Evaluating Specifications to Enhance Program Testing

Debra J. Richardson
Lori A. Clarke

COINS Technical Report 80-01
January 1980

Department of Computer and Information Science
University of Massachusetts, Amherst
Amherst, Massachusetts 01003

This work was supported by the National Science Foundation under grant # NSFMCS 77-02101 and the Air Force Office of Scientific Research under grant # AFOSR 77-3287.

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: University of Massachusetts COINS Technical Report TR80-01
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): Evaluating Specifications to Enhance Program Testing

5. TYPE OF REPORT & PERIOD COVERED: Final

6. PERFORMING ORG. REPORT NUMBER:

7. AUTHOR(s): Debra J. Richardson, Lori A. Clarke

8. CONTRACT OR GRANT NUMBER(s): AFOSR 77-3287, NSFMCS 77-02101

9. PERFORMING ORGANIZATION NAME AND ADDRESS: Computer and Information Science Department, University of Massachusetts, Amherst, Massachusetts 01003

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:

11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office of Scientific Research, Washington, D. C.

12. REPORT DATE: January 1980

13. NUMBER OF PAGES: 29

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):

15. SECURITY CLASS. (of this report): Unclassified

15a. DECLASSIFICATION DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report): Distribution of this document is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report):

18. SUPPLEMENTARY NOTES:

19. KEY WORDS (Continue on reverse side if necessary and identify by block number): program testing, specifications, symbolic evaluation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

An enhancement to program testing by evaluating both a specification and an implementation is described. This evaluation is used to partition the set of input values into subdomains and to analyze the associated computations. Three properties of consistency between a specification and an implementation, which are based on this partition, are proposed. An approach to demonstrating each of the three consistency properties is outlined, and the proposed method is illustrated. A testing strategy based on both the

20.

implementation and the specification is presented. The feasibility of automating this method is discussed. By demonstrating the consistency between a specification and an implementation, and ascertaining the consistency through actual execution, assurance in the reliability of the implementation is established.

ABSTRACT

An enhancement to program testing by evaluating both a specification and an implementation is described. This evaluation is used to partition the set of input values into subdomains and to analyze the associated computations. Three properties of consistency between a specification and an implementation, which are based on this partition, are proposed. An approach to demonstrating each of the three consistency properties is outlined, and the proposed method is illustrated. A testing strategy based on both the implementation and the specification is presented. The feasibility of automating this method is discussed. By demonstrating the consistency between a specification and an implementation, and ascertaining the consistency through actual execution, assurance in the reliability of the implementation is established.

1. INTRODUCTION

To date, tools to assist in program testing have focused on structural testing. These tools attempt to achieve some level of program coverage based on the program's structure. Typical criteria for coverage are executing all statements, executing all branches, and executing all paths. Some of the structural testing tools that have been developed include dynamic testing, symbolic execution, and program mutation systems. Dynamic testing systems [BAL69, FAI75, HUA78, RAM75, GAN78, STU73] monitor program execution and provide an execution history profile, which may assist the user in selecting data to achieve the desired coverage. Symbolic execution systems [BIC79, BOY75, CLA76, HOW77, HUA75, KIN76, MIL75, RAM76, VOG78] represent the domain and computation of a program path by algebraic constraints and expressions. These representations can be examined to assist in the selection of test data that will cause execution of this path. Some systems [BIC79, BOY75, CLA78, RAM76] automatically generate test data for the selected paths and some [BIC79, CLA78] automatically select paths to satisfy a specified program coverage criterion. Program mutation tools [HAM77, DEM78] systematically introduce errors into a program and then determine if the execution of the mutated program on user-provided test data produces erroneous output. By seeding errors into each branch and statement, these systems determine whether the statement and branch coverage criteria are satisfied by the provided test data.

A major drawback to structural testing is that it ignores the program's specification. By basing analysis solely on the program's structure, aspects of the program's specification that were neglected in the implementation may not be detected. Furthermore, the use of program specifications as a guide in the selection of test data is not exploited. Goodenough and Gerhart

[GOO76] have demonstated the value of employing specifications in the test data selection process. Howden [HOW79a] has studied the effectiveness of several program validation techniques and proposed a functional testing method [HOW79b], which requires that both the program specification and implementation be examined. Weyuker and Ostrand [WEY80] have proposed a method that partitions the set of input data into subdomains by analyzing both the specification and the implementation. Error detection is then enhanced by wisely selecting test data from each subdomain.

We propose to explore methods for automatic evaluation of both the program specification and implementation. This evaluation will be used to partition the set of input values into subdomains, to analyze the associated computations, and to astutely select data to test the program. In this paper, we examine a method for determining and analyzing these subdomains; we discuss several of the problems we have encountered and the feasibility of automating this approach. To facilitate the presentation, we assume that the given specification is correct; thus we are considering the correctness of an implementation with respect to this specification. The next section describes the form of program specifications we will consider here. We present a common representation for program specifications and implementations and define the subdomains into which the set of input data is partitioned. The third section defines consistency properties between the program specification and implementation, which are based on this partition. The fourth section first outlines a technique for finding the subdomains and then considers methods for demonstrating whether the consistency properties hold. The fifth section proposes a testing strategy, which selects test data from each subdomain. In the conclusion, several areas of future research are discussed.

2. PROBLEM REPRESENTATION

During the development of a program, the problem to be solved is represented by successively more elaborate descriptions of the proposed program. The initial representation of the problem is a requirements document, an informal description of the desired program behavior. From this document, a program specification is developed as a solution to the problem. The specification is refined until finally resulting in an implementation, a representation in a programming language. The program specification and program implementation are intended to be representations of the same problem at different levels of abstraction. We intend to develop an evaluation technique to exploit the similarities between the specification and implementation. To allow this evaluation, a common representation for the specification and implementation is desired.

A program specification identifies the procedures and their interactions. A procedure specification describes a procedure's intended behavior. To enable meaningful analysis, the function of the procedure must be described in a formal specification language, one whose syntax and semantics are precisely defined. There are basically two techniques for formal specification - *input-output specification* and *operational specification* [LIS79]. By either technique, the specification must be complete and unambiguous in order to correctly describe the intended function as a mapping from the inputs to the outputs. Thus, one and only one output value is specified for each input value.

By the technique of input-output specification, the relationships between the input values and the output values are described by pairs of assertions. Whenever the input values satisfy an initial assertion, a correct implementation of the procedure will compute output values satisfying

the corresponding final assertion. The technique of input-output specification has been used extensively in the inductive assertion method of program verification [LON75].

An operational specification differs from an input-output specification in that transformations on the input values are described explicitly. Operational specifications may take on several forms such as input domains with corresponding output computations, decision tables, procedure designs, and correct implementations. For the analysis method presented here, the operational type of specification is considered because it provides more information. Many of the ideas, however, are applicable to input-output specifications as well. The applicability of this method to decision tables, which are a well-defined and restricted form of operational specification, has been examined [RIC79].

An example of an operational specification is given in figure 1. This specification describes the procedure TRAP, which computes the area between a curve and the x-axis by the trapezoidal method. A procedure implementation of TRAP appears in figure 2. This procedure will be used throughout this paper to demonstrate the analysis method.

An operational specification defines the intended function of a procedure. This function is usually composed of partial functions, where each partial function is defined over a subset of the problem domain. An operational specification S, therefore, can be represented as a set of *subspecifications* $\{S_1, S_2, \ldots, S_M \mid 1 \leq M \leq \infty\}$.* For each subspecification

*The general form of an operational specification must allow an infinite number of subspecifications since some specification languages allow a notation for indefinite repetition. In addition, any subspecification may define a class of related partial functions that differ only by the number of repetitions of some transformations.

Description

TRAP computes the AREA between a curve F and the x-axis from x = A to x = B by the trapezoidal rule using N intervals of size (B-A)/N.

Interface

```
TRAP(function F(X: real): real;
    A: real; B: real; N: integer;
    AREA: real)
input F, A, B, N
output AREA
```

Operation

```
TRAP(F, A, B, N, AREA) ≡
    if N<1 then
       AREA = -1.0
       error("invalid input")
    else
       AREA = Σ_{i=1}^{N} ((F(A+(i-1)*H) + F(A+i*H)) / 2) * abs(H)
    endif
```

Abbreviations

```
H: real = (B-A) / N
```

Figure 1.

Operational Specification of TRAP

```
procedure TRAP(function F(X: real): real;
               A: real; B: real; N: integer;
               var AREA: real);

    (*TRAP computes the AREA between a curve F and the x-axis from x = A
    to x = B by the trapezoidal rule using N intervals of size (B-A)/N.*)

    var H, X: real;
    begin
    if N<1 then
        begin
        AREA := -1.0;
        write("invalid input");
        end
    else
        if A=B then
           AREA := 0.0
        else
           begin
           H := (B-A) / N;
           X := A;
           AREA := F(X) / 2;
           while X<B do
              begin
              X := X + H;
              AREA := AREA + F(X);
              end;
           AREA := (AREA - F(X) / 2) * H;
           if A>B then
              AREA := - AREA;
           end;
    end;
```

Figure 2.

Procedure TRAP

S_I, the *subspecification domain* $D[S_I]$ is the set of input data for which the subspecification is applicable and the *subspecification computation* $C[S_I]$ is the computation specified for those input values. The *specification domain* $D[S]$ is the union of the subspecification domains, $D[S]=\bigcup_{I=1}^{M} D[S_I]$.

Similarly, the implementation of a procedure defines a function, which again may be composed of partial functions. Each partial function corresponds to a *path* through the procedure. Thus, a procedure implementation P defines a set of paths $\{P_1, P_2, \ldots, P_N \mid 1 \leq N \leq \infty\}$.* Associated with each path P_J is the *path domain* $D[P_J]$, which is the set of input data that causes execution of the path, and the *path computation* $C[P_J]$, which is the function that is computed by the sequence of executable statements along the path. The *implementation domain* $D[P]$ is the union of the path domains, $D[P] = \bigcup_{J=1}^{N} D[P_J]$.

The specification is a more abstract representation of the procedure than the implementation, hence the subspecification domains and the path domains may partition the set of input data differently. In fact, the specification domain and implementation domain may not even be the same, but this implies an error in either the specification or implementation domain or both. Since we assume the specification correctly represents the procedure, the specification domain must be correct. In this paper, we partition the set of input data into subdomains so that a single subspecification and a single path are applicable to all input data in each subdomain. A *procedure subdomain* D_{IJ} is the set of input data for which the subspecification S_I and the

*Likewise, there may be an infinite number of paths due to program loops and any path may in fact define a class of related paths through the implementation that differ only by the number of iterations of some loops.

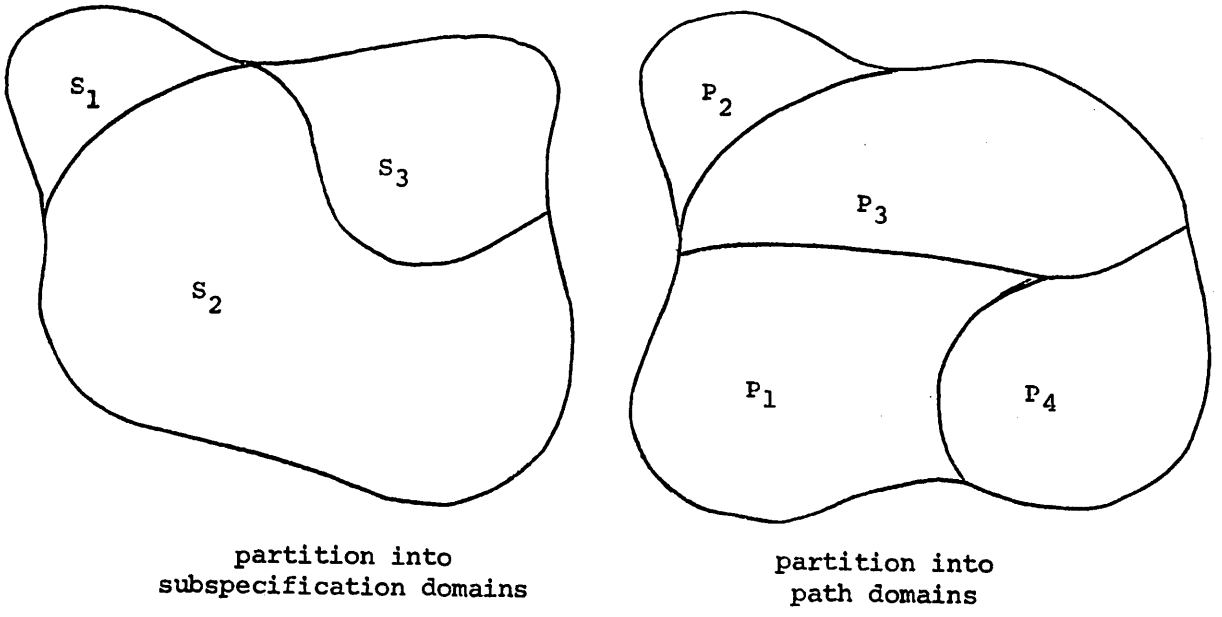
path P_J are applicable - that is, $D_{IJ} = D[S_I] \cap D[P_J]$. In addition, for each subspecification S_I there may be input data in its domain $D[S_I]$ that are not treated by any path; this set, $D_{I0} = D[S_I] - \bigcup_{J=1}^{N} D_{IJ}$, is a procedure subdomain. Also, for each path P_J, there may be input data in its domain $D[P_J]$ that are not treated by any subspecification; this set, $D_{0J} = D[P_J] - \bigcup_{I=1}^{M} D_{IJ}$, is a procedure subdomain as well. Figure 3 shows an example of the procedure subdomains that result from the partitions of the specification and implementation domains. The procedure subdomains will be used in determining consistency between the specification and implementation and in selecting data to test the implementation.

3. CONSISTENCY BETWEEN PROCEDURE SPECIFICATIONS AND IMPLEMENTATIONS

A procedure specification and implementation can be compared to determine if the implementation conforms to the specification. With this comparison in mind, three consistency properties are introduced - compatibility, equivalence, and isomorphism - which differ in the manner in which the implementation conforms to the specification. An approach to demonstrating whether these consistency properties hold between a specification and an implementation is outlined in the next section.

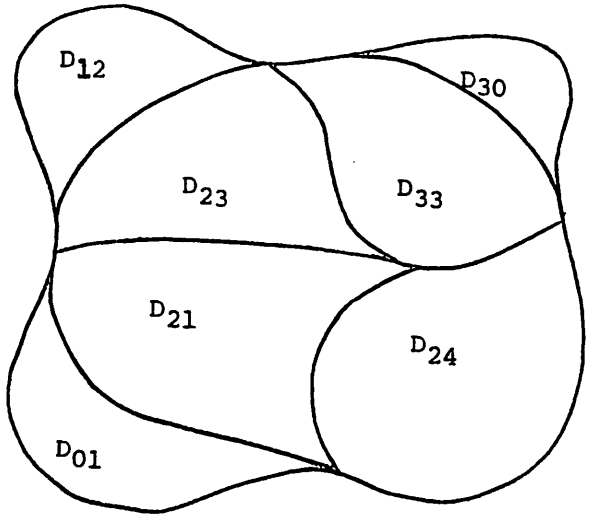
A fundamental form of consistency is the compatibility of a specification and an implementation. Compatibility states that the implementation and the specification have the same interface - that is, they have the same number and type of inputs and outputs - and the inputs are restricted to values from the same domain.

Definition: An implementation P is *compatible* with a specification S if P and S have the same input vector x, the same output vector z, and are defined for the same domain, $D[S] = D[P]$.



partition into
subspecification domains

partition into
path domains



partition into
procedure subdomains

Figure 3.

Development of Procdure Subdomains

In the trivial case, the domain for a particular input is the entire set of values for the type of the input, but some specification and programming languages allow assumptions that further restrict the domain of input values. Assuming that all inputs and outputs are logically independent, compatibility must hold for the implementation to be a correct representation of the specification. Note that $D[S] = D[P]$ implies that all input data in each subspecification domain are treated by some path and all input data in each path domain are treated by some subspecification. Hence, all D_{IG} and D_{OJ} procedure subdomains are empty. The other properties of consistency are defined under the assumption that compatibility holds.

The prevalence of compatibility does not imply that the implementation is correct with respect to the specification. To realize the function described by the specification, the implementation must not only have the same interface, it must compute the output values specified for each input vector in the domain. An implementation P is equivalent to a specification S if for all $x \epsilon D[S]$, $P(x) = S(x)$, where $P(x)$ is the output vector resulting from execution of P on x, and $S(x)$ is the output vector specified by S for x. Equivalence between a procedure implementation and a specification implies the implementation is correct with respect to the specification.

This property of equivalence can be stated in terms of the relationships between the subspecifications and paths over the procedure subdomains. For any input vector x, a particular path, say P_J, is executed; thus $x \epsilon D[P_J]$, and $P(x) = C[P_J](x)$. Similarly, a particular subspecification, say S_I, is applicable; thus $x \epsilon D[S_I]$, and $S(x) = C[S_I](x)$. For this input vector, the specification and the implementation produce the same output values,

$S(x) = P(x)$, if and only if the appropriate subspecification and path computations agree, $C[S_I](x) = C[P_J](x)$. If a subspecification and a path compute equal output values for all input data to which they both apply, their computations are equal over that set of input data, which is the associated procedure subdomain.

Definition: A subspecification computation $C[S_I]$ and a path computation $C[P_J]$ are equal over the associated procedure subdomain, $D_{IJ} = D[S_I] \cap D[P_J]$, if for all $x \epsilon D_{IJ}$, $C[S_I](x) = C[P_J](x)$. This is denoted by $C[S_I] \underset{D_{IJ}}{=} C[P_J]$.

This definition gives rise to a definition of the equivalence of an implementation and a specification, which is in terms of the equality of the computations over procedure subdomains.

Definition: An implementation P is *equivalent* to a specification if for all procedure subdomains D_{IJ}, $1 \leq I \leq M$ and $1 \leq J \leq N$, $C[S_I] \underset{D_{IJ}}{=} C[P_J]$.

Equivalence is sometimes very difficult, if not impossible, to determine. A restricted form of equivalence between a specification and an implementation is isomorphism, which is often an easier property to determine. When isomorphism holds each subspecification is associated with an equivalent path, which is unique. A subspecification and a path are equivalent if their domains are equal and their computations are equal over that domain.

Definition: A subspecification S_I and a path P_J are equivalent if $D[S_I] = D[P_J]$ and for all $x \epsilon D[S_I]$, $C[S_I](x) = C[P_J](x)$. This is denoted by $S_I \equiv P_J$.

Note that $S_I \equiv P_J$ implies $C[S_I] \underset{D_{IJ}}{=} C[P_J]$ since $D_{IJ} = D[S_I] = D[P_J]$. This relationship between subspecifications and paths gives rise to a definition of an isomorphism between a specification and an implementation of a procedure.

Definition: An implementation P is *isomorphic* to a specification S if there exists a bijective mapping B: $S \rightarrow P$ such that $B(S_I) = P_J$ if and only if $S_I \equiv P_J$.

If the specification correctly describes the desired procedure, isomorphism is sufficient, but not necessary, for the implementation to be correct. In addition, isomorphism gives evidence that the internal structure of an implementation and a specification are similar.

Equivalence is certainly the most important of the proposed consistency properties. Under the assumptions that compatibility holds and the specification is correct, the implementation is correct if and only if it is equivalent to the associated specification. Isomorphism might be required, however, when a specification is a detailed design that is to be used as a guideline for implementation of the procedure. On the other hand, isomorphism might not be desired when a specification is written for simplicity, but an implementation ought to be coded for efficiency.

The three properties of consistency allow the attachment of differing requisites on the conformity of an implementation to a specification. Compatibility implies that the implementation conforms to the specified interface. The prevalence of either isomorphism or equivalence implies that the implementation is correct, provided the specification precisely describes the intended function. Moreover, isomorphism implies that the implementation realizes the function in much the same manner as the specification describes that function.

4. DEMONSTRATION OF CONSISTENCY

The definitions of consistency between a specification and an implementation suggest methods for determining whether these properties prevail. Demonstration of compatibility is only briefly considered since it is a better understood process. One approach for demonstrating equivalence and isomorphism between a specification and an implementation, which employs symbolic evaluation to represent the domains and computations, is outlined in this section. A more detailed explanation of this approach is given in [RIC78b]. Demonstration of consistency is illustrated for the specification and implementation of procedure TRAP. Several problems with this approach are described and some solutions and areas of future investigation are proposed.

The process of determining compatibility is similar to determining the uniformity of procedure interfaces in an implementation [GAN78, RAM75, OST76]. The compatibility between levels of design has also been explored [CAI75, ROB77, TEI77]. Determining compatibility between an implementation and a specification is facilitated if the specification and the programming languages have similar constructs for declaring parameters and global variables. By comparing such declarations, it is possible to determine if the implementation and the specification have the same number and type of parameters and global variables. If the languages do not support explicit declarations on how these variables are used, then data flow analysis methods [OST76] may be utilized to determine whether each such variable has the same input and output class in the implementation as its class in the specification. In addition, the specification and programming languages might allow explicit assumptions on the inputs or implicit assumptions in the input statements [ABR79] constraining the values the inputs may assume. These assumptions must also be compared

to determine the equivalence of the domain of input values. If the input vector, output vector, and the domain of the implementation agree with those of the specification, then the implementation is compatible to the specification.

Once compatibility is established, the demonstration of additional consistency can continue with a comparison of the subspecification domains with the path domains and the subspecification computations with the path computations. Symbolic evaluation can be used to create symbolic representations of these domains and computations.

Symbolic evaluation [CLA76, DAR78] of an implementation assigns symbolic names to the input values and "executes" a path. A path domain is represented by a system of constraints on the symbolic names for the input values. The condition formed by these constraints defines the subset of the domain for which the path is executed. This system of constraints can be translated into some canonical form, such as a simplified, conjunctive normal form [DEU73]. A path computation is represented by a vector of symbolic formulas for the output values. Each formula is a symbolic expression in terms of the symbolic names assigned to the input values. These expressions may be converted to a canonical form, such as a simplified, ordered expression [RIC78a]. A similar procedure can be used to "apply" a subspecification to symbolic inputs, and thus construct representations of the subspecification domain and the subspecification computation. Symbolic evaluation of an implementation can be extended [CHE79, CLA80] to a class of paths that differ only by the number of iterations of loops on the paths by a technique that tries to represent each loop by a closed form expression. Figure 4 shows a closed form representation for the WHILE loop in the implementation of procedure TRAP. Likewise, symbolic evaluation of a specification

$$k_e = \text{minimum } \{k \mid (k \geq 1) \wedge (X_1 + (k-1)*H \geq B)\}$$

$$X = X_1 + (k_e - 1)*H$$

$$AREA = AREA_1 + \sum_{i=1}^{k_e - 1} F(X_1 + i*H)$$

Note: X_1 and $AREA_1$ are the values of X and AREA, respectively, at entry to the first iteration of the loop. The loop is iterated $k_e - 1$ times.

Figure 4.

Closed Form Representation
of WHILE Loop in
the Implementation of Procedure TRAP

can be extended. On the other hand, a subspecification may represent the repetition of a transformation by a closed form expression, using, among other things, summation or product notation or a set of recurrence relations. The subspecification domains and computations, which were derived by symbolic evaluation of the specification of procedure TRAP, are given in figure 5. Figure 6 provides the path domains and computations derived by symbolic evaluation of the implementation of procedure TRAP.

Since both the specification and the implementation are unambiguous, the subspecification domains are mutually disjoint as are the path domains. No such restriction, however, has been made on the computations; neither the subspecification computations nor the path computations must be distinct. For example, two paths might perform the same computation for their respective domain. With this in mind, the comparison of a specification and an implementation is driven by the relationships between the subspecification domains and the path domains. After the correspondence between a subspecification domain and path domain has been determined, their associated computations are compared. There are three problems inherent in demonstrating either equivalence or isomorphism: showing that two domains are equal, showing that the intersection of two domains is empty, and showing that two computations are equal over a domain. These problems are treated after the approach to demonstrating isomorphism and equivalence is presented.

The property of isomorphism holds between a specification and an implementation if there is a one-to-one correspondence between the subspecifications and paths, such that each subspecification corresponds to an equivalent path. A correspondence between a subspecification S_I and a path P_J is made if the subspecification domain $D[S_I]$ and the path domain $D[P_J]$ are equal. Demonstration of this equality can sometimes be achieved by a term-by-term

$D[S_1] = \{(a,b,n) \mid (n<1)\}$
$C[S_1] = \text{AREA: } -1.0 \text{ "invalid input"}$

$D[S_2] = \{(a,b,n) \mid (n \geq 1) \wedge (a \leq b)\}$
$C[S_2] = \text{AREA: } (-a*F(a) - a*F(b) + b*F(a) + b*F(b) - 2*a*\sum_{i=1}^{n-1} F(((n-i)*a+i*b) / n) + 2*b*\sum_{i=1}^{n-1} F(((n-i)*a+i*b) / n)) / 2*n$

$D[S_3] = \{(a,b,n) \mid (n \geq 1) \wedge (a > b)\}$
$C[S_3] = \text{AREA: } (a*F(a) + a*F(b) - b*F(a) - b*F(b) + 2*a*\sum_{i=1}^{n-1} F(((n-i)*a+i*b) / n) - 2*b*\sum_{i=1}^{n-1} F(((n-i)*a+i*b) / n)) / 2*n$

Figure 5.

Subspecification Domains and Computations
for the Specification of Procedure TRAP

$D[P_1] = \{(a,b,n) \mid (n<1)\}$
$C[P_1] = \text{AREA: } -1.0 \text{ "invalid input"}$

$D[P_2] = \{(a,b,n) \mid (n \geq 1) \wedge (a=b)\}$
$C[P_2] = \text{AREA: } 0.0$

$D[P_3] = \{(a,b,n) \mid (n \geq 1) \wedge (a > b)\}$
$C[P_3] = \text{AREA: } (a*F(a) + a*F(b) - b*F(a) - b*F(b) + 2*a*\sum_{i=1}^{n-1} F(((n-i)*a+i*b) / n) - 2*b*\sum_{i=1}^{n-1} F(((n-i)*a+i*b) / n)) / 2*n$

$D[P_4] = \{(a,b,n) \mid (n \geq 1) \wedge (a < b)\}$
$C[P_4] = \text{AREA: } (-a*F(a) - a*F(b) + b*F(a) + b*F(b) - 2*a*\sum_{i=1}^{n-1} F(((n-i)*a+i*b) / n) + 2*b*\sum_{i=1}^{n-1} F(((n-i)*a+i*b) / n)) / 2*n$

Figure 6.

Path Domains and Computations
for the Implementation of Procedure TRAP

Note: a, b, and n are the symbolic names assigned to parameters A, B, and N.

comparison of the constraints in the symbolic representations of the two domains. After determining domain equality, the equality of the subspecification computation $C[S_I]$ and the path computation $C[P_J]$ over this domain must be determined. Again a term-by-term comparison of the symbolic representations of the two computations may sometimes determine their equality. If the equality of computations is determined, then $S_I \equiv P_J$. If a one-to-one correspondence can be constructed in this way, then the implementation is isomorphic to the specification.

An implementation is equivalent to a specification if for each procedure subdomain, the associated subspecification computation and path computation are equal over that subset of input data. For each subspecification S_I and path P_J, the procedure subdomain $D_{IJ} = D[S_I] \cap D[P_J]$ can be constructed by conjoining the representations of the subspecification domain and path domain. If this subdomain is empty, then no input data exists that causes execution of the path and for which the subspecification is applicable, thus, the computations are trivially equivalent. For a non-empty procedure subdomain, the corresponding subspecification and path computations must be compared for equality over this subdomain, $C[S_I] \underset{D_{IJ}}{=} C[P_J]$. If the computations are equal over each procedure subdomain, then the implementation is equivalent to the specification.

The methods for determining equivalence and isomorphism can easily be combined. Even when isomorphism does not hold, it is beneficial to first determine the equivalent subspecifications and paths since these need not be considered further. The computations for the remaining subspecifications and paths can then be considered for equality over their associated procedure subdomains. Thus, the method for determining isomorphism is applied, and then the method for determining equivalence is applied for the remaining

subspecifications and paths. Figure 7 illustrates the comparison of the specification and implementation for procedure TRAP. Note that S_1 is equivalent to P_1 and S_3 is equivalent to P_3. The remaining subspecifications and path computations are equal over their associated procedure subdomains. Hence, the implementation of the procedure TRAP is equivalent to the specification, although not isomorphic. For subdomain D_{22}, the equality of the computations requires more than a symbolic comparison, methods for handling this are discussed below.

Although isomorphism and equivalence can often be demonstrated simply by comparing the symbolic representation of the domains and computations, this is not always the case. Here we present some additional techniques that can sometimes be applied. When a subspecification domain $D[S_K]$ and a path domain $D[P_L]$ cannot be shown equal by a symbolic comparison, this equality can be demonstrated by showing that $D[S_K] \cap \sim D[P_L]$ is empty (where $\sim D[P_L]$ is the complement of $D[P_L]$). There is a problem, however, in showing that the intersection of two domains is empty. This occurs not only in this case, but also when determining that the intersection of a subspecification domain $D[S_I]$ and a path domain $D[P_J]$ is empty - that is, whether the procedure subdomain D_{IJ} is empty. (Note that if a subspecification domain and a path domain are not equal, then the intersection of these domains must be determined; if this intersection is empty the associated computations need not be compared.) An approach to this problem, determining the emptiness of the intersection of two domains, is the axiomatic approach, which uses first order predicate calculus to prove whether or not the conjunction defining the intersection is satisfiable. This method is subject to the limitations of automatic theorem proving [ELS72]. Another approach is the algebraic approach, which attempts to find a solution to the

$D_{11} = D[S_1] = D[P_1] = \{(a,b,n) \mid (n<1)\}$

$C[S_1] = C[P_1] =$ AREA: -1.0 "invalid input" $\Rightarrow \quad S_1 \equiv P_1$

$D_{33} = D[S_3] = D[P_3] = \{(a,b,n) \mid (n \geq 1) \wedge (a>b)\}$

$$C[S_3] = C[P_3] = \text{AREA: } (a*F(a) + a*F(b) - b*F(a) - b*F(b) + 2*a*\sum_{i=1}^{n-1} F(((n-i)*a+i*b) / n) - 2*b*\sum_{i=1}^{n-1} F(((n-i)*a+i*b) / n))/ 2*n \Rightarrow \quad S_3 \equiv P_3$$

$D_{22} = D[S_2] \cap D[P_2] = \{(a,b,n) \mid (n \geq 1) \wedge (a \leq b) \wedge (a=b)\}$

$= \{(a,b,n) \mid (n \geq 1) \wedge (a=b)\}$

$$C[S_2] - C[P_2] = \text{AREA: } (-a*F(a) - a*F(b) + b*F(a) + b*F(b) - 2*a*\sum_{i=1}^{n-1} F(((n-i)*a+i*b) / n) + 2*b*\sum_{i=1}^{n-1} F(((n-i)*a+i*b) / n))/ 2*n - 0.0$$

$$= (b-a) * (F(a) + F(b) + \sum_{i=1}^{n-1} F(((n-i)*a+i*b) / n)) / 2*n$$

solution set: $(a = b) \vee (F(a) + F(b) + \sum_{i=1}^{n-1} F(((n-i)*a+i*b) / n) = 0.0) \Rightarrow \quad C[S_2] \underset{D_{22}}{=} C[P_2]$

$D_{24} = D[S_2] \cap D[P_4] = \{(a,b,n) \mid (n \geq 1) \wedge (a \leq b) \wedge (a<b)\}$

$= \{(a,b,n) \mid (n \geq 1) \wedge (a<b)\}$

$$C[S_2] = C[P_4] = \text{AREA: } (-a*F(a) - a*F(b) + b*F(a) + b*F(b) - 2*a*\sum_{i=1}^{n-1} F(((n-i)*a+i*b) / n) + 2*b*\sum_{i=1}^{n-1} F(((n-i)*a+i*b) / n)) / 2*n \Rightarrow \quad C[S_2] \underset{D_{24}}{=} C[P_4]$$

Figure 7.

Demonstration of Equivalence of the
Specification and Implementation of Procedure TRAP

constraints defining the intersection. If the set of constraints is unsatisfiable, the intersection is empty. Several algebraic techniques, such as a linear programming or a gradient hill climbing algorithm, can be used to solve the system of constraints. No method, however, can solve any arbitrary system of constraints [DAV73], and, therefore, determining if the intersection is empty is undecidable.

Another problem that must be addressed is the determination of computation equality over a procedure subdomain. Often, a term-by-term comparison of the symbolic representations of the subspecification computation $C[S_I]$ and the path computation $C[P_J]$ reveals that the two computations are symbolically identical, thus, they are equal over any domain. Otherwise, the two computations are equal over the associated procedure subdomain D_{IJ} if the symbolic difference between their symbolic representations, $C[S_I] - C[P_J]$, equals zero for all elements of that domain. The most straight-forward method for determining whether this holds is by finding the solution set of the equation $C[S_I] - C[P_J] = 0$. This set can be represented symbolically by a disjunct of the factors of this equation, as was done for procedure subdomain D_{22} in figure 7, or when the solution set is discrete, by using a mathematical package for finding the zeroes of a function. If the condition defining the procedure subdomain restricts the inputs to values in this solution set, then the symbolic difference equals zero over that domain. Several other approaches to deciding computation equality are proposed in [RIC78b], although, in general, this is undecidable. When the equality of the subspecification and path computations over the associated procedure subdomain cannot be decided, testing can provide some assurance of their equality. A testing strategy is discussed in the next section.

5. A TESTING STRATEGY

Demonstrating equivalence of an implementation to the associated specification verifies that the implementation performs the intended task. This method of attesting to program reliability, however, divorces itself from the run-time surroundings by showing consistency in a postulated environment, just as proof of correctness is done in an artificial environment. To remedy this, the demonstration of consistency must be complemented by the actual execution of the implementation. In addition, when consistency cannot be shown, testing can often demonstrate the equality of computations or provide counter examples. The method of determining equivalence can be extended to support a testing strategy.

In demonstrating equivalence, the set of input data is partitioned into subsets of input data that are treated by a single subspecification S_I and by a single path P_J; this subset is the procedure subdomain D_{IJ}. Thus, each element of a procedure subdomain should be treated consistently by the specification and the implementation. Wisely selecting test data from each subdomain can check the consistent operation of the specification and implementation.

A test data set can be constructed by selecting one or more input values from each procedure subdomain. The appropriate selection of input values from each subdomain can increase the probability of detecting errors. White and Cohen [WHI80] have proposed a strategy for selecting test data to verify the boundaries of a path domain. This same technique can be carried over to procedure subdomains. Computations tend to be sensitive to various data points. By examining the representations of the subspecification and path computations provided by symbolic evaluation, computationally sensitive data that lies within the procedure subdomain can be selected. In addition,

typical data to exercise the subdomain should be chosen. Some work has been done on determining the appropriate number of data points that should be selected for polynomial computations [DEM77, HOW76b]. Taking these testing strategies into account, the set of test data that might be generated for procedure TRAP is shown in figure 8.

This test data selection strategy is similar to that used in path analysis approaches [CLA76, CLA78, HOW76a], in which a test set is constructed by choosing an element from each path domain. Path analysis testing strategies, however, are based solely on the structure of the implementation. The test data selection approach presented here is based on both the implementation and the specification, and thus takes into consideration what the implementation is supposed to do in addition to what it actually does. Weyuker and Ostrand [WEY80] have also suggested that procedure subdomains be determined and carefully tested. Testing methods that integrate information from the implementation and the specification have been shown to be more effective than methods based on a single source of information [HOW79a].

6. CONCLUSION

The evaluation of specifications to analyze implementations and to assist in the program testing process can greatly enhance the reliability of software. When a specification is available, an implementation can be compared to the specification for consistency. This comparison can be used to prove that the implementation conforms to the specification. We have proposed consistency properties that differ in how closely the implementation conforms to the specification. We have investigated a method, which utilizes symbolic evaluation, to determine whether these properties hold. This method relies on the development of procedure subdomains, which partition

D_{11}	(A: 0.0, B: 0.0, N: - 1)
D_{33}	(A: 0.00001, B: 0.0, N: 1) (A: 0.0, B: -0.00001, N: 2) (A: 10.0, B: 1.0, N: 100) (A: 1.0, B: -10.0, N: 1000)
D_{22}	(A: 0.0, B: 0.0, N: 1) (A: 10.0, B: 10.0, N: 10)
D_{24}	(A: 0.0, B: 0.00001, N: 1) (A: -0.00001, B: 0.0, N: 2) (A: 1.0, B: 10.0, N: 100) (A: -10.0, B: 1.0, N: 1000)

Figure 8.

Test Data Selected for Procedure TRAP

the set of input data based on the implementation and the specification. By examining these subdomains, a more comprehensive set of test data can be generated than one obtained by analyzing the implementation or the specification alone. In light of the work on symbolic evaluation, we believe the method we have proposed could be at least partially automated.

There are several problems that require additional investigation. Strategies for generating test data from the representation of the procedure subdomains have been proposed in this paper, but need to be evaluated further. This paper discussed approaches for dealing with the problems that arise in determining consistency - equality of two domains, emptiness of the intersection of two domains, and equality of two computations over a domain. Additional approaches to these problems must be developed. The proposed evaluation method assumes that loops can be represented in a closed form. While this is often the case, methods for analyzing loops must be further refined. Although symbolic evaluation of programs has been extensively researched, symbolic evaluation of specifications has yet to be seriously considered.

There are several established programming languages, but the design of specification languages is still in its infancy. If program specifications are to contribute effectively to the analysis of programs, more applicable specification languages must be designed. Formal techniques for specifying the intended function of a program can provide a concise and well-understood description, which should reduce the difficulty of symbolically evaluating specifications. The evaluation method presented in this paper assumes that a procedure specification is complete. The evaluation of higher level specifications, which might be incomplete, should also be considered. While strong consistency, such as equivalence, could not be proven with an incomplete

specification, weaker forms of consistency could be demonstrated or inconsistencies could be detected. As progress in the design of formal specification languages is made, the feasibility of a mechanical means of using new types of specification in program analysis must be evaluated.

The approach presented in this paper is concerned with the analysis of an implementation in relation to a specification for the intended procedure. The specifications considered are detailed and might correspond to procedure descriptions developed late in the design of a program. If analysis is not performed throughout the development process, there is no assurance that the specifications indeed capture the desired behavior of the procedures. This problem is addressed by current work [SRS79] in the development of tools that support the design and analysis of program descriptions during the early stages of development. To achieve the goal of producing more reliable software, a complimentary set of software tools for program specification, program design, program verification, and program testing must be integrated.

REFERENCES

ABR79 P. Abrahams and L. A. Clarke, "Compile-Time Analysis of Data List - Format List Correspondences," IEEE Trans. of Software Engineering, SE-5, 6, November 1979, 612-617.

BAL69 R. M. Balzer, "EXDAMS--Extendable Debugging and Monitoring System," 1969 Spring Joint Computer Conf., AFIPS Conf. Proc., 34, AFIPS Press, Montvale, New Jersey, 576-580.

BIC79 J. Bicevskis, J. Borzovs, U. Straujums, A. Zarins, and E. F. Miller, Jr., "SMOTL - A System to Construct Samples for Data Processing Program Debugging," IEEE Trans. on Software Engineering, SE-5, 1, January 1979, 60-66.

BOY75 R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT--A Formal System for Testing and Debugging Programs by Symbolic Execution," Proc. Int. Conf. Reliable Software, April 1975, 234-244.

CAI75 S. H. Caine and E. K. Gordon, "PDL - A Tool for Software Design," Proc. National Computer Conference, 1975.

CHE79 T. E. Cheatham, G. H. Holloway, and J. A. Townley, "Symbolic Evaluation and the Analysis of Programs," IEEE Trans. on Software Engineering, SE-5, 4, July 1979, 402-417.

CLA76 L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," IEEE Trans. on Software Engineering, SE-2, 3, September 1977, 215-222.

CLA78 L. A. Clarke, "Automatic Test Data Selection Techniques," Infotech State of the Art Report on Software Testing, 2, September 1978, 43-64.

CLA80 L. A. Clarke and D. J. Richardson, "Symbolic Evaluation Methods for Program Analysis," to appear in Program Flow Analysis: Theory and Applications, Prentice Hall, Inc., Englewood Cliffs, NJ, 1980.

DAR78 J. A. Darringer and J. C. King, "Applications of Symbolic Execution to Program Testing," Computer, 11, 4, April 1978, 51-68.

DAV73 M. Davis, "Hilbert's Tenth Problem is Unsolvable," American Math. Mon., 80, March 1973, 233-269.

DEM77 R. A. DeMillo and R. J. Lipton, "A Probabilistic Remark on Algebraic Program Testing," School of Information and Computer Science Technical Report, Georgia Institute of Technology, May 1977.

DEM78 R. A. DeMillo, R. J. Lipton, F. G. Sayward, "Program Mutation: A New Approach to Program Testing," Infotech State of the Art Report on Software Testing, 2, September 1978, 107-128.

DEU73 L. P. Deutsch, "An Interactive Program Verifier," Ph.D. Dissertation, University of California, Berkeley, May 1973.

ELS72 B. Elspas, K. N. Levitt, R. J. Waldinger, and A. Waksman, "An Assessment of Techniques for Proving Program Correctness," ACM Computing Surveys, 4, 2, June 1972, 97-146.

FAI75 R. E. Fairley, "An Experimental Program-Testing Facility," IEEE Trans. on Software Engineering, SE-1, 4, December 1975, 350-357.

GAN78 C. Gannon, "JAVS: A JOVIAL Automated Verification System," Proc. COMPSAC '78, November 1978.

GEL78 A. Geller, "Test Data as an Aid to Proving Program Correctness," CACM, 21, 5, May 1978, 368-375.

GOO76 J. B. Goodenough and S. L. Gerhart, "Toward a Theory of Test Data Selection," IEEE Trans. on Software Engineering, SE-1, 2, September 1976, 156-173.

HAM77 R. G. Hamlet, "Testing Programs with the Aid of a Compiler," IEEE Trans. on Software Engineering, SE-3, 4, July 1977, 279-290.

HOW75 W. E. Howden, "Methodology for the Generation of Program Test Data," IEEE Trans. on Computer, C-24, 5, May 1975, 554-559.

HOW76a W. E. Howden, "Reliability of the Path Analysis Testing Strategy," IEEE Trans. on Software Engineering, SE-2, 3, September 1976, 208-215.

HOW76b W. E. Howden, "Algebraic Program Testing," Department of Applied Physics and Information Science, University of California, San Diego, TR-12, November 1976.

HOW77 W. E. Howden, "Symbolic Testing and the DISSECT Symbolic Evaluation System," IEEE Trans. on Software Engineering, SE-3, 4, July 1977, 266-278.

HOW79a W. E. Howden, "An Analysis of Software Validation Techniques for Scientific Programs," University of Victoria, Victoria, British Columbia, DM-171-IR, March 1979.

HOW79b W. E. Howden, "Functional Testing and Design Abstractions," University of Victoria, Victoria, British Columbia, DM-180-IR, May 1979.

HUA75 J. C. Huang, "An Approach to Program Testing," ACM Computing Surveys, 7, 3, September 1975, 113-128.

HUA78 J. C. Huang, "Program Instrumentation and Software Testing," Computer, 11, 4, April 1978, 25-32.

KIN76 J. C. King, "Symbolic Execution and Program Testing," CACM, 19, 7, July 1976, 385-394.

LIS79 B. H. Liskov and V. Berzins, "An Appraisal of Program Specification," in Research Directions in Software Technology, MIT Press, Cambridge, MA, 1979.

LON75 R. L. London, "A View of Program Verification," Proc. Int. Conf. Reliable Software, April 1975, 534-545.

MIL75 E. F. Miller and R. A. Melton, "Automated Generation of Test Case Datasets," Proc. Int. Conf. Reliable Software, April 1975, 51-58.

OST76 L. J. Osterweil and L. D. Fosdick, "DAVE - a Validation Error Detection and Documentation System for FORTRAN Programs," Software - Practice and Experience, 6, 1976, 473-486.

RAM75 C. V. Ramamoorthy and S. F. Ho, "Testing Large Software with Automated Software Evaluation Systems," IEEE Trans. on Software Engineering, SE-1, 1, March 1975, 16-58.

RAM76 C. V. Ramamoorthy, S. F. Ho, and W. T. Chen, "On the Automated Generation of Program Test Data," IEEE Trans. on Software Engineering, SE-2, 4, December 1976, 293-300.

RIC78a D. J. Richardson, L. A. Clarke, and D. L. Bennett, "SYMPLR, Symbolic Multivariate Polynomial Linearization and Reduction," Department of Computer and Information Science, University of Massachusetts, TR-78-16, July 1978.

RIC78b D. J. Richardson, "Theoretical Considerations in Testing Programs by Demonstrating Consistency with Specifications," Dig. Workshop on Software Testing and Test Documentation, December 1978, 19-56.

RIC79 D. J. Richardson, "Program Testing by Demonstrating Consistency with Specifications," Department of Computer and Information Science, University of Massachusetts, TR-79-02, February 1979.

ROB77 L. Robinson and O. Roubine, "SPECIAL, a Specification and Assertion Language," Stanford Research Institute International Technical Report, CSL-46, Menlo Park, CA, January 1977.

SRS79 Proceedings of the Conference on Specifications of Reliable Software, Cambridge, MA, April 1979.

STU73 L. G. Stucki, "Automatic Generation of Self-Metric Software," Rec. 1973 IEEE Symp. Software Reliability, April 1973, 94-100.

TEI77 D. Teichrow and E. A. Hershey,III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Trans. on Software Engineering, SE-3, 1, January 1977, 41-48.

VOG78 U. Voges, W. Geiger, and L. Gmeiner, " A Complementary Approach to Validate Software," Dig. Workshop on Software Testing and Test Documentation, December 1978, 170-190.

WEY80 E. J. Weyuker and T. J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains," to appear in IEEE Trans. on Software Engineering, 1980.

WHI80 L. J. White and E. I. Cohen, "A Domain Strategy for Computer Program Testing," to appear in IEEE Trans. on Software Engineering, 1980.