Testability of Software Components

Roy S. Freedman

Abstract—**The knowledge as to whether a software component is testable or not is important to the software engineering process: a program that is not easily testable may have to go through several iterations of program and test redesign. In this paper we formally investigate the meaning of software testability. We define a new concept, domain testability, by applying the concepts of observability and controllability to software. Observability refers to the ease of determining if specified inputs affect the outputs; controllability refers to the ease of producing a specified output from a specified input. Observability and controllability properties are already used for assessing the testability of hardware components. A domain testable program is observable and controllable: it does not exhibit any test input–output inconsistencies. We discuss the domain testability properties of several programs which have been presented in the literature and investigate the observability and controllability properties of several programming structures. We also define new testing metrics (that can be applied to programs or to functional specifications) which can be used to easily assess the level of effort required to modify a program so that it becomes domain testable. We also show how testability can be assessed from program specifications and discuss an experiment which shows that it takes less time to build and test a program developed from a domain-testable specification than a similar program developed from a nondomain-testable specification. In the Appendix, domain testability is defined in terms of denotational semantics.**

Index Terms—**Software testing, functional testing, testability, observability, controllability.**

I. Introduction

Testing is a crucial software development activity that is used for determining whether a program has errors. Testing is used to assess the compliance of a program to its intended specifications and to assess the reliability of the program to inputs which were not intended to be part of the specifications. Good test inputs are those that result in a high probability of discovering errors upon program execution.

Before testing begins, it is necessary to identify the input and output data characteristics (i.e., type, format, range) of the component and the "states" of the component. During testing, the input–output data characteristics may be inferred from specific identifiers in the program; states and state transitions usually are not associated with specific identifiers.

Test plans are documented in a Test Design Specification [32]. This document specifies the test inputs, the expected outputs, and information related to timing, display, or the test execution environment. The actual outputs and information such as timing or display anomalies (and descriptions of attempts to repeat the test) are documented in the Test Incident Report [32].

Research in testing has focused on the following two problems:

- Test Effectiveness: What is the best selection of test data?
- Test Adequacy: How do we know that sufficient testing was performed?

Test effectiveness is concerned with generating the smallest set of input test data whose output would result in the discovery of the largest set of errors. The definition of an "effective" test was addressed by [11], [12]: if we can partition the input space into a finite set of input equivalence classes, then an effective test is a test which executes one input from each class. The problem is in finding the partition. In the extreme case where the partition is discrete and the equivalence class consists of singletons, the effective test becomes an exhaustive test: all inputs must be executed. Exhaustive tests should be avoided on pragmatic grounds: they are combinatorially explosive, and, for programs with infinite input domains, can never be completed.

A testing strategy can be considered to be a method of generating a finite set of input equivalence classes. These methods range from functional (specification or "black-box") methods which select test inputs based on specification-derived values [20], to structural ("white-box") methods which select test inputs based on a percentage of the statements (or other structures) which can be executed [16], [29]. Test adequacy is concerned with determining the effectiveness of test selection strategies. Several strategies were experimentally compared by Basili [2]. And given a test selection method, Weyuker [30] and Zweben [31] discuss sets of rules to determine whether or not sufficient testing has been performed.

One assumption in determining test effectiveness or test adequacy is that testing can be improved by improving the test-selection strategy or the test-adequacy criteria. It is assumed that all programs are "easily testable," or at least that the amount of work needed to identify the input and output data characteristics and states of the component is small. The problem with this assumption is that the emphasis is on selecting input data and not on determining whether the given program is easily testable or not. In this paper we address the problem of testability: what are the properties of "easily testable" programs?

The knowledge as to whether a component is easily testable is important to the test planning and budgeting process: a program that is not easily testable may have to go through several iterations of test redesign.

Software engineering design methodologies have informally stated some properties of "easily testable" software compo-

Manuscript received January 4, 1990; revised January 15, 1991. Recommended by M.S. Deutsch.

The author was with the Department of Computer Science, Polytechnic University, 333 Jay Street, Brooklyn, NY 11201. He is now with Inductive Solutions, Inc., 380 Rector Place, New York, NY 10280.

IEEE Log Number 9144256.

	Test 1	Test 2	Test 3	Test 4	Test 5
Input X:	0	1	2	3	0
Output A(X):	RED	GREEN	RED	GREEN	GREEN

Fig. 1. Input–output table showing test anomalies.

nents [3], [5]–[7], [14], [25], [27]. Some properties of "easily testable" hardware components have been also identified [8], [23]. Intuitively, a software component that is "testable" has the following desirable properties:

Test Sets are Small and Easily Generated: Exhaustive tests are infeasible. Testing pragmatics dictate finite-test budgets.

Test Sets are Nonredundant: Repeated values for test inputs are inefficient. On the other hand, something may be seriously wrong if the same test value yields different results.

Test Outputs are Easily Interpreted: Test cases are represented in a Test Specification [32], where inputs and the anticipated outputs are tabulated. The precise identification of these inputs and outputs is necessary for an effective and adequate test.

Software Faults are Easily Locatable: Software faults are easily traced to specific components and inputs.

The knowledge as to whether a program is easily testable software or not is not explicitly provided to testers *a priori*. During testing, testers may determine that certain software components are not "easily testable," because they may exhibit one of the following input–output inconsistencies:

Input Inconsistencies: If a test input value is repeated, its output result should be the same. If it is not, then the test inputs are incomplete: the outputs are not functionally related to the inputs alone, but on some other states which have not been identified by the testers. For example, given software component A with the following specification-based (black-box) context:

```
type CODE is (0,1,2):
type COLOR is (RED, YELLOW, GREEN):
function A (X:in CODE) return COLOR;
```

An excerpt from a Test Incident Report in Fig. 1 shows that function A returns different values for some identical inputs. It may be that function A returns these values due to certain side-effects in the execution environment. There may be undiscovered state and state transitions which are not associated by the tester with specific inputs. Input inconsistency illustrates the fact that just because easily testable programs should have nonredundant test sets, it does not imply that any program has a nonredundant test set. This type of inconsistency occurs in applications involving database management systems and distributed systems.

Output Inconsistencies: If an output identifier is specified to range over a domain of values, we should be able to construct a test input whose execution can "cover" any one of the specified values for the output identifier. If the component is a predicate in a conditional statement or a loop, an output inconsistency can imply the existence of unreachable statements or unreachable control paths.

Domain specifications are stated in identifier declarations, type definitions, comments, and in functional specifications. For example, the excerpt from the Test Incident Report in Fig. 1 shows that A(X) never returns the value YELLOW for all test inputs specified. It may be that either:

- A was never specified in the functional requirements to assume YELLOW
- A was incorrectly or incompletely specified
- A returns the value YELLOW due to unidentified states in the execution
- A contains an infinite loop, and the tester (via the run-time environment) interrupts computation so that an interrupting value is returned.

An example of this last possibility seen in the call to A as in the following testing environment:

```
procedure MAIN is
  type CODE is (0,1,2):
  type COLOR is (RED, YELLOW, GREEN):
  function A (X:in CODE) return COLOR is ...
  end A;
  OUTPUT : CODE := GREEN; -- initial_value;
  ...
begin
  loop
    GET(INPUT); -- generate an input
    OUTPUT := A(INPUT); -- compute an output
            -- print last output if -- interrupted
    PUT (INPUT, OUTPUT);
    when INTERRUPT ⇒
    PUT (INPUT, OUTPUT);
  end loop;
end MAIN;
```

Input–output inconsistencies lead to large redundant test sets, where test outputs are not easily understood because of a dependency on unidentified states. Programs with input–output inconsistencies are not easily testable. In some sense, input–output inconsistencies are a consequence of using von Neumann programming languages [1]. Input–output inconsistencies can provide evidence for program errors [3], [14], [25], but do not imply that the program has faults: a program containing unreachable statements or an infinite loop may not be easily testable, but may be correct.

In this paper we formally define a property called "domain testability" as an extensional property of programs. Programs that are domain testable have no input–output inconsistencies, so in some sense they are are "easily testable." Domain testability is defined in terms of the properties of observability and controllability. Informally, a software component is observable if distinct outputs are observed for distinct inputs: observability is the ease of determining if specified inputs affect the outputs. A software component is controllable if, given any desired output value, an input exists which "forces" the component output to that value: controllability is the ease of producing a specified output from a specified input.

Observability and controllability were first developed for dynamical systems and automata [17]. Dynamical systems have inputs, outputs, states, and state-transition functions. For hardware components, the state-transition functions are more easily identifiable than the state-transition functions

for software components. Controllability and observability properties have been used for specifying hardware testability [19].

Our initial focus is in specifying the observability and controllability properties for two software components: expression procedures (similar to Ada function subprograms), and command procedures (similar to Ada procedure subprograms). The domain-testability properties of these components are defined and generalized in Section II. We define metrics for practically comparing observability and controllability. The properties and metrics are illustrated in Section III. In Section IV we discuss how testability can be assessed from program specifications, and also discuss an experiment which shows that it takes less time to build and test a program developed from a domain-testable specification than a similar program developed from a nondomain-testable specification. In Section V other aspects of software observability and controllability are suggested for future work.

We show in the Appendix how software inputs, outputs, and state transitions can be represented with the direct execution models of denotational semantics [13], [24], [26] to yield a more formal definition of observability and controllability.

II. Domain Testability

Domain testability is defined in terms of the execution semantics of program inputs and outputs. Our model programming language has the syntax of an Ada-like language: a more formal definition of domain testability in terms of denotational semantics is specified in the Appendix.

Domain testability is defined by specifying the semantics of expression evaluation and command execution. Expressions are denoted by functions (also called *expression procedures*). For example, the following function:

```
function SIDE_EFFECT(X: in INTEGER) return
INTEGER is
begin
  GLOBAL_VARIABLE :=
    GLOBAL_VARIABLE + 1;
  return X*GLOBAL_VARIABLE;
end SIDE_EFFECT;
```

shows that the evaluation of an expression depends on the state, and results in a value and a new state.

Commands are denoted by commands (also called *command procedures*). For example, the following command procedure:

```
procedure SIDE_EFFECT(X: in INTEGER;
                      Y: out INTEGER) is
begin
  GLOBAL_VARIABLE :=
    GLOBAL_VARIABLE + 1;
  Y:= X*GLOBAL_VARIABLE;
end SIDE_EFFECT;
```

shows that execution depends on the state, and results in a new state.

A. Observability

An expression procedure F is *observable* if distinct outputs are generated from distinct inputs. Since the evaluation of the output of F is a function of the evaluation of its input *and the state*, observability implies that the output value of F is a function of the input value only.

For example, the following expression procedure is not observable:

```
function F(X: in INTEGER) return INTEGER is
begin
  return X*GLOBAL_VARIABLE;
end F;
```

The reason that it is not observable is that subsequent calls to the function F with the same arguments can yield different results. In an environment where GLOBAL_VARIABLE evaluates to 0, the function evaluation F(3) evaluates to 0. If the environment is changed after this function call where GLOBAL_VARIABLE = 2, then F(3) = evaluates to 6.

In general, given an expression procedure F:

```
function F (E1: in T1; E2: in T2; ... ; En: in Tn)
return TF;
```

F is *observable* if:

$$F(B1, \ldots, Bn) \neq F(A1, \ldots, An) \text{ implies } (B1, \ldots, Bn) \neq (A1, \ldots, An).$$

An execution of a command procedure is observable if distinct outputs are generated from distinct inputs. For example, the following command procedure is not observable:

```
procedure C(X: in INTEGER;
            Y: out INTEGER) is
begin
  Y := X*GLOBAL_VARIABLE;
end C;
```

The reason that it is not observable is that subsequent calls to the procedure with the same arguments can yield different results. In an environment where GLOBAL_VARIABLE = 0, C(3,Y) yields Y = 0. If the environment is changed after this function call where GLOBAL_VARIABLE = 2, then Y = 6.

In general, given a command procedure P:

```
procedure P (E1: in S1; ...En: in Sn;
             V1: out T1;...Vm: out Tm);
```

P is *observable* if:

$$(Y1,\ldots, Ym) \neq (Z1, \ldots, Zm) \text{ implies } (X1, \ldots, Xn) \neq (U1, \ldots, Un)$$

given the executions:

P (X1, ..., Xn, Y1,..., Ym);

and

P (U1,..., Un, Z1, ..., Zm);

for all

(X1, . . . , Xn) and (U1, . . . , Un) in domains (S1, . . . , Sn).

B. Controllability

Given a state *s*, the domain of values of the evaluation for expressions is usually a proper subset of the target type. An evaluation of expression procedure F(E) is *controllable* if, for any state *s*, the domain of values of the evaluation map *equals* the domain of values denoted by its output specification.

For example, in an environment where type POSITIVE denotes the set of values {0,1,2,...,MAX_INTEGER}, and given the expression procedure G:

```
function G(X:in POSITIVE)return POSITIVE is
begin
        return X mod 3;
end;
```

G is observable, because if G(A1) = b1 and G(A1) = b2, then b1 = b2. However, the expression procedure G is not controllable. Even though G returns a subset of type POSITIVE for every input (the set {0,1,2}), there is no set of inputs that evaluates to POSITIVE: {0,1,2} POSITIVE.

If expression procedure F:

```
function F (E1: in T1; E2: in T2; . . . ;
            En: in Tn) return TF;
```

is controllable, then for for all input values of A1, . . . , An:

{ all evaluations of F (A1, . . ., An) } = TF

If F is observable and controllable, then F is an *onto* function.

Observability is not required for controllability: programming languages which define functions having no input variables may be controllable, but not observable. For example, given

```
function RANDOM() return INTEGER;
```

if the set of all evaluations of RANDOM() covers all values in INTEGER, then RANDOM is controllable.

An execution of a command procedure is controllable if, for any state *s*, the domain of each output value equals its type. Given command procedure P:

```
procedure P (E1: in S1; . . .En: in Sn;
             V1: in T1;. . .Vm: out Tm);
```

P is controllable if:

{(Y1,. . ., Ym)} = (T1, . . ., Tm)

given all executions:

P (X1, . . ., Xn, Y1,. . ., Ym);

for all (X1, . . . , Xn) in (S1, . . . , Sn).

C. Domain Testability

Since testing assesses the compliance of a program to intended input–output specifications, it is necessary to identify the data characteristics of the *explicit* test inputs and outputs (inferred from specific identifiers in the program) and *implicit* test inputs and outputs (inferred from component *states*—not from specific identifiers). Part of the difficulty with testing is concerned with identifying and specifying implicit program inputs and outputs from states as auxiliary inputs and outputs that can be controlled and observed, so that test results do not exhibit the input–output inconsistencies defined in Section I.

The definitions of observability and controllability immediately establish the following:

- *Corollary:* If an expression (command) procedure is not observable, then its evaluation (execution) may exhibit output inconsistencies. If an expression (command) procedure is not controllable, then its evaluation (execution) can exhibit input inconsistencies
- *Definition:* An expression (command) procedure is *domain testable* if it is observable and controllable.

Programs that are not *domain testable* can be difficult to test. For example, the following is a specification for a common expression procedure which returns a random floating point number between zero and one according to a uniform probability distribution:

```
function RANDOM(): return REAL;
```

This function is not observable, since it has no input (in fact, it is not supposed to be observable by *design*). It is desired to be controllable. Testing this function is difficult without any additional knowledge about its internal states.

Most function and expression procedures are not *a priori* observable or controllable. This means that *without any further modification*, effective and adequate testing is difficult to assess—there may be input or output inconsistencies. In order to test, many software engineers modify the program by explicitly creating additional program inputs and outputs which denote the implicit program states. This improves testability by making these states observable and controllable: the program can then be input to a test driver. One problem with this modification is that the tested component may be different from the actual deployed component.

We formalize this program modification process with the following:

- *Definition:* Domain testability refers to the ease of modifying a program so that it is observable and controllable.

The program modifications required to achieve domain testability are called *extensions*. Observable extensions introduce program inputs based on implicit states; controllable extensions modify outputs.

D. Observable Extensions

Let F denote the expression procedure definition:

```
function F (E: in T) return TF;
```

F has an observable extension with observability index *n* if there exists an observable expression procedure FO, with definition:

```
function FO (E: in T ; E1: in T1;
             E2: in T2; . . .; En: in Tn)
return TF;
```

such that for all inputs of F there exist inputs for FO, such that:

F(E) = FO (E, E1, E2, . . . ,En)

For example, the following expression procedure F:

```
function F(X: in INTEGER) return INTEGER is
begin
        return X*GLOBAL_VARIABLE;
end F;
```

has an observable extension FO:

```
function FO (X: in INTEGER; G: in INTEGER)
return INTEGER is
begin
        return X*G;
end FO;
```

FO is observable, since for given inputs (A1, A2) and (B1, B2), F(A1, A2) $\neq$ F(B1, B2) implies (A1, A2)$\neq$ (B1, B2).

A command procedure P:

```
procedure P (E: in T; O: out TP);
```

has an observable extension with observability index *n* if there exists an observable command procedure PO, with definition:

```
procedure PO (E: in T; E1: in S1; ...En: in Sn;
              O: out TP; OE: TE);
```

such that for all inputs for P there exists a set of inputs for PO such that after the execution of

```
P (E, O);
```

and

```
PO (E, E1, ..., En, OE);
```

then

```
O = OE.
```

For example, command procedure C:

```
procedure C(X: in INTEGER; Y: out INTEGER) is
begin
    Y := X*GLOBAL_VARIABLE;
end C;
```

has an observable extension of command procedure CO:

```
procedure CO (X: in INTEGER; G: in INTEGER;
Y: out INTEGER) is
begin
    Y := X*G;
end CO ;
```

E. Controllable Extensions

Given:

```
function F (E1: in T1; ... ; En: in Tn) return TF;
```

F has an controllable extension if there exists a type TC that is a subset of TF and a controllable expression function FC, with definition:

```
function FC (E: in T ; E1: in T1;
    E2: in T2; ... ; En: in Tn)
return TC;
```

such that for all inputs and any state s for F, there exists a state s* for FC such that:

```
F(E1, ... ,En) = FC (E1, ... ,En)
```

For example, the following expression procedure GC is a controllable extension of G (defined in Section II-C):

```
type SMALL is POSITIVE range 0..2;
function GC (X: in POSITIVE) return SMALL is
begin
    return SMALL'(X mod 3);
end GC;
```

GC is controllable, since the set of values it returns is the same set denoted by the type SMALL:

```
{GC (E) for all E} = { 0,1,2 } = SMALL
```

Moreover, for all states s for G, there exists an s^* for GC so that

```
G(E) = GC (E)
```

A command procedure P:

```
procedure P (E: in T; O1: out S1; ... ;
             Om: out Sm);
```

has a controllable extension with controllability index m if there exists types T1, ... ,Tm that are a subset of S1, ... ,Sm, respectively, and a controllable command procedure PC, with definition:

```
procedure PC (E: in T; OC1: out T1;... ;
              OCm: out Tm);
```

such that for all inputs and any state s for P, there exists a state s^* for PC such that:

```
(O1,...,Om) = (OC1,...OCm)
```

given the executions,

```
P(A,B1,...Bm);
```

and

```
PC (A,BC1,...BCm);
```

Controllable extensions depend on the richness of type-domain definitions. For conventional languages these definitions may be modeled by inductive assertions [10]. For functional languages, type domains are function domains [1].

F. Measures of Domain Testability

We formally define domain-testability measures in terms of controllable and observable extensions.

Observability is the ease of determining if specified inputs effect the outputs; *controllability* is the ease of producing a specified output from a specified input.

Given the following expression procedure:

```
function F (E: in T) return TF;
```

and observable extension with observability index n:

```
function FO (E: in T ; E1: in T1; E2:
in T2; ... ; En: in Tn) return TF;
```

For an exhaustive test, the extra number of test cases required would be a multiple of

$$|T1| * \ldots {}^*|Tn| \leq |T_{max}|^n.$$

Here, $|Tk|$ denotes the cardinality of the domain Tk, and $|T_{max}|$ is the maximum cardinality of the domain over all input types. Even though this value can be infinite for dynamic data structures such as strings and lists, during testing we enforce pragmatic limits to dynamic data structures.

The extra testing work associated with the n extra inputs depends on the type; we can normalize this factor by considering the effective number of extra binary inputs:

$$Ob = \log_2(|T1|^* \ldots {}^*|Tn|)$$

where Ob is the measure of observability and corresponds to the number of extra binary inputs required to make a program observable. The measure of observability for command procedures is similar.

Given the command procedure P

```
procedure P (E: in T; O1: out S1; ... ;
Om: out Sm);
```

and controllable extension with controllability index m

```
procedure PC (E: in T; OC1: out T1; ... ;
OCm: out Tm);
```

we define

$$Ct = \log_2(|T1|^* \ldots {}^*|Tm|)$$

where Ct is the measure of contollability and corresponds to the number of binary inputs which must be modified to make a program controllable. We note that

$$0 \leq Ob \leq n * \log_2(|T_{max}|)$$
$$0 \leq Ct \leq m * \log_2(|T_{max}|).$$

These last two expressions are similar in form to the software science volume complexity metrics [25]. The lower bounds are obtained if the program is observable or controllable.

For imperative programming languages, we have the following results that allow us to easily compute Ob and Ct so that they can be compared. These results follow from the definition of the logarithm and combinatorial (multiplicative) nature of additional testing inputs.

1) Sequence: The measures of observability and controllability of a sequence (composition) of commands (expressions) are the sum of the measures of the components. In other words,

$$Ob(C1; C2) = Ob(C1) + Ob(C2)$$

$$Ct(C1; C2) = Ct(C1) + Ct(C2).$$

2) Selection: Given an observable predicate E, the measures of observability and controllability of a selection of commands are a function of the measures of the components. In other words,

Ob (if E then C1; else C2 end if;)
= Ob(C1) + Ob(C2)
Ct (if E then C1; else C2 end if;)
= Ct(C1) + Ct(C2) + Ct(E)

3) Iteration: Given an observable predicate E, the measures of observability and controllability of the iteration of a command are a function of the measures of the components. In other words,

Ob (while E loop C1; end loop;)
= Ob(C1)
Ct (while E loop C1; end loop;)
= Ct(C1) + Ct(E)

III. Examples

A. Text Formatter

Naur [22] investigated the application of formal program proving in the construction of a simple formatting program. This program was subsequently tested and analyzed by Good-enough [11]. Meyers [21] translated the program to PL/I and had it tested by a number of professional programmers. Basili [2] also used this program in a similar study of test effectiveness. In a review of the Meyers study, House [15] stated that the program was virtually untestable, since it was so poorly specified.

Part of the problem was in its reliance on the side-effects of input and output. For example, the PL/I output character procedure is shown in Fig. 2.

This procedure has the side-effects (among others) of writing a character to the output stream, and changing an implicit column number, line number, or possibly a page number. If the command results include text output displays, subsequent executions of PCHAR(X) result in different results, so that PCHAR is not observable. In order to test it, an observable and controllable extension such as:

```
PCHAR: PROCEDURE (C);
DECLARE C CHAR,
OUTLINE (LINESIZE) CHAR STATIC INIT ((LINESIZE) ( ')),
I FIXED DECIMAL (3) STATIC INIT (1);
DECLARE SOUTPUT FILE STREAM;
IF (C=LINEFEED)
 THEN DO;
    PUT FILE (SOUTPUT) SKIP EDIT (STRING (OUTLINE))
    (A (LINESIZE));
OUTLINE = ' ';
I = 1;
  END;
 ELSE DO;
    OUTLINE (I) = C;
    I = I + 1;
  END;
END;
END;
```

Fig. 2. Output character procedure for formatting example.

```
procedure PCHART(COLUMN: in COL-
UMN_TYPE;
  LINE: in LINE_TYPE;
  PAGE: in PAGE_TYPE;
  FILE: in FILE_TYPE
  ITEM: in CHARACTER;
  STRING: out STREAM);
```

can be defined so that the execution of this command would yield the same result for equal arguments. The execution of this procedure will write all literal "printable" characters to a specified output stream. In this example, the observability index is 4 and the controllability index is 1. Given the following domain cardinalities for the test,

|COLUMN_TYPE| = 30
|LINE_TYPE| = 66
|PAGE: in PAGE_TYPE| =10
|FILE_TYPE| = 2
|CHARACTER| = 84
$|STREAM| = 200^{84}$

The STREAM cardinality assumes that a stream is 200 characters. The observability measure is 21.7 and the controllability measure is 642.

B. Calendar Program

This program, initially proved correct, was later discovered to have several errors. This program is discussed in Geller [10] and Lamport [18]. The program is specified in ALGOL-W in Fig. 3.

The procedure is not observable, since there is no output. It is also not controllable: for example, in the procedure, day1 is allowed to be *negative*; allowing negative-valued days indicates an error in the specification.

An observable and controllable extension can be given as:

```
procedure calendart
        (day1, day2 in : DAY_TYPE;
        month1, month2 : in MONTH_TYPE;
        year: in YEAR_TYPE; days_between:
        out POSITIVE_INTEGER);
```

This modification required five new inputs and one new output. Given the following domain cardinalities for a test:

```
procedure calendar (integer value day1, month1, day2, month2, year);
begin
integer days;
if month2 = month1 then days := day2 - day1
        comment if the dates are in the dame month, we can
                compute the number of days between them
                immediately
else
begin
integer array (1::12);
daysin(1):=31; daysin(3):=31; daysin(4):=31;
daysin(5):=31; daysin(6):=31; daysin(7):=31;
daysin(8):=31; daysin(9):=31; daysin(10):=31;
daysin(11):=31; daysin(12):=31;
if ((year rem 4) := 0) or
    ((year rem 100):= 0) and (year rem 400):= 0)
    then daysin(2) := 28
    else daysin(2) := 29;
    comment   set daysin (2) according to whether or
          not year is a leap year;
days := day2 + daysin(month1) - day1);
    comment   this gives (the correct number of days-
          days in complete intervening months);
for I := month1 + 1 until month2 - 1 do
    days := daysin(I) + days;
    comment   add in the days in complete intervening
        months;
end;
write ( days );
end;
```

Fig. 3. The calendar program.

$|DAY\_TYPE| = 31$

$|MONTH\_TYPE| = 12$

$|YEAR\_TYPE| = 300$

$|POSITIVE\_INTEGER| = 2^{31}$

The observability measure is 25.3 and the controllability measure is 31.

C. 91 Function

This function was also discussed by Geller [10] as an example of a program that was proved correct. The function returns the number 91 when X⩽ 101 or returns X−10 when X>100. Its full specification is:

```
function F(X: in INTEGER) return INTEGER is
begin
      if X>100 then return X−10
                        else return F(F(X+11));
    end if;
end;
```

The function is observable. It is not controllable, since it does not return all integers. A controllable extension can be defined with the auxiliary type:

```
type RETURN_TYPE is new INTEGER range
91..MAX_INT;
```

If MAX_INT has a value of 2^{31}, then the controllability measure is

$$\log_2(2^{31} - 90) = 30.99.$$

D. Concurrent Tasks

This example demonstrates a common testing problem seen in transaction processing. The testing anomaly is discussed in Eswaren *et al.* [4] and Gardarin and Valduriez [9]. In Ada-like syntax, we have:

```
procedure DBMS is
      type T is ...
      DATA: T;
   procedure READ (D: in T; X: out T)
      is .. end; - - read D into X
   procedure WRITE (X: in T; D: out T)
      is .. end; - - write X onto D
   function UPDATE (X: in T) return T
      is ... end;
   task type TRANSACTION;
   task body TRANSACTION is
        Y: T;
        begin
              READ (DATA, Y);
              ...
              Y := MODIFY(Y);
              ...
              UPDATE (Y,DATA);
        end TRANSACTION;

    A, B: TRANSACTION;
begin
      null;
end DBMS;
```

If READ and UPDATE are observable and controllable in a sequential execution environment, they are not observable and controllable in a concurrent execution environment because of the side-effects due to the scheduling of operations. For example, suppose while task A starts reading DATA, task B has already read and updated DATA. Any subsequent reference to DATA by A yields different results. Consequently, READ is not observable. Moreover, any updates made by B are lost. Consequently, UPDATE is not controllable. Procedures READ and WRITE have observable and controllable extensions if the tasks can be provided with serializable schedules [9]: read and updating must be encapsulated so that these operations appear as if executing sequentially. Observable and controllable extensions may introduce the states GLOBAL_TIME and LOCK? as extra input and outputs.

IV. Discussion

A. Software Requirements, Specifications, and Domain Testability

A key principle of software engineering is that one of the best ways to prevent errors and improve the effectiveness of testing is by having a good specification and good set of functional requirements.

Knowledge of module inputs, outputs, and states does not require executable programs: they should be stated in a functional requirement or design specification. Consequently, testability can be assessed from program specifications. By incorporating the ideas of observability and controllability into the requirement and specification process, hidden program states are made explicit in the specification. This means that the testing task of "identifying the input and output, states, and state transitions" and modifying the program so that these characteristics are observable and controllable is simplified.

By building software from domain-testable specifications, software builders do not need to waste time on input and output inconsistencies and the resultant program modifications.

Student	Translation Time	Redesign Time	Redesign Time / Translation Time
Rivera	4	2.5	0.625
Hendrie	8	5	0.625
Finegan	8	5	0.625
Huang	11	9	0.818181818
Aoki	13	11	0.846153846
Ip	5	4.75	0.95
Chieu	10	10	1
Kornish	7.5	10.5	1.4
Average	8.3125	7.21875	0.861166958
Std. Deviation	2.99	3.05	0.26

Fig. 4. Text formatter experiment: results.

A domain-testable program reduces the redundancy of test sets (and consequently, the amount of testing). The test results are also more readable, since implicit states are made explicit.

B. Advantages of Domain Testability: An Experiment

As observed above, programs developed from domain-testable specifications should be "easier" to test. The following experiment shows that it takes less time to build and test a program developed from a domain-testable specification than a similar program developed from a nondomain-testable specification.

Eight undergraduate students were asked to build and test two Text Formatting programs in a language of their choice. They were given two references on the Text Formatter program ([15] and [21]) to help them develop their specifications.

For the first program, all students were asked to use the Meyers PL/I program as a specification, and then translate the PL/I program into the language of their choice. As observed in Section III-A, this PL/I program contains components which are neither observable nor controllable. They were then asked to test the translated program to determine if their translation uncovered any of the 15 errors documented in [21]. They were not required to debug the program. The students recorded the number of hours required for this build-and-test activity.

The second program involved a total redesign of the Text Formatter. The students were split into two groups: Each student in Group A developed a domain-testable specification of the Text Formatter: their functions and procedures were specified to be domain testable, as indicated in Section III-A. Each student in Group B did not develop a domain-testable specification. They were then asked to test the redesigned program to determine if their program uncovered any of the 15 errors documented in [21]. The students also recorded the number of hours required for this build-and-test activity.

Our hypothesis is that there is a measurable difference in time for the build-and-test activities between domain testable- and nondomain-testable specifications.

The results of the experiment are indicated in Fig. 4.

As expected, there are large variances in the reported development time, One reason for these variances may be due to a student bias in reporting times. In order to eliminate this bias, we compute a normalizing ratio

$$R = \frac{\text{Redesign Time}}{\text{Translation Time}}$$

Student	Translation Time	Redesign Time	Redesign Time / Translation Time
Aoki	13	11	0.846153846
Finegan	8	5	0.625
Huang	11	9	0.818181818
Rivera	4	2.5	0.625
Average	9	6.875	0.728583916
Std. Deviation	3.92	3.84	0.12

Fig. 5. Text formatter experiment: Group A results.

Student	Translation Time	Redesign Time	Redesign Time / Translation Time
Ip	5	4.75	0.95
Kornish	7.5	10.5	1.4
Chieu	10	10	1
Hendrie	8	5	0.625
Average	7.625	7.5625	0.99375
Std. Deviation	2.06	3.11	0.32

Fig. 6. Text Formatter experiment: Group B results. From Fig. 5 we see that the average value of R for Group A is 0.72, and that here the average value of R for Group B is 0.99. This indicates that it takes on the average 27% less time to build and test a program developed from a domain-testable specification than it takes to build and test a similar program developed from a nondomain-testable specification.

which provides a normalized measure of productivity for each individual student.

In this experiment, the average value of R is 0.87; this means that the time required to develop a "testable" program from the original PL/I specification was, in most cases, at least as great as the time required to develop a redesigned program. This was noted informally in [15].

When we display the results in Fig. 4 by group, we see the advantages of a domain-testable specification. This is seen in Fig. 5 for Group A, and in Fig. 6 for Group B.

C. Applying the Measures of Observability and Controllability

The observability and controllability measures of a domain-testable component are zero. For components that are not domain testable, the observability and controllability measures can provide a practical normalized metric for assessing the work involved in modifying a program to be domain testable. It is important to note that these metrics indicate the testing effort only so far as the effort required to modify a component to a domain-testable form.

The observability and controllability measures can also be used to develop a normalized metric for assessing the work involved in modifying a system to be domain testable according to the properties in Section II-C; observability and controllability increase linearly with the number of components.

The metrics can be calibrated by building a database of program components coupled with the time used to make these components domain testable. In this way, the metrics can be used indirectly to assess testing time.

V. Conclusions and Future Work

A computer program can be modeled as a complex state machine which transforms inputs to outputs. In a liberal

reading of the second law of thermodynamics, since a "perfect" machine does not exist, it follows that a "perfect" computer program does not exist either. The goal of testing is less demanding than the goal of producing a "perfect" program: the testing goal is to determine the likelihood of a program having errors. This goal can be met if we understand the meaning of software testability, as well as software computability.

The significant contribution of this paper was to formally investigate the meaning of software testability. We defined a new concept—domain testability—by applying the concepts of observability and controllability to programs. We showed that a domain-testable program does not exhibit any input–output inconsistencies. Consequently, domain-testable programs will support small test sets, where test outputs are easily understood.

We also defined new metrics that can be used to assess the level of effort required in order to modify a program so that it is domain testable.

Several other areas suggest themselves for future research:

- *Domain Testability* for other programming structures: Can observability and controllability be defined for programming structures that are not restricted to commands or expressions, like definitions, generic units and objects? Are the denotational methods described in the Appendix applicable?
- *Observability and Continuity*: Watkins [28] discusses a testing technique based on a predictor-corrector strategy, which is based on the continuity properties of a real-valued expression procedure. Can observability be extended to include a topology defined on an input domain so that the output is a continuous function of the input?
- *Domain Testability and Functional Dependencies*: Is there a relationship between the transitive functional dependencies (used, for example, in normalizing databases [9]) and the functional dependencies between inputs and outputs of an observable component?

APPENDIX

Denotational semantics provides a rigorous mathematical description of the domains and function spaces in applicative, conventional (von Neumann), and functional programming languages [1]. For our purposes, we explicitly specify the input, output, and state transformations for our model language to define a *direct semantics*. The semantics of most conventional programming languages are usually specified in terms of a *continuation semantics*. Continuations are used to represent the meanings of constructs that depend on the rest of the program: state transformations are specified indirectly [13]. In our discussion of testability it is easier to consider direct semantics.

A. Specification of Test Execution Semantics

A direct semantics requires the specification of syntactic domains, syntactic clauses, semantic domains, semantic functions, and semantic clauses. Domains are sets that support recursive definition [24].

Syntactic domains and clauses specify the form of our language. We assume that the following syntactic domains are defined:

Ide
The domain of programming language identifiers includes I, P, F, O, . . .

Type
The domain of types supported by the programming language includes INTEGER, FLOAT, . . .

Exp
The domain of expressions.

Com
The domain of commands.

Def
The domain of definitions.

In the sequel we denote

D ∈ **Def**
T1, T2, . . . , Tn, . . . T^*, . . . ∈**Type**
F, P, I1, I2, . . . , In, . . . O1, O2, . . . , On, . . . ∈**Ide**

Here, $a \in S$ denotes that a is an element of a set S. Syntactic clauses are used to specify the form of these domains. We specify the forms of n-ary expression procedures and command procedures in a variant of BNF (we follow the notation specified in [13], [26]).

Expression procedure abstracts define "functions" for n arguments, $n \geq 0$:

D::= function F (I1: in T1; I2: in T2; . . . , In: in Tn;) return T^*;

Similarly, command procedure abstracts define command procedures for n input arguments and m output arguments, $n, m \geq 0$:

D::= procedure P (I1: in T1; . . . , In: in Tn; O1: out $T1^*$; . . . Om: out Tm^*);

The syntax of expressions and commands includes the evaluation of expression procedures and the execution of command procedures: For

E, E1, E2, . . . , En, . . . ∈**Exp**
E ::= F () | F(E1) | F(E1, E2) | . . . | F(E1, E2, . . . , En) | . . .
C ::= P(); | P(E1); | P(O1); | P(E1, O1); | . . . | P(E1, . . . , En, O1, . . . , Om); | . . .

Semantic domains and clauses specify the denotations of our language. We assume that the following semantic domains are defined:

Err
A domain of error values.

Val
This denotes the domain of values. We assume that the domain of values includes the domain **Err** of error values.

Env = Ide → Val

The environment is the domain of all functions from the domain of identifiers to the domain of values. To simplify the discussion, we assume that all identifiers are bound to a value (which may be an error value). In our notation, AB denotes the set of functions from A to B.

TypeEnv = Ide → Type

The environment of types is the domain of all functions, from the domain of identifiers to the domain of types.

State = [Env + TypeEnv] × Input × Output + Err

The states are denoted by the domain of all 3-tuples (u, i, o), where u∈**Env + TypeEnv, i** ∈ **Input** is an input domain, and **o** ∈ **Output** is an output domain. The state also includes the domain **Err**, which is used to capture program errors. State transformations are caused by the execution of commands and evaluation of expressions. In our notation, A × B denotes a set of pairs (the cartesian product of sets A and B) and A+B denotes set union.

Semantic functions and clauses provide the semantics of our language by specifying the correspondence between the syntax and literal values (the denotations). We assume that the following semantic functions are defined: The semantics of expression evaluation is specified by

E **: Exp → (State → (Val × State))**

This means that the evaluation of an expression depends on the state, and results in a value and a new state. For example, the evaluation of the expression procedure F(E) is denoted by:

E [F(E)] s = (v, s′), for v ∈ **Val** and s, s′ ∈ **State**

To reduce the number of parentheses, it is convenient to place brackets ([ ]) around elements of a syntactic domain.

In most programming languages, the result of evaluating an expression procedure affects the state, as in

```
function SIDE_EFFECT(X: in INTEGER)
        return INTEGER is
begin
        GLOBAL := GLOBAL + 1;
        return X*GLOBAL;
end SIDE_EFFECT;
```

For convenience, we define a projection function "getval":

getval: **Val** × **State** → **Val**

Command execution depends on the state, and results in a new state:

C: **Com → State → State**

For example, the evaluation of the command procedure P(E, O) is denoted by:

C [P(E, O);] s = s′, for s, s′ ∈ **State**

Context evaluation for expressions depends on the state (the **TypeEnv**) and results in the type of the expression:

T **: Exp → State →Type**

For example, the context evaluation of the expression procedure F(E) is denoted by:

T [F(E)] s = t, where s ∈ **State** and t ∈ **Type**

B. Observability

Given an expression procedure defined by definition:

D::= function F(I: in T) return T*;

where F is observable if for any s_1 and s_2, distinct outputs are generated from distinct inputs:

getval (***E*** [F(E_1)] s_1) = getval (***E*** [F(E_2)] s_2)
if
getval (***E*** [E_1] s_1) = getval (***E*** [E_2] s_2)

We note that, in general, the evaluation of the output of F is a function F^* of the evaluation of the input and the states, such that for ***E*** [E] s = (i, s^*) and ***E*** [F(E)] s = (o, s^+)

o = F^* (i, s^*)

In other words, the evaluation of the function procedure may have a side effect in changing the state.

Observability implies that the output value of F is a *function* ***F*** of the input expression:

getval (***E*** [F(E)] s) = ***F*** (getval (***E***[E] s))

or

o = ***F*** (i)

For example, the following expression procedure is not observable:

```
function F(X: in INTEGER) return INTEGER is
begin
        return X*GLOBAL_VARIABLE;
end F;
```

Subsequent calls to the function F with the same arguments can yield different results. In an environment where GLOBAL_VARIABLE = 0, F(3) = 0. If the environment is changed after this function call where GLOBAL_VARIABLE = 2, then F(3) = 6. We note that:

getval (***E*** [F(E)] s)
= getval(***E*** [GLOBAL] s)*
getval(***E*** [E] s)
= F^* (g, e) = ***F*** (e)

Observability can be defined for an expression procedure:

D::= function F (I1: in T1; I2: in T2; . . . , In:
in Tn;) return T*;.

F is observable if distinct outputs are generated from distinct inputs:

getval (***E*** [F($E1_1$, $E2_1$, . . . , En_1)] s1)
= getval (***E*** [F($E1_2$, $E2_2$, . . . , En_2)] s_2)
if
getval (***E*** [$E1_1$] s_1) = getval (***E*** [$E1_2$] s_2)
getval (***E*** [$E2_1$] s_1) = getval (***E*** [$E2_2$] s_2)
. . .
getval (***E*** [En_1] s_1) = getval (***E*** [En_2] s_2)

Similar definitions of observability are defined for commands. The execution of a Command Procedure is observable if distinct outputs are generated from distinct inputs. For explicit outputs bound in the procedure abstract, the execution of C(E, O) is output observable where for C(E_1, O_1) and C(E_2, O_2):

E [O_1] s_1 = ***E*** [O_2] s_2 if getval (***E*** [E_1] s_1)
= getval (***E*** [E_2] s_2)

C. Controllability

Given a state *s*, the domain of values of the evaluation map for expressions is usually a subset of the target type:

T [F] s⊇ {getval (***E*** [F(E)] s) | for all E ∈ **Exp** }

For example, given the expression procedure G:

```
function G(X: in POSITIVE) return POSITIVE is
begin
        return X mod 3;
end;
```

G is observable, since G returns a subset of the type POSITIVE for every state s:

$$\text{POSITIVE} \supseteq \{\text{getval}(\boldsymbol{E}[\text{G(E)}]\ s), \text{ for all } E \in \textbf{Exp}\} = \{0, 1, 2\}$$

An evaluation of expression procedure F(E) is controllable if, for any state s, the domain of values of the evaluation map equals its type:

$$\boldsymbol{T}[\text{F}]\ s = \{\text{getval}(\boldsymbol{E}[\text{F(E)}]\ s), \text{ for all } E \in \textbf{Exp}\}$$

For example, the expression procedure G is not controllable.

D. Observable Extensions

Let expression procedure F have definition abstract:

D:: = function F (E: in T) return T^*;

F has an observable extension if there exists an observable expression procedure Fo, with definition abstract:

D:: = function Fo (E_1: in T_1; E_2: in T_2; ... ; E_n: in T_n) return T^*;

such that for all $E \in$ **Exp** and any state s, there exists a state s^* such that:

$$\text{getval}(\boldsymbol{E}[\text{F(E)}]\ s) = \text{getval}(\boldsymbol{E}[\text{Fo}(E_1, E_2, ..., E^n)]\ s^*)$$

For example, the following expression procedure Fo is an observable extension of F:

```
function Fo (X: in INTEGER; G: in INTEGER)
return INTEGER is
begin
        return X*G;
end Fo;
```

Fo is observable, since:

$$\text{getval}(\boldsymbol{E}[\text{Fo}(A_1, A_2)]\ s) = \text{getval}(\boldsymbol{E}[A_1]\ s) * \text{getval}(\boldsymbol{E}[A_2]\ s)$$
$$= \text{getval}(\boldsymbol{E}[\text{Fo}(B_1, B_2)]\ s) = \text{getval}(\boldsymbol{E}[B_1]\ s) * \text{getval}(\boldsymbol{E}[B_2]\ s)$$

as long as both

$$\text{getval}(\boldsymbol{E}[A_1]\ s) = \text{getval}(\boldsymbol{E}[B_1]\ s)$$
$$\text{getval}(\boldsymbol{E}[A_2]\ s) = \text{getval}(\boldsymbol{E}[B_2]\ s)$$

Moreover, for all states we can set:

$$\text{getval}(\boldsymbol{E}[\text{GLOBAL}]\ s) = \text{getval}(\boldsymbol{E}[E_2]\ s)$$

so that

$$\text{getval}(\boldsymbol{E}[\text{F(E)}]\ s) = \text{getval}(\boldsymbol{E}[\text{GLOBAL}]\ s) * \text{getval}(\boldsymbol{E}[\text{E}]\ s)$$
$$= \text{getval}(\boldsymbol{E}[E_1]\ s) * \text{getval}(\boldsymbol{E}[E_2]\ s)$$
$$= \text{getval}(\boldsymbol{E}[\text{Fo}(E_1, E_2)]\ s)$$

Let command procedure P have the definition abstract:

D:: = procedure P (E: in T; O: out T^*);

P has an observable extension of P if there exists an observable command procedure P^o, with the definition abstract:

D:: = procedure P^o (E_1: in T_1; E_2: in T_2; ... ; E_n: in T_n; O^*: out T^*);

such that for all $E \in$ **Exp** and any state s, there exists a state s^* such that

$$\boldsymbol{E}[O^*]\ r^* = \boldsymbol{E}[O]\ r$$

where

$$r^* = \boldsymbol{C}[P^o(E_1, E_2, ..., E_n, O^*);]\ s^*$$
$$r = \boldsymbol{C}[P(E, O);]\ s$$

E. Controllable Extensions

F has a controllable extension if there exists a controllable expression procedure F^c, with definition abstract:

D:: = function F^c (E_1: in T_1; E_2: in T_2; ... ; E_n: in T_n) return T^c;

such that for all $E \in$ **Exp** and any state s, there exists a state s^* such that

$$\text{getval}(\boldsymbol{E}[\text{F(E)}]\ s) = \text{getval}(\boldsymbol{E}[F^c(E_1, E_2, ..., E_n)]\ s^*)$$

For example, the following expression procedure G^c is a controllable extension of G (defined in Section II-C):

```
type Small is POSITIVE range 0..2;
function Gc (X: in POSITIVE) return SMALL is
begin
        return SMALL'(X mod 3);
end Gc;
```

F^c is controllable, since:

$$\{\text{getval}(\boldsymbol{E}[F^c(E)]\ s) \mid \text{for all } E \in \text{Exp}\} = \{0, 1, 2\} = \text{SMALL} = \boldsymbol{T}[F^c]\ s$$

Moreover, for all states s:

$$\boldsymbol{E}[\text{G(E)}]\ s = \boldsymbol{E}[G^c(E)]\ s$$

P has a controllable extension if there exists a controllable command procedure P^c, with definition abstract:

D:: = procedure P^c (E_1: in T_1; E_2: in T_2; ... ; E_n: in T_n; O^*: out T^+);

such that for all $E \in$ **Exp** and any state s, there exists a state s^* such that:

$$\boldsymbol{E}[O^*]\ r^* = \boldsymbol{E}[O]\ r$$

where

$$r^* = \boldsymbol{C}[P^c(E_1, E_2, ..., E_n, O^*);]\ s^*$$
$$r = \boldsymbol{C}[P(E, O);]\ s$$

ACKNOWLEDGMENT

The author wishes to thank E. J. Finegan for help in revising this paper and the reviewers for their very helpful suggestions.

REFERENCES

[1] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Comm. ACM*, vol. 21, no. 8, pp. 613–641, Aug. 1978.

[2] V. Basili and R. W. Selby, "Comparing the effectiveness of software testing strategies," *IEEE Trans. Software Eng.*, vol. SE-13, pp. 1278–1296, Dec. 1987.

[3] B. Beizer, *Software Testing Techniques*. New York: Van Nostrand Reinhold, 1983.

[4] K. P. Eswaran, J. N. Gray, R. A. Lorie, and L. L. Traiger, "The notion of consistency and predicate locks in a database system," *Comm. ACM*, vol. 19, no. 11, pp. 624–633, Nov. 1976.

[5] R. Freedman, *Programming with APSE Software Tools*. Princeton, NJ: Petrocelli, 1985.

[6] R. Freedman, M. Shooman, M. Corcoran, and R. Frail, *A Knowledge-Based Testing Assistant* (Polytechnic Notes on Artificial Intelligence, no. 5). Brooklyn, NY: Polytechnic Univ., 1987.

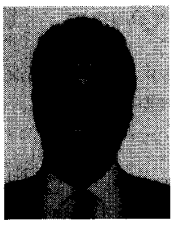
[7] R. Freedman, M. Shooman, M. Corcoran, and R. Frail, "An expert system for software component testing," presented at the Univ. Seminar on Artificial Intelligence, Unisys Corp., Nice, France, Oct. 1987.

[8] H. Fujiwara, *Logic Testing and Design for Testability*. Cambridge: MIT Press, 1985.

[9] G. Gardarin and P. Valduriez, *Relational Databases and Knowledge Bases*. New York: Addison–Wesley, 1989.

[10] M. Geller, "Test data as an aid in proving program correctness," *Comm. ACM*, vol. 21, no. 5, pp. 368–375, May 1978.

[11] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 156–173, June 1975.
[12] J. B. Goodenough, "Ada compiler validation: An example of software testing theory and practice," Wang Institute of Graduate Studies, Tech. Rep. TR-86-07, July 1986.
[13] M. Gordon, *The Denotational Description of Programming Languages*. New York: Springer-Verlag, 1979.
[14] B. Hetzel, *The Complete Guide to Software Testing*, 2nd ed. Wellesley, MA: QED Info. Sci., 1988.
[15] R. House, "Comments on program specification and testing," *Comm. ACM*, vol. 23, no. 6, pp. 324–331, June 1980.
[16] W. E. Howden, "A survey of dynamic analysis methods," in *Tutorial: Software Testing and Validation Techniques*, 2nd ed., E. Miller and W. E. Howden, Eds. Washington, DC: IEEE Computer Soc. Press, 1981, pp. 209–231.
[17] R. E. Kalman, P. L. Falb, and M. A. Arbib, *Topics in Mathematical System Theory*. New York: McGraw-Hill, 1969.
[18] L. Lamport, "On the proof of correctness of a calendar program," *Comm. ACM*, vol. 22, no. 10, pp. 554–556, Oct. 1979.
[19] E. J. McCluskey, *Logic Design Principles*. New York: Prentice-Hall, 1986.
[20] G. J. Meyers, *The Art of Software Testing*. New York: Wiley, 1979.
[21] G. J. Meyers, "A controlled experiment in program testing and code walkthroughs/inspections," *Comm. ACM*, vol. 21, no. 9, pp. 760–768, Sept. 1978.
[22] P. Naur, "Programming by action clusters," *BIT*, vol. 9, no. 3, pp. 250–258, 1969.
[23] S. Reddy, "Easily testable realizations for logic functions," *IEEE Trans. Comput.*, vol. C-21, pp. 1183–1188, Nov. 1972.
[24] D. Scott and C. Strachey, "Toward a mathematical semantics for programming languages," in *Proc. Symp. on Computers and Automata*, J. Fox, Ed. New York: Polytechnic Instit. of Brooklyn Press, 1971.
[25] M. L. Shooman, *Software Engineering: Design, Reliability, and Management*. New York: McGraw-Hill, 1983.
[26] J. Stoy, *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*. Cambridge: MIT Press, 1977.
[27] L. Tsalalikhin, "Dialog with a tester (architecture and functions of one unit test facility)," in *Proc. 1986 Workshop on Software Testing*. Washington, DC: IEEE Computer Soc., 1986, pp. 51–60.
[28] M. Watkins, "A technique for testing command and control software," *Comm. ACM*, vol. 25, no. 4, pp. 228–232, Apr. 1982.
[29] E. J. Weyuker and S. Rapps, "Selecting software tests using data flow information," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 367–375, Apr. 1985.
[30] E. J. Weyuker, "Axiomatizing software test data adequacy," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 1128–1138, Dec. 1986.
[31] S. H. Zweben and J. S. Gourlay, "On the adequacy of Weyuker's test data adequacy axioms," *IEEE Trans. Software Eng.*, vol. SE-15, pp. 496–501, Apr. 1989.
[32] ANSI/IEEE Std 829-1983 (*Standard for Software Test Documentation*) and ANSI/IEEE Std 1008-1987 (*Standard for Software Unit Testing*), in *Software Engineering Standards*, 3rd ed. New York: IEEE, 1989.



Roy S. Freedman obtained the Ph.D. degree from the Polytechnic University, Brooklyn, NY.

He was an Associate Professor of Computer Science at Polytechnic University, Brooklyn, NY, where his research areas included probabilistic reasoning, knowledge-based software testing, and induction. He worked for 6 years at the Hazeltine Corporation Research Laboratories, where he was responsible for the design, management, and software engineering of several knowledge-based systems for information fusion, process planning, and intelligent computer-aided instruction. He is the author of over 25 technical articles, as well as 2 books, *Programming Concepts with the Ada Language* and *Programming with APSE Software Tools*. He is also an Associate Editor of *IEEE Expert*. His consulting firm has been engaged by Hazeltine, the Eaton Corporation, the Grumman Aerospace Corportion, Chemical Bank, Equitable Life Assurance, and the New York Stock Exchange. He is now President of Inductive Solutions, Inc., a firm that builds knowledge-based systems for decision support applications.