. Problem pruning seems to be fundamental to contemporary computing. In some cases, the difference may not be between the solvable and unsolvable, but rather the restricted problem may be tractable whereas the original was not. By the previous theorem we know that any completely r.e. set has a testable subset. Our final result indicates that any completely r.e. set can be made infinitely smaller and that this process can be iterated arbitrarily often. The results of this section exhibit the existence of effective, not necessarily efficient, techniques to prune testable sets. Starting with a testable set, apply Theorem 14 to get a completely r.e. superset. Then apply the following Theorem arbitrarily often, each application making the completely r.e. set "smaller." Finally, apply Theorem 15 to obtain another, smaller, testable set.

Theorem 17: For every nonempty completely r.e. set $\mathcal{A}$ there is a nonempty completely r.e. set $\mathcal{B}$ such that $\mathcal{B} \subset \mathcal{A}$ and $\mathcal{A} - \mathcal{B}$ is infinite.

Proof: Let f generate $\mathcal{A}$, the key array for $\mathcal{A}$. We may suppose by Lemma 16 that if $i < j$ then $\pi_{f(i)} \not\subseteq \pi_{f(j)}$. Let y be the least number not in the domain of $\pi_{f(0)}$. Choose k such that $\pi_k = (\pi_{f(0)} \cup \{(y, 0)\})$. Define a recursive function g as follows:

$$g(x) = \begin{cases} k, & \text{if } \pi_{f(x)} = 0 \text{ or } \pi_{f(x)} \subset \pi_k; \\ f(x), & \text{otherwise.} \end{cases}$$

Let $\mathcal{B}$ be completely r.e. set with key array B generated by g. Suppose $h \in \mathcal{B}$. Then $(\exists i)[\pi_{g(i)} \subset h]$. Either $g(i) = f(i)$ placing $h \in \mathcal{A}$ or $g(i) = k$ and $\pi_{f(0)} \subset \pi_k = \pi_{g(i)} \subset h$, again placing $h \in \mathcal{A}$. Therefore, $\mathcal{B} \subset \mathcal{A}$. The proof is completed by exhibiting infinitely many functions in $\mathcal{A} - \mathcal{B}$. Define recursive functions $h_1, h_2 \cdots$ as follows:

$$h_i(x) = \begin{cases} \pi_{f(0)}(x), & \text{if } x \in \text{domain } \pi_f(0); \\ i, & \text{if } x = y; \\ 0, & \text{otherwise.} \end{cases}$$

The first clause ensures that $\forall i$, $h_i \in \mathcal{A}$. If $h_i = h_j$ then $i = j$ by the second clause. Hence $\{h_i \mid i > 0\}$ is infinite. The choice of y and the first clause of the definition of g guarantee that $\forall i \nexists z[\pi_{g(z)} \subset h_i]$. Consequently, $\{h_i \mid i > 0\} \cap \mathcal{B} = 0$. □

VII. Conclusions

We have outlined the beginnings of a recursion theoretic approach to testing using insights from inductive inference. The notions of inductive inference and program testing were shown to be incomparable. The results involving oracles indicate that program testing is *much* harder than inductive inference. The results relating completely r.e. sets to testable sets suggest that there are *effective* (but not necessarily practical) methods to trim a testable set down to a (perhaps) efficiently testable set. We believe that the hierarchy of testable subsets of completely r.e. set represent a tradeoff between the complexity of the testing agent and the size of the set of distinguishable functions. We believe that additional hierarchical results can be obtained.

There is much work to be done in a number of areas. Along more practical lines, we have preliminary results that lead to practical (i.e., both effective and time efficient) testing methods for simple classes of functions. These results are obtained by restricting the complexity of the testing agent and by weakening the notation for expressing the set of testable functions [7].

We have not discussed the intriguing idea of testing to within a confidence level. For this investigation we would use a notion of probabilistic inference in obtaining notions of probabilistic testing akin to those developed for inference by Pitt [15] and Valiant [21].

We emphasize again that the results in this paper are intended to convey the limitations inherent in testing. However the connections between testing and inductive inference can also be exploited to develop practical test methods for certain restricted classes of functions [7].

Acknowledgment

R. Hamlet made some valuable comments on our definition of testable and interpretations of our results. Unknown referees suggested many improvements in the presentation and the results. Computer time was provided by the Departments of Computer Science at the University of Maryland and Georgetown University.

References

[1] W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky, "Validation, verification, and testing of computer software," *Comput. Surveys*, vol. 14, pp. 159–192, 1982.

[2] D. Angluin and C. H. Smith, "Inductive inference: Theory and methods," *Comput. Surveys*, vol. 15, pp. 237–269, 1983.

[3] J. A. Barzdin and R. V. Freivalds, "On the prediction of general recursive functions," *Soviet Math Dokl.*, vol. 13, pp. 1224–1228, 1972.

[4] L. Blum and M. Blum, "Toward a mathematical theory of inductive inference," *Inform. Contr.*, vol. 28, pp. 125–155, 1975.

[5] T. A. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta Inform.*, vol. 18, pp. 31–45, 1982.

[6] J. Case and C. Smith, "Comparison of identification criteria for machine inductive inference," *Theoret. Comput. Sci.*, vol. 25, no. 2, pp. 193–220, 1983.

[7] J. C. Cherniavsky and C. H. Smith, "Using telltales in developing program test sets," Dep. Comput. Sci., Georgetown Univ., Washington DC, Rep. TR 4, 1986.

[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, pp. 34–41, 1978.

[9] E. M. Gold, "Language identification in the limit," *Inform. Contr.*, vol. 10, pp. 447–474, 1967.

[10] J. Gourlay, "A mathematical framework for the investigation of testing," *IEEE Trans. Software Eng.*, vol. SE-9, no. 6, pp. 686–709, 1983.

[11] S. Kleene, "On notation for ordinal numbers," *J. Symbolic Logic*, vol. 3, pp. 150–155, 1938.

[12] H. Lewis and C. Papadimitriou, *Elements of the Theory of Computation*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[13] M. Machtey and P. Young, *An Introduction to the General Theory of Algorithms*. New York: North-Holland, 1978.

[14] A. Meyer, "Program size in restricted programming languages," *Inform. Contr.*, vol. 21, pp. 382–394, 1972.

[15] L. Pitt, "A characterization of probabilistic inference," in *Proc. 25th Annu. Symp. Foundations of Computer Science*, Palm Beach, FL, 1984.

[16] H. Rice, "On completely recursively enumerable classes and their key arrays," *J. Symbolic Logic*, vol. 21, pp. 304–308, 1956.

[17] H. Rogers, Jr., "Gödel numberings of partial recursive functions," *J. Symbolic Logic*, vol. 23, pp. 331–341, 1958.

[18] —, *Theory of Recursive Functions and Effective Computability*. New York: McGraw-Hill, 1967.

[19] J. Shoenfield, *Degrees of Unsolvability*. Amsterdam, The Netherlands: North Holland, 1971.

[20] C. Smith, "Applications of classical recursion theory to computer science," in *Recursion Theory: Its Generalisations and Applications*, S. Wainer and E. F. Drake, Eds. London: Cambridge University Press, 1980.

[21] L. G. Valiant, "A theory of learnable," *Commun. ACM*, vol. 27, no. 11, pp. 1134–1142, 1984.

[22] E. Weyuker and M. Davis, "A formal notion of program-based test data adequacy," *Inform. Contr.*, vol. 56, pp. 52–71, 1983.

[23] E. J. Weyuker, "On testing non-testable programs," *Comput. J.*, vol. 25, no. 4, pp. 465–470, 1982.

[24] —, "Assessing test data adequacy through program inference," *ACM Trans. Program., Lang., Syst.*, vol. 5, no. 4, pp. 641–655, 1983.

John C. Cherniavsky received the B.S. degree in mathematics from Stanford University, Stanford, CA, in 1969 and the M.S. and Ph.D. degrees in computer science from Cornell University, Ithaca, NY, in 1971 and 1972, respectively.

He is Professor and Chairman of Computer Science at Georgetown University, Washington, DC. He was on the faculty of the Department of Computer Science at the State University of New York at Stony Brook from 1972 and 1980. From 1980 to 1984 he was Program Director for Theoretical Computer Science at the National Science Foundation. He has also been associated with the Institute for Computer Science and Technology of the National Bureau of Standards since 1978 as a consultant.

Carl H. Smith received the B.S. degree in mathematics from the University of Vermont, Burlington, in 1972 and the M.S. and Ph.D. degrees in computer science from the State University of New York at Buffalo in 1975 and 1979, respectively.

He was on the faculty of the Department of Computer Sciences at Purdue University from 1978 until moving to the University of Maryland, College Park, in 1982. In 1985 he became a member of the University of Maryland Institute for Advanced Computer Studies as well as joining the research staff of the Institute for Computer Science and Technology of the National Bureau of Standards.