A PARTITION ANALYSIS METHOD TO INCREASE PROGRAM RELIABILITY

Debra J. Richardson
Lori A. Clarke

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

ABSTRACT

A major drawback of most program testing methods is that they ignore program specifications, and instead base their analysis solely on the information provided in the implementation. This paper describes the partition analysis method, which assists in program testing and verification by evaluating information from both a specification and an implementation. This method employs symbolic evaluation techniques to partition the set of input data into procedure subdomains so that the elements of each subdomain are treated uniformly by the specification and processed uniformly by the implementation. The partition divides the procedure domain into more manageable units. Information related to each subdomain is used to guide in the selection of test data and to verify consistency between the specification and the implementation. Moreover, the test data selection process, called partition analysis testing, and the verification process, called partition analysis verification, are used to enhance each other, and thus increase program reliability.
Keywords: program testing, program verification, symbolic evaluation.

INTRODUCTION

A major drawback of most program testing methods is that program specifications are ignored; test data selection is based solely on the information derived from the implementation. Such methods are unlikely to detect errors that arise when a implementation neglects aspects of the problem. Utilizing an understanding of the specification, however, may direct attention to such errors. Recently, several attempts have been made to employ sources of information over and above the implementation in selecting test data. Goodenough and Gerhart [10] have argued that the specification and the implementation are both valuable sources of information that must be used by testing methods. Thus far, however, no method has been developed that exploits formal specifications, even though they are becoming more readily available as their value in the development of reliable software is recognized.

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

We are exploring a method, called partition analysis, that assists in program testing and program verification by incorporating information from both a formal specification of the procedure and an implementation for the procedure. The partition analysis method employs symbolic evaluation techniques to partition the set of input data into procedure subdomains, so that the elements of each subdomain are treated uniformly by the specification and processed uniformly by the implementation. By forming these subdomains, the procedure domain is divided into more manageable units, as is the task of demonstrating program reliability. Information related to each procedure subdomain is used to guide in the selection of test data that reveals errors in the implementation or provides confidence in its correctness. This information is also used to verify consistency between the specification and the implementation. Moreover, the test data selection process, called partition analysis testing, and the verification process, called partition analysis verification, are used to enhance each other; the execution of some elements in the subdomain may assist in verification, while the verification process may direct the selection of test data.

In this paper, we describe the partition analysis method. To facilitate the presentation, we assume that a given procedure specification is correct; thus we are considering the correctness of an implementation with respect to this specification. The next section presents common representations for procedure specifications and implementations and defines the procedure subdomains into which the set of input data is partitioned. The third section defines consistency properties between a procedure specification and implementation, which are based on this partition. The fourth section outlines partition analysis verification, a technique for demonstrating whether the consistency properties hold. The fifth section describes partition analysis testing, a testing strategy that astutely selects test data from each procedure subdomain. In the conclusion, several areas of future research are discussed.

PROBLEM REPRESENTATION

A program specification and program implementation are intended to be representations of the same problem at different levels of abstraction. We are developing an analysis method to exploit the similarities between a specification and an implementation. This method compares the implementation of a procedure to a specification of

that procedure. The procedure specification must be written in a formal specification language. While work on applying the partition analysis method to high-level specifications is underway, thus far the method has only been applied to low-level specifications such as those developed during the late stages of design.

An example of such a procedure 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 rule. A procedure implementation of TRAP appears in Figure 2. This procedure will be used throughout this paper to demonstrate the partition analysis method.

A procedure specification and a procedure implementation both describe a function, where the function is usually composed of partial functions. Each partial function is a computation defined over a subset of the procedure domain. To facilitate the comparison of an implementation to a specification, the partition analysis method takes advantage of this similarity by translating the two descriptions into common representations, which explicitly describe the partial functions.

In a procedure implementation, a partial function is represented by a path, which is a sequence of statements through the implementation. Thus, the representation of a procedure implementation P is 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, $D[P] = \cup_{J=1,N} D[P_J]$.

In a procedure specification, each partial function is represented by a subspec. The actual form of a subspec depends on the specification language. For low-level specifications a subspec is similar to a path, although more abstract constructs may appear. The representation of a specification S, therefore,** is a set of subspecs $\{S_1, S_2, \ldots, S_M \mid 1 \leq M \leq \infty\}$. For each subspec S_I, the subspec domain $D[S_I]$ is the set of input data for which the subspec is applicable and the subspec computation $C[S_I]$ is the computation specified for those input values. The specification domain D[S] is the union of the subspec domains, $D[S] = \cup_{I=1,M} D[S_I]$.

The partition analysis method utilizes symbolic evaluation techniques [2,4,5,14,17] to provide these common representations of the specification and the implementation. In applying symbolic evaluation to an implementation, symbolic

---

*There may be an infinite number of paths due to program loops. As will be explained shortly, paths through the implementation that differ only by the number of iterations of some loops may be classified as a single path.

**The general form of a specification must allow for an infinite number of subspecs since some specification languages allow a notation for indefinite repetition. In addition, subspecs that differ only by the number of repetitions of some transformations may be represented as a single subspec.

```
Description
   TRAP computes the AREA between the curve F and
   the x-axis from x=A to x=B by the trapezoidal
   rule using N intervals of size (B-A)/N.
   When the curve is above the x-axis, the area
   is positive, otherwise the area is negative.
   ERROR=true if N<1, else ERROR=false.

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

Operation
   if N<1 then
      ERROR := true;
      AREA := 0.0;
   else
      ERROR := false;
      AREA := sum(i:=1,N,
              (F(A+(i-1)*H) + F(A+i*H))/2)*abs(H);
   endif;

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

Figure 1.
Specification of Procedure TRAP

```
   procedure TRAP(function F(X: real): real;
             A,B: in REAL; N: in INTEGER;
             AREA: out REAL; ERROR: out BOOLEAN) is
   --TRAP computes the AREA between the curve F and
   --the x-axis from x=A to x=B by the trapezoidal
   --rule using N intervals of size (B-A)/N.
   --When the curve is above the x-axis, the area
   --is positive, otherwise the area is negative.
   --ERROR=true if N<1, else ERROR=false.

      H,X: REAL;
0    begin
      if N<1 then
         --invalid input
1        ERROR:=true;
2        AREA:=0.0;
      else
3        ERROR:=false;
         if A=B then
4           AREA:=0.0;
         else
5           H:=(B-A)/REAL(N);
6           X:=A;
7           AREA:=F(X)/2.0;
            while X<B loop
8              X:=X+H;
9              AREA:=AREA+F(X);
            end loop;
10          AREA:=(AREA-F(X)/2.0)*H;
            if A>B then
11             AREA:=-AREA;
            end if;
         end if;
      end if;
12   end TRAP;
```

Figure 2.
Implementation of Procedure TRAP

names are assigned to the input values as a path is "executed". The values of variables are maintained as algebraic expressions in terms of these symbolic names. The path computation is, therefore, represented by a vector of algebraic expressions for the output values, which may be converted to a canonical form. Similarly, the branching conditions for the conditional statements encountered on a path are represented by constraints in terms of the symbolic names for the input values. The path domain is defined by the conjunct of these constraints, which also may be translated into a canonical form. Symbolic evaluation of an implementation can be extended [4,6] to classify paths that differ only by the number of loop iterations. This technique attempts to represent each loop by a closed form expression. The development of a closed form expression for the loop in procedure TRAP and the symbolic evaluation of the path on which this loop occurs are shown in the appendix. Figure 3 provides the domains and computations for the paths derived by symbolic evaluation of the implementation of procedure TRAP, where a, b, and n are the symbolic names for the input values A, B, and N, respectively.

$D[P_1]$ = {(a,b,n) | (n<1)}
$C[P_1]$ = AREA: 0.0
ERROR:true

$D[P_2]$ = {(a,b,n) | (n≥1) and (a=b)}
$C[P_2]$ = AREA: 0.0
ERROR:false

$D[P_3]$ = {(a,b,n) | (n≥1) and (a>b)}
$C[P_3]$ = AREA: 0.0
ERROR:false

$D[P_4]$ = {(a,b,n) | (n≥1) and (a<b)}
$C[P_4]$ = 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
ERROR:false

Figure 3.
Path Domains and Computations
for the Implementation of Procedure TRAP

$D[S_1]$ = {(a,b,n) | (n<1)}
$C[S_1]$ = AREA: 0.0
ERROR:true

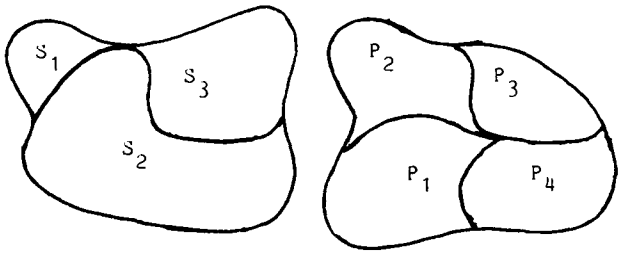
$D[S_2]$ = {(a,b,n) | (n≥1) and (a≤b)}
$C[S_2]$ = AREA: (-a*F(a) - a*F(b) + b*F(a) + b*F(b) -
2*a*sum(i:=1,n-1,F(((n-i)*a+i*b)/n)) +
2*b*sum(i:=1,n-1,F(((n-i)*a+i*b)/n))) / 2*n
ERROR:false

$D[S_3]$ = {(a,b,n) | (n≥1) and (a>b)}
$C[S_3]$ = 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
ERROR:false

Figure 4.
Subspec Domains and Computations
for the Specification of Procedure TRAP

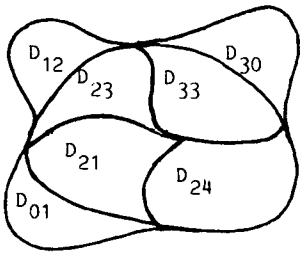
Symbolic evaluation can be applied to a specification in a similar manner, thus providing representations of the subspecs. Symbolic evaluation techniques must be extended to handle the abstract constructs that appear in specification languages. For instance, a specification language may represent the repetition of a transformation by a closed form expression, such as the summation notation that was used in the specification of procedure TRAP. The subspec domains and computations for procedure TRAP, which were derived by symbolic evaluation of the specification, are given in Figure 4.

The specification and implementation impose two partitions on the procedure domain, representing two ways in which a problem may be divided. A partition that takes into account both the specification and implementation can be constructed by overlaying these two partitions. The subdomains so formed are called procedure subdomains and each is the set of input data for which a subspec and a path are applicable. Figure 5 shows a hypothetical example of the procedure subdomains that would result from overlaying partitions of the specification and implementation domains.



partition into subspec domains

partition into path domains



partition into procedure subdomains

Figure 5.
Development of Procedure Subdomains

It would not be surprising to find a subspec domain and a path domain that are equal. The testing and verification of the associated computations can then be considered over this subdomain as a whole. On the other hand, a subspec domain may overlap with more than one path domain or vice versa. This may occur for various reasons. For example, the implementation may handle some input data as a special case for efficiency while the specification concentrates on simplicity.

Alternatively, a discrepancy may be due to a domain or missing path error [10,12]. Each subdomain formed by overlaying the two partitions, therefore, is of interest and should be verified and tested separately. Procedure subdomains appear to be the largest units of input data that can be analyzed independently and yet the smallest units into which the problem can be practically decomposed.

A procedure subdomain D_{IJ} is the intersection of the subspec domain $D[S_I]$ and the path domain $D[P_J]$ - that is, $D_{IJ} = D[S_I] \cap D[P_J]$. In addition, for each subspec S_I, there may be input data in its domain $D[S_I]$ that are not treated by any path; this set, $D_{I0} = D[S_I] - \cup_{J=1,N} D_{IJ}$, 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 subspec; this set, $D_{0J} = D[P_J] - \cup_{I=1,M} D_{IJ}$, is a procedure subdomain as well. The existence of any DIO or D0J implies that the specification domain D[S] and implementation domain D[P] are not the same. Since we assume the specification correctly represents the procedure, the specification domain must be correct. Discrepancies between the domains could imply an error in the implementation or may be due to restrictions on either the specification or implementation domain. For instance, the specification may have input assertions that limit the domain, while the implementation explicitly checks for violations of these assertions.

The representations of the procedure subdomains are constructed from the representations of the subspec and path domains, which are created by symbolic evaluation. Subspec domains and path domains are first compared for equality. For each subspec domain and path domain that are equal, one procedure subdomain is provided. Demonstration of the equality of two domains, say $D[S_K]$ and $D[P_L]$, can sometimes be achieved by a term-by-term comparison of the constraints in their symbolic representations. In procedure TRAP, for instance, $D[S_1] = D[P_1]$ and $D[S_3] = D[P_3]$ are shown in this manner. When domain representations cannot be shown equal by a symbolic comparison, equality can be demonstrated by showing that $D[S_K] \cap \tilde{}D[P_L]$ and $\tilde{}D[S_K] \cap D[P_L]$ are empty. There are several approaches to showing that the intersection of two domains is empty. One approach to this problem is an axiomatic approach, which uses first order predicate calculus to prove whether or not the conjunction defining the intersection is satisfiable. Another approach is an algebraic approach, which attempts to find a solution to the constraints defining the intersection. If the set of constraints is unsatisfiable, the intersection is empty. No method, however, can solve any arbitrary system of constraints [7].

A subspec domain and path domain that are equal need not be considered further. For each remaining subspec S_I and path P_J, the intersection can be constructed by conjoining the representations of the subspec domain and path domain. If this intersection is empty, then no input data exists that causes execution of the path and for which the subspec is applicable; the associated computations are thus trivially equivalent. A non-empty intersection provides procedure subdomain, D_{IJ}, for which computation equality must be considered. For procedure TRAP, these non-empty intersections are $D[S_2] \cap D[P_2]$ and $D[S_2] \cap D[P_4]$.

The representations of the procedure subdomains for procedure TRAP are given in Figure 6. In the partition analysis method, procedure subdomains form the basis for selecting test data and verifying consistency.

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

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

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

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

D_{12}, D_{13}, D_{14}, D_{21}, D_{23}, D_{31}, D_{32}, and D_{34} are empty, as are all D_{I0} and D_{0J}.

Figure 6.
Procedure Subdomains of Procedure TRAP

CONSISTENCY PROPERTIES

The partition analysis method is concerned with determining if a procedure implementation conforms to its specification. In this section, three consistency properties are introduced - compatibility, equivalence, and isomorphism - which differ in the manner in which the implementation conforms to the specification. Partition analysis verification, which is outlined in the next section, is an approach to demonstrating whether these consistency properties hold.

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 both P and S input a vector x, output a vector z, and are defined for the same domain, D[S] = D[P].

In the trivial case, the domain for a particular input is the entire set of values for the type of that input, but some specification and programming languages allow assumptions that further restrict the domain of input values. Note that D[S] = D[P] implies that all elements of each subspec domain are treated by some path and all elements of each path domain are treated by some subspec. Hence, all D_{I0} and D_{0J} procedure subdomains are empty. The definition of compatibility given here may be stronger than necessary. A procedure implementation that explicitly checks for violations of input assertions that restrict the specification domain may, in fact, be correct. To simplify the discussion, however, 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.

Definition: An implementation P is equivalent to a specification S if for all $x \in D[S]$, $P(x) = S(x)$.

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 subspecs and paths over the procedure subdomains. For an input vector x, a particular path, say P_J, is executed - $x \in D[P_J]$ - and a particular subspec, say S_I, is applicable - $x \in D[S_I]$. For this input vector, the specification and the implementation produce the same output values, $S(x) = P(x)$, if and only if the appropriate subspec and path computations agree, $C[S_I](x) = C[P_J](x)$. A subspec S_I and a path P_J compute equal output values for all input data to which they both apply if for all $x \in D_{IJ}$, $C[S_I](x) = C[P_J](x)$. Thus, the computations are equal over the associated procedure subdomain; this is denoted by $C[S_I] =_{D_{IJ}} C[P_J]$. The equivalence of an implementation and a specification, can be restated in terms of the equality of the computations over procedure subdomains.

An implementation P is equivalent to a specification S if and only if for all procedure subdomains D_{IJ}, $1 \le I \le M$ and $1 \le J \le N$, $C[S_I] =_{D_{IJ}} C[P_J]$.

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 for those cases in which it applies. When isomorphism holds each subspec is uniquely associated with an equal path. A subspec S_I and a path P_J are equal if $D[S_I] = D[P_J]$ ($= D_{IJ}$) and $C[S_I] =_{D_{IJ}} C[P_J]$. Thus, the domains are equal and the computations are equal over the procedure subdomain; this is denoted by $S_I = P_J$. This relationship between subspecs and paths leads 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 $B: S \to P$ such that $B(S_I) = P_J$ if and only if $S_I = P_J$.

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.

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 assumption that compatibility must hold simplifies the analysis and yet is a reasonable restriction, since this is often a requirement of an implementation. Slight violations of compatibility can often be handled without additional effort. Isomorphism might be required 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 impose too strict a conformity when a specification is written for comprehensibility, but the implementation must be coded for efficiency. Determining isomorphism, like determining equivalence, is in general an undecidable problem. In practice, however, it can often be accomplished. Since isomorphism and equivalence are defined here in terms of the relationships between the subspecs and paths, the partition analysis method will be driven by the procedure subdomains. By dividing the problem domain into manageable units, the subdomains divide the process of determining consistency between a specification and implementation into more practical steps. When isomorphism between the specification and implementation does not hold, an isomorphism between a subset of the subspecs and paths can often be determined. Equivalence will then be considered for the remaining subdomains.

PARTITION ANALYSIS VERIFICATION

To demonstrate consistency between a specification and an implementation, partition analysis verification employs several established verification and validation techniques. In this section, demonstration of compatibility is first briefly discussed and then an approach for demonstrating equivalence and isomorphism is outlined. The described method is illustrated for the specification and implementation of procedure TRAP.

Demonstrating compatibility between an implementation and a specification is similar to demonstrating uniformity between procedure interfaces in an implementation [9,19,20] or between levels of design [3,22]. If the specification and programming languages have similar constructs for declaring parameters and global variables, then such declarations can be compared 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 [19] may be utilized to determine the input and output class of each such variable in the implementation and in the specification. In addition, input and output statements within the implementation and specification of the procedure must be considered. Assumptions constraining the input values must be compared to determine the equivalence of the specification and implementation domains. The input values might be constrained by explicit assumptions, such as input assertions, or implicit assumptions [1], such as data formats. 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. For procedure TRAP, the compatibility of the implementation and specification is clear.

Once compatibility is established, partition analysis verification can proceed with the demonstration of additional consistency by comparing the subspec and path domains and the subspec and path computations. Since both the specification and the implementation are unambiguous, the subspec domains are mutually disjoint as are the path domains. No such restriction, however, has been made on the computations; neither the subspec 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, further comparison of a specification and an implementation is driven by the relationships between the subspec domains and the path domains. These relationships are characterized by the procedure subdomains, which thus form the basis for determining equivalence or isomorphism.

Once the procedure subdomains have been constructed, the associated computations are compared. Figure 7 shows the results of this process for procedure TRAP. For each procedure subdomain, the equality of the subspec computation and the path computation over that subdomain must be determined. Often, a term-by-term comparison of the symbolic representations of the subspec computation $C[S_I]$ and the path computation $C[P_J]$ reveals that the two computations are symbolically identical, and thus equal over any domain. In procedure TRAP, for example, $C[S_1]$ and $C[P_1]$ are symbolically identical, as are $C[S_2]$ and $C[P_4]$. Two computations are also 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 subdomain. The most straightforward method for determining whether this holds is to find the solution set of the equation $C[S_I] - C[P_J] = 0.0$. This set can be represented symbolically by a disjunct of the solutions to this equation, as was done for $C[S_2]$ and $C[P_2]$ in procedure TRAP. When the solution set is discrete, the zeroes of a function can be found. 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. Procedure subdomain D_{22} in TRAP restricts the input values to those for which a=b, all lying in the solution set of $C[S_2] - C[P_2] = 0.0$. It can sometimes be determined that the subspec and path computations are not equal over the associated procedure subdomain, thus indicating an error in the implementation.

Partition analysis verification enabled the detection of a fairly subtle error in the implementation of procedure TRAP. In implementing the *while* loop, the incorrect assumption was made that the lower bound on the integral is less than the upper bound; thus the loop exit condition is incorrect. This error was uncovered because it was determined that procedure subdomain D_{33} is not contained in the solution set of $C[S_3] - C[P_3] = 0.0$. Thus, the implementation of TRAP is not equivalent to its specification. A correct implementation could be achieved by replacing the loop exit condition by ((X>B) and (A>B)) or ((X<B) and (A<B)).

When the equality or inequality of the subspec and path computations over the associated procedure subdomain cannot be determined, testing can provide some assurance of their equality or find examples of their inequality. Partition analysis testing is discussed in the next section. Although, in

---

$D_{11} = D[S_1] = D[P_1] = \{(a,b,n) \mid (n<1)\}$
$C[S_1] = C[P_1] =$ AREA: 0.0
ERROR: true
$\Rightarrow S_1 = P_1$

$D_{33} = D[S_3] = D[P_3] = \{(a,b,n) \mid (n\geq 1) \text{ and } (a>b)\}$
$C[S_3] - C[P_3] =$ 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
=(a-b)*(F(a)+F(b)+sum(i:=1,n-1,F(((n-i)*a+i*b)/n))) / 2*n
ERROR: false-false
Solution set: (a=b) or (F(a)+F(b)+sum(i:=1,n-1,F(((n-i)*a+i*b)/n)) = 0.0) $\Rightarrow C[S_3] \neq C[P_3]$ on D_{33}

$D_{22} = D[S_2] \cap D[P_2] = \{(a,b,n) \mid (n\geq 1) \text{ and } (a\leq b) \text{ and } (a=b)\}$
$= \{(a,b,n) \mid (n\geq 1) \text{ and } (a=b)\}$
$C[S_2] - C[P_2] =$ 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
ERROR: false-false
Solution set: (a=b) or (F(a)+F(b)+sum(i:=1,n-1,F(((n-i)*a+i*b)/n)) = 0.0) $\Rightarrow C[S_2] = C[P_2]$ on D_{22}

$D_{24} = D[S_2] \cap D[P_4] = \{(a,b,n) \mid (n\geq 1) \text{ and } (a\leq b) \text{ and } (a<b)\}$
$= \{(a,b,n) \mid (n\geq 1) \text{ and } (a\leq b)\}$
$C[S_2] = C[P_4] =$ 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
ERROR: false
$\Rightarrow C[S_2] = C[P_4]$ on D_{24}

Figure 7.
Demonstration of Equivalence of the
Specification and Implementation of Procedure TRAP

general, computation equality is undecidable, several approaches have been shown effective [21].

Partition analysis verification, which compares the symbolic representations of the domains and computations, is a variation on symbolic testing [14]. Usually, symbolic testing involves merely examining the symbolic representations of the path domains and computations. Partition analysis, however, compares these representations with those derived from the specification. This symbolic testing frequently leads to the detection of errors in the implementation. It is apparent that this method facilitates the detection of computation errors. The example given above demonstrates its utility in detecting domain errors as well.

If partition analysis verification is complete and no errors are detected - that is, if the procedure subdomains are completely determined and all the path computations and subspec computations are shown equal over their respective subdomains - then the implementation is equivalent to the specification. If, in addition, there is a one-to-one correspondence between the subspecs and paths, then the implementation and specification are isomorphic. Note that a subset of the paths and subspecs may be in a one-to-one correspondence, as in procedure TRAP. Since the subspecs and paths in this subset are often similar in structure, it is sometimes easy to determine their equality. Partition analysis verification capitalizes on this by first determining the equal subspecs and paths and by then concentrating on those parts of the implementation that deviate from the specification.

PARTITION ANALYSIS TESTING

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. To remedy this, partition analysis testing complements verification by astutely selecting test data for actual execution. Moreover, when consistency cannot be determined, testing can often substantiate the equality of computations or provide counter examples. The partition into procedure subdomains provides the basis for a test data selection strategy. Each subdomain is a conceptual unit that should be examined carefully and tested independently.

The concept of dividing the problem into smaller units and concentrating on selecting test data for those units is not new. Myers [18] has described some guidelines for partitioning the set of input data into equivalence classes such that one can "reasonably" assume that if the implementation is correct for a representative element of such a class then it is correct for any other element in that class. Path analysis testing strategies [5,12,24] construct a test set by choosing elements from each path domain. This approach uses symbolic representations of the paths provided by symbolic evaluation, but is based solely on the implementation. Howden has proposed a functional testing method [16] that decomposes the implementation into pieces of code that corresponds to subfunctions of the problem. This method symbolically evaluates each subfunction and compares the symbolic representation to the internal documentation and selects test data for each subfunction. Weyuker and Ostrand [23] have proposed a testing method that is based on a partition of the problem that is similar to procedure subdomains. Their method, however, assumes that the specification partition is provided, rather than derived from a specification. Partition analysis decomposes the problem into procedure subdomains using symbolic evaluation of a formal specification and an implementation. In this section, we describe partition analysis testing, which uses the symbolic representations of these subdomains and of their associated computations to assist in selecting test data.

A test data set for a procedure can be constructed by selecting one or more elements from each procedure subdomain. An appropriate selection of input values from each subdomain can increase the probability of detecting errors. The test data selected to exercise the implementation should include typical data points in the subdomain, as well as data explicitly chosen to uncover computation or domain errors.

By examining the representations of the subspec and path computations, test data that is likely to expose computation errors can be selected. Computationally sensitive data that lies within the procedure subdomain may reveal an error in the path computation. Computationally sensitive data [15,18] includes, among other things, data that will cause the computations to be zero, data that causes individual terms within the computation to take on troublesome values, such as 0, 1, or -1, and data of small and large magnitude. For polynomial computations, the number of data points that should be selected to guarantee its correctness may be determined from the degree of the polynomial [8,13].

Test data that is apt to reveal domain errors can be chosen by examining the representation of the procedure subdomain. It has been observed that boundary points are most sensitive to domain errors. Domain testing [24,11] is based on this observation and guides in the selection of test data that lies on and near the boundaries of path domains. By applying this strategy in partition analysis, test data on and near the boundaries of procedure subdomains is selected and errors in the path domain are likely to be detected. Furthermore, whereas most testing strategies are not likely to detect missing path errors, which occur when a subspec is neglected in the implementation, they will in all probability be detected by partition analysis testing. This is because each subspec domain imposes the construction of at least one procedure subdomain; thus data will be chosen that should exercise the missing path.

A categorization of the test data for procedure subdomain D_{24} of procedure TRAP based upon the strategies outlined above is shown in Figure 8. Note that each category does not completely specify an input vector, so categories may be combined in selecting actual test data.

n=1	criteria to
n=2	detect domain
n>>1	errors
a=b-ε	
a=0,b=1	
a=-1,b=0	
a=-1,b=1	
a=-k,b=-1	criteria to
a=1,b=k	detect computation
a=-k,b=k	errors
a=-m,b=+m	
a=0,b=+m	
a=-m,b=0	
b-a=n	

where, ε is a small positive real value
k is a typical positive real value
m is a large positive real value

Figure 8.
Categorization of Test Data to be Selected for Procedure Subdomain D_{24} of Procedure TRAP

Testing an implementation with data selected by the partition analysis testing method should detect most, if not all, program errors. If a path is incorrect, it is unlikely that all the test data selected from the corresponding procedure subdomains will result in correct output. Our initial experimentation supports this claim, although more empirical evidence is needed. The analysis of both the implementation and the specification enables partition analysis testing to generate a more comprehensive set of test data than strategies that rely on a single source of information.

CONCLUSION

The partition analysis method incorporates information derived from both a formal specification and an implementation to assist in program testing and verification. This method relies on the construction of procedure subdomains, which partition the set of input data based on both the implementation and the specification. We have proposed consistency properties that differ in how closely the implementation conforms to the specification. Partition analysis verification compares the implementation and the specification in an attempt to determine whether these properties hold. Partition analysis testing selects test data by analyzing both the implementation and the specification, and thus generates a more comprehensive set of test data than one obtained by analyzing the implementation or the specification alone. In light of the work on symbolic evaluation, we believe the partition analysis method we have proposed could be, at least partially, automated.

There are several problems in the partition analysis method 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 discusses 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 only recently been considered.

While there are several established programming languages, 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. 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.

Currently, partition analysis is concerned with the analysis of an implementation in relation to a specification. It is believed that the method may also be applicable to two specifications of the same problem. This will enable the determination of consistency between consecutive levels of design and the demonstration of reliability of earlier phases of program development. If analysis is not performed throughout the development process, there is no assurance that the specifications indeed capture the desired behavior of the procedures. 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

1. P.W. Abrahams and L.A. Clarke, "Compile-Time Analysis of Data List - Format List Correspondences," IEEE Trans. on Software Engineering, SE-5, 6, November 1979, 612-617.
2. R.S. Boyer, B. Elspas, and K.N. Levitt, "SELECT--A Formal System for Testing and Debugging Programs by Symbolic Execution," Proc. Int. Conf. on Reliable Software, April 1975, 234-244.
3. S.H. Caine and E.K. Gordon, "PDL - A Tool for Software Design," Proc. National Computer Conference, 1975.
4. 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.
5. 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.
6. 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, New Jersey, 1981.

7. M.Davis, "Hilbert's Tenth Problem is Unsolvable," American Math. Mon., 80, March 1973, 233-269.
8. 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.
9. C. Gannon, "JAVS: A JOVIAL Automated Verification System," Proc. COMPSAC '78, November 1978.
10. 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.
11. J. Hassell, L.A. Clarke, and D.J. Richardson, "A Close Look at Domain Testing," Computer and Information Science, University of Massachusetts, Amherst, TR-80-16, October 1980.
12. W.E. Howden, "Reliability of the Path Analysis Testing Strategy," IEEE Trans. on Software Engineering, SE-2, 3, September 1976, 208-215.
13. W.E. Howden, "Algebraic Program Testing," Department of Applied Physics and Information Science, University of California, San Diego, TR-12, November 1976.
14. W.E. Howden, "Symbolic Testing and the DISSECT Symbolic Evaluation System," IEEE Trans. on Software Engineering, SE-3, 4, July 1977, 266-278.
15. W.E. Howden, "A Survey of Dynamic Analysis Methods," Tutorial: Software Testing and Validation Techniques, IEEE Computer Society, Long Beach, CA, 1978, 184-206.
16. W.E. Howden, "Functional Program Testing," IEEE Trans. on Software Engineering, SE-6, 2, March 1980, 162-169.
17. J.C. King, "Symbolic Execution and Program Testing," CACM, 19, 7, July 1976, 385-394.
18. G.J. Myers, The Art of Software Testing, John Wiley and Sons, New York, 1979.
19. 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.
20. 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.
21. 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.
22. 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.
23. E.J. Weyuker and T.J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains," IEEE Trans. on Software Engineering, SE-6, 3, May 1980, 236-246.
24. L.J. White and E.I. Cohen, "A Domain Strategy for Computer Program Testing," IEEE Trans. on Software Engineering, SE-6, 3, May 1980, 247-257.

APPENDIX

Symbolic evaluation is typically applied to each path in a procedure. A procedure with loops, however, may have an effectively infinite number of paths. The symbolic evaluation method employed by partition analysis uses a loop analysis technique [4,6] to represent the loops in a procedure by a closed form expression. Using this technique, paths that differ only by their number of loop iterations are classified as a path. Procedures only contain a finite, and usually small, number of classes of paths so that symbolic evaluation can be applied to each such class.

In this symbolic evaluation method, loops are analyzed first in an attempt to generate the closed form expressions. For each analyzed loop, a conditional expression is created representing the final iteration count for any arbitrary execution of the loop. The final iteration count is expressed in terms of the symbolic values of the variables at entry to the loop. In addition, for each variable modified within the loop, its symbolic value at exit from the loop is created. Each such expression is in terms of the final iteration count as well as the symbolic values of the variables at entry to the loop.

Figure A.1 shows the loop analysis for the while loop in TRAP. To initiate loop analysis, an iteration counter, k, is associated with the loop. The symbolic names X_k and $AREA_k$ are used to represent the values of the variables X and AREA at the beginning of the kth iteration of the loop. Note that X_1 and $AREA_1$ represent the values on entry to the first iteration of the loop. Symbolic evaluation of a representative iteration, k-1, is performed, thus providing the symbolic values for each X_k and $AREA_k$, $k \geq 2$, as recurrence relations in terms of the values of the variables at iterations k and k-1. A representation for the loop exit condition, denoted LECk, is also obtained; this is the condition under which the loop will be exited before the kth iteration (after k-1 iterations). Loop analysis then solves the recurrence relations, in terms of X_1 and $AREA_1$. The solutions are given by providing X(k) and AREA(k). The solution for the loop exit condition, LEC(k), is obtained by replacing X_k and $AREA_k$ by X(k) and AREA(k) in the condition. Finally, the closed form representation of a loop can be created. The fall-through case, in which the values at entry to the first iteration of the loop satisfy the loop exit condition, must be added to the loop representation. The final iteration count k_e is the iteration before which exit occurs and is the minimum k, such that the loop exit condition is true. The symbolic values of the variables at exit from the loop are represented by $X(k_e)$ and $AREA(k_e)$.

The closed form representation of a loop captures the behavior of the loop. Thus, when a loop is encounterred during symbolic evaluation of a path, the loop body is evaluated by "executing" its closed form representation. The while loop in TRAP is encounterred along paths P_3 and P_4. Path P_3 exits the loop before the first iteration, and thus represents a single path. Path P_4 represents the class of paths that perform one or more iterations of the while loop. The symbolic evaluation of path P_4 appears in Figure A.2.

```
Recurrence Relations (k≥2)
   X_k = X_k-1 + H
   AREA_k = AREA_k-1 + F(X_k)
Loop Exit Condition  (k≥2)
   LECk = ~(X_k<B)

Solved Recurrence Relations  (k≥2)
   X(k) = X_1 + (k-1)*H
   AREA(k) = AREA_1 + sum(i:=2,k,F(X_1+(i-1)*H))
Solved Loop Exit Condition  (k≥2)
   LEC(k) = X_1 + (k-1)*H ≥ B

Closed Form Representation
   if X_1 ≥ B then
      --exit loop before first iteration
      X = X_1
      AREA = AREA_1
   else if (X_1<B) and
           exists(k_e: integer in  [2.. ) =>
           k_e=minimum {k: integer in  [2.. ) =>
           (X_1+(k-1)*H ≥ B)})
      X = X_1 + (k_e-1)*H
      AREA = AREA_1 + sum(i:=2,k_e,F(X +(i-1)*H))
   endif
```

Figure A.1.
Loop Analysis of wnile loop in Procedure TRAP

statement or edge	condition defining path domain	changes in path computation
0	TRUE	(A=a,B=b,N=n AREA=$,ERROR=$, H=$,X=$)
(0,3)	TRUE and ~(n<1) =(n≥1)	
3		ERROR=false
(3,5)	(n≥1) and ~(a=b) =(n≥1) and (a/=b)	
5		H=(b-a)/n
6		X=a
7		AREA=F(a)/2.0
loop(8-9)	(n≥1) and (a/=b) and (a<b) and k_e=minimum{k: integer in [2..) => ($a+(k_e-1)*(b-a)/n \geq b$)} =(n≥1) and (a<b) and (k_e=n+1)*	
		X=a+(k_e-1)*(b-a)/n AREA=F(a)/2.0+sum(i:=2,k_e,F(a+(i-1)*(b-a)/n))
10		AREA=(F(a)/2.0+sum(i:=2,k_e,F(a+(i-1)*(b-a)/n))- F(a+(k_e-1)*(b-a)/n)/2.0)*(b-a)/n
(10,12)	(n≥1) and (a<b) (k_e=n+1) and ~(a>b) =(n≥1) and (a<b) and (k_e=n+1)	
12		end

Figure A.2.
Symbolic Evaluation of Path P_4
in the Implementation of Procedure TRAP

*Elementary algebraic techniques were used to solve for k_e.