Partition Analysis: A Method Combining Testing and Verification

Debra J. Richardson
Lori A. Clarke

COINS Technical Report 85-10

Software Development Laboratory
Computer and Information Science
University of Massachusetts
Amherst, MA 01003

This research was supported in part by the National Science Foundation under grant MCS-8104202.

ABSTRACT

The partition analysis method compares a procedure's implementation to its specification, both to verify consistency between the two and to derive test data. Unlike most verification methods, partition analysis is applicable to a number of different types of specification languages, including both procedural and nonprocedural languages. This means it can be used on high-level descriptions as well as on low-level designs. Partition analysis also improves upon existing test data selection strategies. These strategies usually consider only the implementation, but partition analysis selects test data that characterize the procedure in terms of both its intended behavior (as described in the specifications) and the structure of its implementation. To accomplish its goals, partition analysis divides or *partitions* the procedure's domain into subdomains in which all elements of each subdomain are treated uniformly by the specification and processed uniformly by the implementation. This 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 and the verification process are designed to enhance each other. Initial experimentation has shown that through the integration of testing and verification, as well as through the use of information derived from both the implementation and the specification, the partition analysis method is effective for determining program reliability. This paper describes the partition analysis method and reports the results obtained from an evaluation of its effectiveness.

Keywords: software testing, software verification, symbolic evaluation.

1. INTRODUCTION

It has been repeatedly demonstrated that neither testing nor verification can guarantee the correctness of a program (e.g., [DEMI79,DIJK70]). Yet either technique, when judiciously applied, uncovers errors. By building upon the strengths of both these techniques, *partition analysis* [RICH81a] combines them into a more effective method for determining program reliability.

The partition analysis method incorporates information from a specification with information from the corresponding implementation. This is done by applying symbolic evaluation techniques to both the specification and implementation, thereby providing a functional representation of both. These functional representations are in terms of partial functions over subdomains of the input domain. By forming the intersection of the specification subdomains with the implementation subdomains, the subdomains of the *procedure partition* are formed.

The subdomains of the procedure partition reflect the partitioning of the problem by both the specification and implementation, indicating both where they are the same and where they are different. Because of the way they are derived, the elements for each procedure subdomain are treated in a uniform, and usually simple, manner by the specification. They are also treated in a uniform, and usually simple, manner by the implementation. Evaluating any differences in these treatments, using both verification and testing strategies, is the crux of the partition analysis method.

Thus, partition analysis is composed of three steps. First, symbolic evaluation and

other analysis techniques are applied to determine the procedure partition. For each subdomain in this partition, there is a description of the elements in the domain, a description of the computation to be performed on those elements as specified by the specification, and a description of the computation to be performed on those elements as specified by the implementation. Second, verification techniques are applied to the two computational descriptions to determine their consistency over the specified subdomain. Third, the subdomain and computational descriptions are used to derive an extensive test data set that attempts to characterize the functional behavior of the subdomain. In addition, problems that arise in the verification process are used to drive some aspects of the test data selection process.

There are several advantages of the partition analysis method. It is one of the few methods that tries to combine the complementary approaches of testing and verification. Furthermore, partition analysis derives a more comprehensive set of test data than other testing methods. Most testing methods select test data based only on the program structure and thus test the actual behavior of the implementation rather than its intended behavior. By basing test data selection on the procedure partition, however, partition analysis derives a set of test data that characterizes both the specification and the implementation, and consequently both the intended and actual behavior. Moreover, the testing strategy used by partition analysis involves the integration of a number of complementary testing criteria. This has led to some interesting results and is currently the focus of additional research.

Another advantage of partition analysis is its widespread applicability. The method can be employed with a number of different specification languages, such as high-level formal specification languages based on predicate calculus [FLOY67,HOAR71] or state transformation [SILV79] as well as low-level procedural languages [KERN83]. In fact, if the relationships between the data objects can be established (as will be discussed later), the method can be applied to any two levels of description such as a high-level specification with a design or one level of design specification with another. To be applicable with a language, it is only necessary that descriptions in the language can be reformulated as subdomains and associated computations. For predicate calculus specifications, for example, this requires rewriting the specification in disjunctive normal form. Each type of specification language requires different processing. A few, such as algebraic specification, do not appear to be amenable to this type of analysis.

In our evaluation of partition analysis we used a hybrid specification language of our own making, called SPA [RICH81b]. SPA is an extended PDL/Ada [KERN83] that combines predicate calculus and procedural constructs. In particular, the SPA language has constructs for describing conditional values, existential and universal quantification, finite summation and product, assertions, and encapsulations such as abstract data types, as well as the typical procedural programming language constructs. Since SPA has a variety of high-level and low-level constructs, it facilitates the representation of an abstract specification as well as successively more detailed designs. SPA's flexibility allows it to be used throughout pre-implementation.

```
   procedure PRIME(N: in integer inset {1...}) return boolean =

     -- PRIME returns true if N is prime or false if N is not prime

s  begin
   return case

1          N = 1 ⟹
2            false;

3          N = 2 ⟹
4            true;

5          otherwise ⟹
             -- if N has no factor <= N - 1, N has no factor
6            forall {i: integer inset {2 .. N - 1} | (N mod i /= 0)};

           endcase;
f  end PRIME;
```

Figure 1: Specification of PRIME

In Figure 1, SPA is used to specify a procedure to determine whether a number is prime. This specification was developed by formalizing the simple mathematical properties of a prime number. An implementation of *PRIME* in Ada [WEGN80] appears in Figure 2. This implementation makes use of several facts that improve on efficiency. The procedure *PRIME* is used throughout this paper to illustrate the partition analysis method. The full application of partition analysis to *PRIME* is provided elsewhere [RICH81b].

In this paper we describe the partition analysis method, demonstrating how it can be

```
    function PRIME(N: in integer range 2..max'int) return boolean is
      -- implementation in Ada
      -- PRIME returns true if N is prime or false if N is not prime

      FAC: integer;
      ISPRIME: boolean;

s   begin
1     if N mod 2 = 0 or N mod 3 = 0 then
        -- if N is even and N /= 2 or N is divisible by 3 and N /= 3,
        -- N is not PRIME
2       ISPRIME := (N < 4);

      else
        -- if N is odd, any FACtor of N is odd
        -- if N is not divisible by 3, N has no FACtor in the sequence 9, 15, 21, ...
        -- if N has no FACtor <= sqrt (N), N has no FACtor
        -- loop checking for FACtors in the sequence 5, 7, 11, 13, 17, 19, ...
3       ISPRIME := true;
4       FAC := 5;
5       while FAC * *2 <= N loop
6         if N mod FAC = 0 or N mod (FAC + 2) = 0 then
7           ISPRIME := false;
            exit;
          else
8           FAC := FAC + 6;
          endif;
9       endloop;

      endif;
10    return ISPRIME;
f   end PRIME;
```

Figure 2: Implementation of PRIME

applied to procedural languages at the design and implementation level. The next three sections, respectively, describe each of the three steps: forming the procedure partition, partition analysis verification, and partition analysis testing. Section 5 reports on an experimental evaluation of partition analysis. This evaluation involved the application of partition analysis to thirty-four procedures, which were obtained from the software engineering literature and programming texts. The reliability of the partition analysis method was measured in terms of its ability to detect errors in those procedures and by mutation analysis [DEMI78b], which measures the adequacy of the selected test data. The final section discusses limitations of the method and areas of future research.

2. FORMING THE PROCEDURE PARTITION

A procedure corresponds to the mathematical concept of a function — that is, a mapping from a domain to a codomain. As is evident in the *PRIME* example, a specification and an implementation are intended to be descriptions of the same function at different levels of abstraction. To facilitate a comparison of these two descriptions, the partition analysis method decomposes both descriptions into functional representations, called the specification partition and the implementation partition. For procedural languages, symbolic evaluation techniques [CHEA79,CLAR81] are employed to create these representations. If the specification and implementation have consistent interfaces, they are said to be compatible. Provided that there are no major violations of compatibility, the specification and implementation partitions are combined to form the procedure partition, which

forms the basis for partition analysis verification and testing. This section describes the specification and implementation partitions for procedural languages and the symbolic evaluation techniques employed to develop them. The property of compatibility is then considered. Finally, the actual construction of the procedure partition is outlined.

2.1 Creating the Functional Representations

A function F is a mapping from a set of input values X, called the domain of F, to a set of output values Z, called the codomain of F, thus $F : X \mapsto Z$. For a function that corresponds to a procedure with m input parameters [1] , an input value of F is a vector x corresponding to a point in the m-dimensional domain X. An output value z of F is the image $F(x)$ for some x in X. If the procedure corresponding to F has n output parameters, an output value z is a vector corresponding to a point in the n-dimensional codomain Z. In general, the mapping of a function F is partitioned as a set of partial mappings —

$$F = \{F_1, F_2, \ldots, F_L | 1 \leq L \leq \infty\}.$$

Each partial mapping F_H is defined over a subset of the domain X, called the *domain of definition* of F_H and denoted $D[F_H]$, and undefined elsewhere in X. The *function domain* $D[F]$, or domain of definition of F, is the union of the domains of definition of the partial mappings, $\bigcup_{H=1}^{L} D[F_H]$, and is a subset of the domain X.

Both an implementation and a specification have a structure that can be related to

[1] For simplicity in the presentation we only consider procedures where input and output is only done via a known number of parameters. In practice, this restriction can easily be dropped.

the structure described for the function of a procedure. Both represent a mapping from a domain to a codomain and partition their domain by describing partial mappings over certain subdomains. The partition analysis method applies symbolic evaluation techniques to both the specification and the implementation to derive functional representations of the two descriptions of a procedure. The similarities and differences between the partial mappings described by the functional representations are then examined.

In applying symbolic evaluation techniques to an implementation, symbolic names are assigned for the input values of the parameters and a *path*, a sequence of statements through the implementation, is interpreted. While interpreting the statements on the path, the values of all variables are maintained as algebraic expressions in terms of the symbolic names. Similarly, the branch predicates for the conditional statements on a path are represented by constraints in terms of the symbolic names. After symbolically evaluating a path P_J, its symbolic representation consists of the *path computation* $C[P_J]$, which is a vector of algebraic expressions for the output parameters, and the *path domain* $D[P_J]$, which is defined by the conjunction of the path's branch predicate constraints. Some paths through a program may not be executable. The path domain for such a path is empty since the conjunction of the path's branch predicate constraints is inconsistent. Although constraint consistency can not always be determined, in practice it can usually be done.

Most implementations contain loops, thus a symbolic representation of all executable paths through such a program is usually unreasonable since there may be an effectively

infinite number of executable paths. With the addition of loop analysis techniques paths that differ only by the number of iterations of loops can often be represented as a class of paths [CHEA79,CLAR81]. Loop analysis techniques attempt to replace each loop by a closed form expression that captures the effect of the loop. When successful, this enables an implementation to be decomposed as a finite set of classes of paths.

Hence, an implementation P is represented by the *implementation partition*, which is the set of domains and computations of the (classes of) executable paths in P,

$$\{ (D[P_1], C[P_1]), \ldots, (D[P_N], C[P_N]) \mid 1 \leq N \leq \infty \}.$$

The domain of definition of P is the union of the path domains, $\bigcup_{J=1}^{N} D[P_J]$, and is called the *implementation domain* $D[P]$. The implementation partition of $PRIME$ is shown in Figure 3.

Note that the symbolic representations of the domains for paths P_5 and P_6 contain a closed form representation for the loop in $PRIME$.

Symbolic evaluation and loop analysis techniques have been extended to be applicable to SPA specifications [RICH81b]. Using these techniques, a feasible sequence of statements through a specification, referred to as a *subspec*, is evaluated in terms of symbolic names assigned to the input values. A specification can then be decomposed as a finite set of (classes of) subspecs. Each subspec S_I is described by a *subspec domain* $D[S_I]$ and a *subspec computation* $C[S_I]$. The *specification partition* that represents a specification S is the set of domains and computations of the (classes of) subspecs in S,

$$\{ (D[S_1], C[S_1]), \ldots, (D[S_M], C[S_M]) \mid 1 \leq M \leq \infty \}.$$

P_1 : $(s, 1, 2, 10, f)$
$D[P_1]$: $((trunc(n/2) * 2 - n = 0)$ or $(trunc(n/3) * 3 - n = 0))$
$C[P_1]$: $(n < 4)$

P_2 : $(s, 1, 3, 4, 5, 9, 10, f)$
$D[P_2]$: $(n < 25)$ and $(trunc(n/2) * 2 - n \neq 0)$ and $(trunc(n/3) * 3 - n \neq 0)$
$C[P_2]$: $true$

P_3 : $(s, 1, 3, 4, 5, 6, 7, 9, 10, f)$
$D[P_3]$: $(n \geq 25)$ and $(trunc(n/2) * 2 - n \neq 0)$ and $(trunc(n/3) * 3 - n \neq 0)$
and $((trunc(n/5) * 5 - n = 0)$ or $(trunc(n/7) * 7 - n = 0))$
$C[P_3]$: $false$

P_4 : $(s, 1, 3, 4, 5, 6, 8, 5, 9, 10, f)$
$D[P_4]$: $(n \geq 25)$ and $(n < 121)$
and $(trunc(n/2) * 2 - n \neq 0)$ and $(trunc(n/3) * 3 - n \neq 0)$
and $(trunc(n/5) * 5 - n \neq 0)$ and $(trunc(n/7) * 7 - n \neq 0)$
$C[P_4]$: $true$

P_5 : $(s, 1, 3, 4, (5, 6, 8), 5, 6, 7, 9, 10, f)$
$D[P_5]$: $(n \geq 121)$ and $(trunc(n/2) * 2 - n \neq 0)$ and $(trunc(n/3) * 3 - n \neq 0)$
and exists$\{k_e := 2 \ldots \mid ((trunc(n/(6 * k_e - 1)) * (6 * k_e - 1) - n = 0)$
or $(trunc(n/(6 * k_e + 1)) * (6 * k_e + 1) - n = 0))$
and forall$\{k := 1..k_e - 1 \mid (trunc(n/(6 * k - 1)) * (6 * k - 1) - n \neq 0)$
and $(trunc(n/(6 * k + 1)) * (6 * k + 1) - n \neq 0)$ and $(36 * k * *2 + 60 * k - n \leq -25)\}\}$
$C[P_5]$: $false$

P_6 : $(s, 1, 3, 4, (5, 6, 8), 5, 9, 10, f)$
$D[P_6]$: $(n \geq 121)$ and $(trunc(n/2) * 2 - n \neq 0)$ and $(trunc(n/3) * 3 - n \neq 0)$
and exists$\{k_e := 2 \ldots \mid (trunc(n/(6 * k_e - 1)) * (6 * k_e - 1) - n \neq 0)$
and $(trunc(n/(6 * k_e + 1)) * (6 * k_e + 1) - n \neq 0)$ and $(36 * k_e * *2 + 60 * k_e - n > -25)$
and forall$\{k := 1..k_e - 1 \mid (trunc(n/(6 * k - 1)) * (6 * k - 1) - n \neq 0)$
and $(trunc(n/(6 * k + 1)) * (6 * k + 1) - n \neq 0)$ and $((36 * k * *2 + 60 * k - n \leq -25)\}\}$
$C[P_6]$: $true$

Figure 3: Implementation Partition of PRIME

S_1 : $(s, 1, 2, f)$
$D[S_1]$: $(n = 1)$
$C[S_1]$: $false$

S_2 : $(s, 1, 3, 4, f)$
$D[S_2]$: $(n = 2)$
$C[S_2]$: $true$

S_3 : $(s, 1, 3, 5, 6, f)$
$D[S_3]$: $(n \geq 3)$
$C[S_3]$: $forall\{i := 2..n - 1 \mid (trunc(n/i) * i - n \neq 0)\}$

Figure 4: Specification Partition of PRIME

The domain of definition of S is the union of the subspec domains, $\bigcup_{I=1}^{M} D[S_I]$, and is called the *specification domain* $D[S]$. Figure 4 provides the specification partition of $PRIME$.

2.2 Determining Specification and Implementation Compatability

The specification and implementation partitions provide common representations of the specification and implementation of a procedure. The partition analysis method attempts to determine consistency between these two descriptions of the procedure by comparing these partitions. This comparison begins by determining whether the implementation and the specification have consistent interfaces — that is, whether they have the same number and type of inputs and outputs and the inputs are restricted to values from the same domain. This property is referred to as compatibility.

Definition: An implementation P is *compatible* with a specification S if both accept as input a vector $x = (x_1 : X_1, \ldots, x_m : X_m)$ in domain $X = X_1 \times \cdots \times X_m$, produce as output a vector $z = (z_1 : Z_1, \ldots, z_n : Z_n)$ in codomain $Z = Z_1 \times \cdots \times Z_n$, and are defined over the same domain, $D[S] = D[P]$.

In the trivial case, the domain X_i for input x_i or the codomain Z_j for output z_j is the entire set of values for a predefined type (e.g., the set of integers). Some specification and programming languages have constructs that can be used to further restrict the input and output values. User-defined types, such as those provided by Pascal and Ada, support such restrictions. Moreover, the domain over which a specification or an implementation is defined may be constrained by initial assertions, such as those provided by Gypsy and Alphard. When input and output statements are included, then the restrictions imposed by format statements must also be considered [ABRA79]. The specification language SPA has constructs for both user-defined types and initial assertions.

To demonstrate compatibility, the correspondence between the specification's parameters and those of the implementation must be determined and then equality of the domains must be shown. The correspondence between parameters can be determined by matching names, comparing the types and input/output modes of the parameters, or, as a last resort, asking the user. Finally, the domain over which the specification and implementation are defined must be checked for equality by comparing any statements restricting the application of the specification and the implementation.

Violations of compatability are sometimes unavoidable due to the differences in the

specification and implementation languages. Some programming languages, FORTRAN for instance, do not support either user-defined types or initial assertions. Implementations written in such a language have input parameters whose domain is determined by the predefined types of the programming language. If such an implementation explicitly checks for violations of user-defined types or initial assertions that restrict the specification domain, it is reasonable to consider the two compatible. Further, the implementation may never be executed for input values that are not in the specification domain, and violations of compatibility may not be a problem. The definition of compatibility given here, therefore, is stronger than necessary.

Certain violations of compatibility can be handled within the partition analysis method. In the situation described above, in which compatibility is not satisfied due to the inequality of the specification domain and the implementation domain, partition analysis notes the discrepancy between the domains of definition but continues by considering only the domain over which both the specification and implementation are defined, $D[S] \cap D[P]$ (in most cases this will be the specification domain D[S]). Partition analysis can also continue when the input/output mode of a parameter in the specification is different than in the implementation; the mode indicated by the specification is assumed. Partition analysis can not continue when a violation of compatibility inhibits the construction of the procedure partition. The procedure partition could not be constructed, for example, if the specification and implementation have different numbers of parameters or if the base types of a parameter are different.

The specification and implementation of $PRIME$ are not compatible. While both the specification and the implementation state that the input parameter N is an integer, the specification indicates that $N \geq 1$, but the implementation indicates that $N \geq 2$. This does not inhibit the construction of the procedure partition, and partition analysis can continue by comparing the implementation and the specification over their mutual domain $N \geq 2$.

2.3 Construction of the Procedure Partition

The specification and implementation impose two partitions on a procedure, representing two ways in which the procedure may be divided. It is not surprising to find a subspec domain and a path domain that are equal. The testing and verification of the subspec and path computations can then be considered over this subdomain as a whole. On the other hand, there are often differences between these two partitions; a subspec domain may overlap with more than one path domain or vice versa. Such a discrepancy may be due to an error. Alternatively, this may not be indicative of an error, but rather occurs because the specification is a more abstract description of the problem than the implementation. $PRIME$ provides an excellent illustration of the variation that can occur between the different levels of abstraction. The one element in the "$N = 2$" subspec domain and some of the elements in the "otherwise" subspec domain, those for which N is divisible by 2 or 3, are grouped in the "$N \ mod \ 2 = 0$ or $N \ mod \ 3 = 0$" path domain, hence a path domain overlaps with more than one subspec domain. The other elements in the "otherwise"

subspec domain, those for which N is not divisible by 2 or 3, are in the remaining path domains, hence a subspec domain overlaps with more than one path domain.

It is clearly not adequate to use either the specification partition alone or the implementation partition alone as the basis for demonstrating program reliability. As is seen in the procedure $PRIME$, both partitions must be considered or potentially useful information is lost. The procedure partition is constructed by overlaying, or intersecting, these two partitions, thereby taking into account both descriptions. Each subdomain so formed is the set of input data for which a subspec and a path are mutually applicable.

The subdomains are constructed by taking the pairwise intersection of the set of subspec domains and the set of path domains. The nonempty intersection of a subspec domain $D[S_I]$ and a path domain $D[P_J]$ is referred to as a *procedure subdomain*, and denoted D_{IJ} — that is, $D_{IJ} = D[S_I] \cap D[P_J] \neq \emptyset$. Associated with each such intersection are two computations, the subspec computation and the path computation, which are intended to specify equal values for all elements to which they both apply — that is, all elements in the procedure subdomain. The *computation difference* C_{IJ} for a procedure subdomain D_{IJ} is the difference between the subspec computation $C[S_I]$ and the path computation $C[P_J]$ — that is, $C_{IJ} = C[S_I] - C[P_J]$. If the specification domain and the implementation domain are not compatible, there may be input data in a subspec domain $D[S_I]$ that are not treated by any path; this set is denoted $D_{I0} = D[S_I] \setminus D[P] \neq \emptyset$ (where $\setminus$ is the set difference operator). In addition, there may be input data in a path domain $D[P_J]$ that are not treated by any subspec; this set is denoted $D_{0J} = D[P_J] \setminus D[S] \neq \emptyset$.

Procedure subdomains of these types, D_{I0} or D_{0J}, are called *noncompatible* procedure subdomains (since their existence implies that the compatibility property is not satisfied), while all others are called *compatible*. For any noncompatible procedure subdomain D_{I0} or D_{0J}, there is only one associated computation, $C[S_I]$ or $C[P_J]$, respectively, the other computation is undefined. The computation difference for a noncompatible procedure subdomain D_{I0} or D_{0J} is represented by $C_{I0} = C[S_I] - \lambda$ or $C_{0J} = C[P_J] - \lambda$, respectively (where λ represents the undefined result).

Thus, the *procedure partition* is composed of the procedure subdomains and the associated computation differences,

$$\begin{aligned}&\{\ (D_{IJ}, C_{IJ}) \mid 1 \le I \le M \text{ and } 1 \le J \le N \text{ and } D[S_I] \cap D[P_J] \neq \emptyset\ \} \\ &\cup\{\ (D_{I0}, C_{I0}) \mid 1 \le I \le M \text{ and } D[S_I] \setminus D[P] \neq \emptyset\ \} \\ &\cup\{\ (D_{0J}, C_{0J}) \mid 1 \le J \le N \text{ and } D[P_J] \setminus D[S] \neq \emptyset\ \}.\end{aligned}$$

To help conceptualize the formation of this partition, Figure 5 shows a hypothetical example of the procedure subdomains that would result by overlaying partitions of a specification and an implementation domain.

The construction of the procedure partition is driven by the relationships between the subspec and path domains, because if for a particular subspec and path pair, the intersection of their domains is empty, then the relationship between their computations is immaterial. Moreover, since both the specification and the implementation are deterministic, the subspec domains are mutually disjoint as are the path domains. No such restriction is made on the computations; the subspec computations need not be distinct nor need the path computations. For these reasons, the creation of the procedure partition is driven

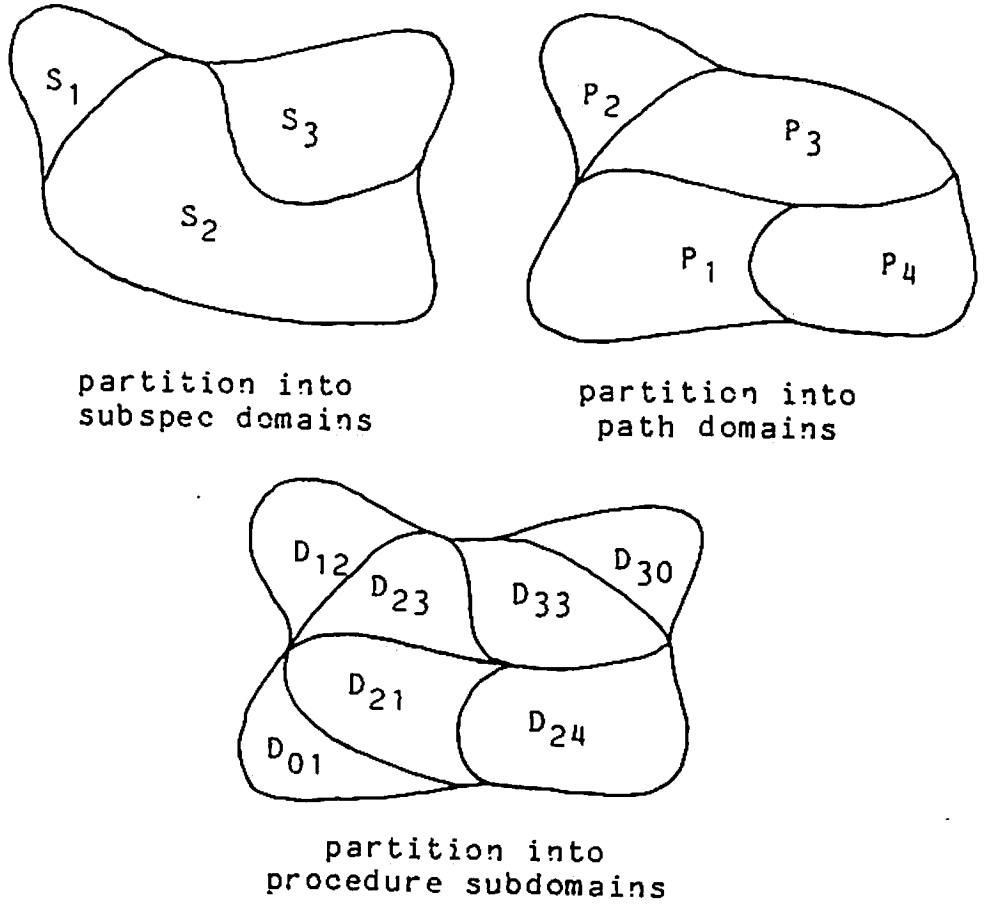

Figure 5: Hypothetical Procedure Subdomains

by the the relationships between the subspec domains and the path domains. The construction of the procedure subdomains and computation differences is illustrated for the procedure $PRIME$ and the procedure partition created for $PRIME$ appears in Figure 6. In the construction of the procedure partition, subspec domains and path domains are first compared for equality. For each subspec domain $D[S_K]$ and path domain $D[P_L]$ that are equal, one procedure subdomain D_{KL} is provided. Demonstration of the equality of these two domains can sometimes be achieved by a term-by-term comparison of the constraints in their symbolic representations. When domain representations cannot be shown equal by such a comparison, equality can be demonstrated by showing that $D[S_K] \cap \neg D[P_L]$ and $D[S_K] \cap \neg D[P_L]$ are empty. Since the subspec domains are disjoint, as are the path domains, a subspec domain that is equal to a path domain does not have any elements in common with any other path domain, and vice versa. Thus, a subspec domain and a path domain that are determined to be equal need not be considered further in the pairwise intersection of the subspec domains and the path domains.

For each remaining subspec S_I and path P_J, their intersection is constructed by conjoining the representations of the subspec domain $D[S_I]$ and path domain $D[P_J]$ — that is, $D_{IJ} = D[S_I] \wedge D[P_J]$. In procedure $PRIME$, $D[S_2]$ has elements in common with $D[P_1]$ and $D[S_3]$ has elements in common with each of the path domains. Hence, compatible procedure subdomains D_{21}, D_{31}, D_{32}, D_{33}, D_{34}, D_{35}, and D_{36} are derived.

For each compatible procedure subdomain D_{IJ}, the symbolic representation of C_{IJ} is derived by forming the difference between the symbolic representation of the path computa-

D_{11} : $(n = 2)$
C_{11} : $(true) - (n < 4)$

D_{21} : $(n \geq 3)$ and $((trunc(n/2) * 2 - n = 0)$ or $(trunc(n/3) * 3 - n = 0))$
C_{21} : $(\text{forall}\{i := 2..n - 1 \mid (trunc(n/i) * i - n \neq 0)\}) - (n < 4)$

D_{22} : $(n \geq 3)$ and $(n < 25)$ and $(trunc(n/2) * 2 - n \neq 0)$
and $(trunc(n/3) * 3 - n \neq 0)$
C_{22} : $(\text{forall}\{i := 2..n - 1 \mid (trunc(n/i) * i - n \neq 0)\}) - (true)$

D_{23} : $(n \geq 25)$ and $(trunc(n/2) * 2 - n \neq 0)$ and $(trunc(n/3) * 3 - n \neq 0)$
and $((trunc(n/5) * 5 - n = 0)$ or $(trunc(n/7) * 7 - n = 0))$
C_{23} : $(\text{forall}\{i := 2..n - 1 \mid (trunc(n/i) * i - n \neq 0)\}) - (false)$

D_{24} : $(n \geq 25)$ and $(n < 121)$ and $(trunc(n/2) * 2 - n \neq 0)$
and $(trunc(n/3) * 3 - n \neq 0)$ and $(trunc(n/5) * 5 - n \neq 0)$
and $(trunc(n/7) * 7 - n \neq 0)$
C_{24} : $(forall\{i := 2..n - 1 \mid (trunc(n/i) * i - n \neq 0)\}) - (true)$

D_{25} : $(n \geq 121)$ and $(trunc(n/2) * 2 - n \neq 0)$ and $(trunc(n/3) * 3 - n \neq 0)$
and exists $\{k_e := 2 \ldots \mid ((trunc(n/(6 * k_e - 1)) * (6 * k_e - 1) - n = 0)$
or $(trunc(n/(6 * k_e + 1)) * (6 * k_e + 1) - n = 0))$
and forall$\{k := 1..k_e - 1 \mid (trunc(n/(6 * k - 1)) * (6 * k - 1) - n \neq 0)$
and $(trunc(n/(6 * k + 1)) * (6 * k + 1) - n \neq 0)$ and $(36 * k * *2 + 60 * k - n \leq -25)\}\}$
C_{25} : $(\text{forall}\{i := 2..n - 1 \mid (trunc(n/i) * i - n \neq 0)\}) - (false)$

D_{26} : $(n \geq 121)$ and $(trunc(n/2) * 2 - n \neq 0)$ and $(trunc(n/3) * 3 - n \neq 0)$
and exists$\{k_e := 2 \ldots \mid (trunc(n/(6 * k_e - 1)) * (6 * k_e - 1) - n \neq 0)$
and $(trunc(n/(6 * k_e + 1)) * (6 * k_e + 1) - n \neq 0)$ and $(36 * k_e * *2 + 60 * k_e - n > -25)$
and forall$\{k := 1..k_e - 1 \mid (trunc(n/(6 * k - 1)) * (6 * k - 1) - n \neq 0)$
and $(trunc(n/(6 * k + 1)) * (6 * k + 1) - n \neq 0)$ and $((36 * k * *2 + 60 * k - n \leq -25)\}\}$
C_{26} : $(\text{forall}\{i := 2..n - 1 \mid (trunc(n/i) * i - n \neq 0)\}) - (true)$

Figure 6: Procedure Partition of PRIME

tion and the symbolic representation of the subspec computation — that is, $C_{IJ} = C[S_I] - C[P_J]$. In general, both of these symbolic representations are vectors corresponding to the output vector $z = (z_1, z_2, \ldots, z_n)$. For the output value z_i, which corresponds to the ith output parameter, we denote the computational difference by $C_{IJ}.z_i = C[S_I].z_i - C[P_J].z_i$. In procedure $PRIME$, the only output parameter is $PRIME$ and the computation difference for a procedure subdomain D_{IJ} is $C_{IJ}.PRIME = C[S_I].PRIME - C[P_J].PRIME$. Note that in Figure 6, $C_{11} = C[S_1] - C[P_1] : .PRIME = (true) - (n < 4)$ means that $C_{11}.PRIME = (true) - (n < 4)$; this notation is used because it is cleaner when computations are composed of more than one symbolic value.

If the specification domain and the implementation domain are not the same, then the representations for the noncompatible procedure subdomains must also be developed. Of course, if any subspec domain does not intersect any path domain or vice versa, it is clearly a noncompatible procedure subdomain. In procedure $PRIME$, for example, the intersections of the subspec domain $D[S_1]$ with each of the path domains are found to be empty, thus the noncompatible procedure subdomain D_{10} is formed. It is also necessary to determine whether all the elements of a subspec or path domain are in some compatible procedure subdomain. For subspec domain $D[S_I]$, the easiest way to accomplish this is to conjoin the representation of $D[S_I]$ and the negation of the implementation domain $D[P]$; if this represents a nonempty set, then the noncompatible procedure subdomain D_{I0} is formed as $D_{I0} = D[S_I] \wedge \neg D[P]$. Likewise, for each path domain $D[P_J]$, $D_{0J} = D[P_J] \wedge \neg D[S]$. The computation difference for each noncompatible procedure subdomain is trivially derived.

The procedure partition provides the basis for the application of both verification and testing techniques. Each procedure subdomain and its associated computation difference are of interest and should be verified and tested independently of the rest of the procedure partition. In the next two sections, both steps are described along with the manner in which they interact.

3. PARTITION ANALYSIS VERIFICATION

Partition analysis verification attempts to determine consistency between an implementation and a specification. Thus far, compatibility has been demonstrated and the procedure partition has been constructed. Compatibility is a necessary condition for the implementation to be correct with respect to the specification. To realize the function described by the specification, however, the implementation must not only have the same interface, it must compute the specified output values for each input value in the domain. This property is referred to as equivalence.

> **Definition:** Given a specification $S : X \mapsto Z$ and an implementation $P : X \mapsto Z$, with $D[S] = D[P]$, P is *equivalent* with S if and only if for all $x \in D[S]$, $P(x) = S(x)$.

Equivalence between a procedure specification and its implementation implies that the implementation is correct with respect to the specification. This property can be related to the procedure partition. Since both the specification and the implementation are deterministic — that is, for each input vector, one and only one subspec applies and one

and only one path applies — the subspec domains are mutually disjoint as are the path domains. Consequently, the procedure subdomains are mutually disjoint; thus, each input vector is in one and only one procedure subdomain. Given an input vector x, suppose $x \in D_{IJ}$ ($x \in D[S_I]$ and $x \in D[P_J]$). Then $S(x) = P(x)$, if and only if the subspec S_I and the path P_J compute equal output values — $C[S_I](x) = C[P_J](x)$. The subspec computation $C[S_I]$ and the path computation $C[P_J]$ are equal when restricted to their mutual procedure subdomain D_{IJ}, if for all $x \in D_{IJ}$, $C_{IJ}(x) = 0$; this is denoted $C_{IJ}|D_{IJ} = 0$. The equivalence of an implementation and a specification can thus be restated in terms of the equality of the computations over procedure subdomains.

> Given a specification $S : X \mapsto Z$ and an implementation $P : X \mapsto Z$, with $D[P] = D[S]$, P is *equivalent* with S if and only if for all I and J, $1 \leq I \leq M$ and $1 \leq J \leq N$, such that $D[S_I] \cap D[P_J] \neq \emptyset$, $C_{IJ}|D_{IJ} = 0$.

This alternative definition enables the decomposition of the process for determining whether equivalence holds. Equivalence is demonstrated in terms of the procedure partition by proving that for each procedure subdomain, the corresponding subspec and path computations produce equal values for all elements of this mutual procedure subdomain. This is done by employing standard proof techniques to demonstrate that the symbolic representation of the procedure subdomain implies that the computation difference is zero.

The property of equivalence states that the implementation and specification have the same domain of definition, which implies that compatibility holds. After all, it does not make much sense to say that an implementation is equivalent with a specification

if it has completely different parameters or is defined over a different domain. Yet at times, it is virtually impossible to achieve compatibility with the specification when the procedure must be implemented in a language lacking certain high-level constructs (e.g., FORTRAN). As mentioned previously, however, the premises underlying equivalence are valid in the context of certain restricted violations of compatibility, in particular, those that do not inhibit the construction of the procedure partition. Although incompatibility implies that the equivalence property is not satisfied, it is usually worthwhile to continue with the determination of equivalence for the compatible procedure subdomains.

To determine whether the equivalence property prevails, the equality of the subspec computation and the path computation over each compatible procedure subdomain must be determined. For procedure subdomain D_{IJ}, the equality of $C[S_I]$ and $C[P_J]$ is determined by demonstrating whether or not the associated computation difference C_{IJ} is zero when it is restricted to the procedure subdomain. In general, the computation difference C_{IJ} is a vector corresponding to the output vector $z = (z_1, z_2, \ldots, z_n)$. Thus, each component $C_{IJ}.z_i$ must be considered and $C_{IJ}|D_{IJ} = 0$ if and only if $C_{IJ}.z_i|D_{IJ} = 0$ for $1 \leq i \leq n$. In many cases, the simplification of the computation difference $C_{IJ}.z_i$ reduces that expression to zero, in which case the two computations $C[S_I].z_i$ and $C[P_J].z_i$ are symbolically identical and thus equal over any domain. The simplification of $C_{IJ}.z_i$ is synonomous to a term-by-term comparison of the symbolic values of $C[S_I], z_i$ and $C[P_J].z_i$. Two computations $C[S_I].z_i$ and $C[P_J].z_i$ are also equal over the associated procedure subdomain D_{IJ} if the condition defining D_{IJ} implies that the computation difference $C_{IJ}.z_i$ is zero. Proving

that $D_{IJ} \Rightarrow C_{IJ}|D_{IJ} = 0$ can be done through the application of proof techniqus such as those employed by automatic program verifiers.

The process of applying partition analysis verification to the procedure *PRIME* is somewhat complicated, primarily due to the properties that are used in producing the efficient implementation. Only the proof developed by partition analysis verification for procedure subdomain D_{21} is discussed here, and shown in Figure 7, because it illustrates a way in which the verification process guides in the test data selection process. In proving that D_{21} implies that $C_{21} = 0$, partition analysis verification notes that the value of the computation difference C_{21} will clearly vary depending on whether $(n < 4)$ or $(n \geq 4)$. This prompts the further division of the procedure subdomain into two subsets — D_{21a}, which contains those elements in D_{21} for which $(n < 4)$, and D_{21b}, which contains those elements in D_{21} for which $(n \geq 4)$. The proof is then done in two parts, $D_{21a} \Rightarrow C_{21} = 0$ and $D_{21b} \Rightarrow C_{21} = 0$. The condition defining D_{21a} implies that the sequence $[2..n-1]$ contains only the element 2 (this fact is denoted 21a-1 in the proof) and also that $(trunc(n/2) * 2 - n \neq 0)$ (21a-2). These two facts imply that $forall\{i := 2..n-1 \mid (trunc(n/i) * i - n \neq 0)\}$ is true (21a-3). The condition defining D_{21a} also implies $(n < 4)$ (21a-4). The facts denoted by (21a-3) and (21a-4) imply that $C_{21} = 0$. A similar proof is generated to show that $D_{21b} \Rightarrow C_{21} = 0$. The two proofs serve to demonstrate that $D_{21} \Rightarrow C_{21} = 0$. Because this proof was contingent on the further division of the procedure subdomain, it is clear that the differences between the subspec and path computations vary between these subsets of the procedure subdomain. It is thus important to test elements in both subsets of the

D_{21}: $(n \geq 3)$ and $((trunc(n/2) * 2 - n = 0)$ or $(trunc(n/3) * 3 - n = 0))$
C_{21}: $(\text{forall}\{i := 2..n-1 \mid (trunc(n/i) * i - n \neq 0)\}) - (n < 4)$

D_{21}: $\{D_{21} \mid (n < 4)\} \cup \{D_{21} \mid (n \geq 4)\}$
D_{21a}: $(n \geq 3)$ and $((trunc(n/2) * 2 - n = 0)$ or $(trunc(n/3) * 3 - n = 0))$ and $(n < 4)$
$= (n = 3)$ and $((trunc(n/2) * 2 - n = 0)$ or $(trunc(n/3) * 3 - n = 0))$
D_{21b}: $(n \geq 3)$ and $((trunc(n/2) * 2 - n = 0)$ or $(trunc(n/3) * 3 - n = 0))$ and $(n \geq 4)$
$= (n \geq 4)$ and $((trunc(n/2) * 2 - n = 0)$ or $(trunc(n/3) * 3 - n = 0))$

Proof of $D_{21a} \Rightarrow C_{21} = 0$:

$(n = 3) \Rightarrow (2..n-1 = 2)$	(21a-1)
$(n = 3) \Rightarrow (trunc(n/2) * 2 - n \neq 0)$	(21a-2)
(21a-1) and (21a-2) $\Rightarrow \text{forall}\{i := 2..n-1 \mid (trunc(n/i) * i - n \neq 0)\}$	(21a-3)
$(n = 3) \Rightarrow (n < 4)$	(21a-4)
(21a-3) and (21a-4) $\Rightarrow D_{21a} \Rightarrow C_{21} = (true) - (true) = 0$	(21a-5)

Proof of $D_{21b} \Rightarrow C_{21} = 0$:

$(n \geq 4) \Rightarrow (2 < n-1)$ and $(3 \leq n-1)$	(21b-1)
(21b-1) and $((trunc(n/2) * 2 - n = 0)$ or $(trunc(n/3) * 3 - n = 0)) \Rightarrow$ exists$\{i := 2..n-1 \mid (trunc(n/i) * i - n = 0)\} \Rightarrow$ not forall$\{i := 2..n-1 \mid (trunc(n/i) * i - n \neq 0)\}$	(21b-2)
$(n \geq 4) \Rightarrow$ not $(n < 4)$	(21b-3)
(21b-2) and (21b-3) $\Rightarrow D_{21} \Rightarrow C_{21b} = (false) - (false) = 0$	(21b-4)

Proof of $D_{21} \Rightarrow C_{21} = 0$:
(21a-5) and (21b-4) $\Rightarrow$
$D_{21} \Rightarrow C_{21} = 0$

Figure 7: Partition Analysis Verification of PRIME (Procedure Subdomain D_{21})

procedure subdomain. In general, whenever partition analysis verification must divide a procedure subdomain into subsets and prove that the computation difference is zero over each subset independently, partition analysis testing is directed to select test data from each such subset of the procedure subdomain.

Partition analysis verification is a variation on symbolic testing [HOWD77] in which the symbolic representations of the domains and computations derived from a program by symbolic evaluation are examined. Partition analysis verification, however, compares these representations with those derived from the specification.

Partition analysis verification uses standard proof techniques to determine the equality of computations over a domain. In general, the problem of equivalence is undecidable and thus partition analysis verification suffers one of the same drawbacks as other verification approaches to demonstrating program reliability. Traditional verification approaches decompose the implementaton into sequences of statements and employ proof techniques to show that assertions are true at points between these sequences. By so doing, failure to prove a single assertion may cause failure to show that any of the implementation is correct. When partition analysis verification fails to prove the equality or inequality of the associated computations for a procedure subdomain, it does not affect the proofs for other procedure subdomains.

In the absence of a proof of equivalence, counterexamples can sometimes be found to demonstrate that the subspec computation and the path computation are not equal over the procedure subdomain. Partition analysis verification attempts to find some element in

the procedure subdomain for which the computation difference is nonzero. For procedure subdomain D_{IJ}, this is done by simplifying the computation difference C_{IJ} and attempting to find a solution to the conjunction (D_{IJ} and ($C_{IJ} \neq 0$)). If this conjunction is satisfiable, then input data exists in D_{IJ} for which C_{IJ} is not zero. Thus, $C_{IJ}|D_{IJ} \neq 0$ and the subspec computation $C[S_I]$ and the path computation $C[P_J]$ are not equal when restricted to D_{IJ}. When no counterexample can be found, partition analysis testing often succeeds in detecting errors thereby pointing out counterexamples, otherwise it provides assurance of the equality of the computations.

4. PARTITION ANALYSIS TESTING

Since partition analysis verification works with a simplified symbolic representation of the differences between a subspec and a path, it is very important to test that the conclusions drawn in this postulated environment are also valid in the natural run-time environment. To demonstrate the run-time behavior, it is important to select test data for which the paths are sensitive to error. Further, there is always a chance (a very good one at that) that a subspec is incorrect, thus test data for which each subspec is sensitive to error are also selected. Selecting sensitive test data for both may draw attention to an error that might otherwise remain undetected.

Within partition analysis, the symbolic representations of a procedure subdomain and the associated computations are employed to direct the selection of test data for the subdomain. Partition analysis testing thereby draws on information describing both the intended

and actual function of the procedure. To increase the likelihood of detecting errors, sophisticated test data selection criteria are employed for each procedure subdomain.

The testing literature has classified program errors into two categories: *computation errors* and *domain errors*, according to whether the effect is an incorrect path computation or an incorrect path domain. A domain error may be either a *missing path error*, which occurs when a special case requires a unique sequence of actions but the program does not contain a corresponding path, or a *path selection error*, which occurs when a program recognizes the need for a path but incorrectly determines the conditions under which that path is executed. A computation error occurs when the correct path through the program is taken, but the output is incorrect because of faults in the computation along the path.

Error-sensitive test data selection criteria are often geared toward the detection of either computation errors or domain errors. Many of these criteria have consisted of intuitive guidelines [FOST80,HOWD80,HOWD81,MYER79,REDW83,WEYU81], but several have been made more rigorous, often by incorporating the symbolic representations created by symbolic evaluation[CLAR85a]. These more formalized strategies include *computation testing* [CLAR83,HOWD78,HOWD80] and *domain testing* [CLAR82,WHIT80], which under certain conditions guarantee the absence of computation errors and domain errors, respectively. These formalized strategies have been extended to be applicable in the context of procedure subdomains [RICH81b].

Computation testing is based on the assumption that the way an input value is used within the subspec and path computations is indicative of a class of potential computation

errors. Analysis of the symbolic representation of the computation difference reveals the manipulations of the input values that have been performed to compute the output values. For each procedure subdomain, data are selected 1) to demonstrate whether or not the subspec computation and the path computation are equal over the procedure subdomain, and 2) to characterize the computations in terms of potential errors and run-time behavior.

Partition analysis testing is sometimes able to demonstrate that the subspec computation and the path computation are equal over the procedure subdomain. When the procedure subdomain D_{IJ} is small and discrete, each element in D_{IJ} can be selected as test data. This amounts to exhaustive testing for this procedure subdomain and thus is not feasible for nondiscrete subdomains or those with many elements.

Another more interesting case is when a computation is a polynomial; then polynomial testing techniques can be applied to demonstrate equality. In this case, the computation difference is a polynomial and the number of test points that must be selected to guarantee it is zero is dependent on algebraic properties of the polynomial. For instance, if the computation difference C_{IJ} is a univariate polynomial with integer coefficients whose magnitudes do not exceed a known bound, a single test point can be found to demonstrate that $C_{IJ} \mid D_{IJ} = 0$ [ROWL81]. Alternately, if C_{IJ} is a univariate polynomial of maximal degree t, the selection of $t + 1$ elements is sufficient to determine whether $C_{IJ}|D_{IJ} = 0$ [HOWD78]; this holds if $C_{IJ} = 0$ for the $t + 1$ elements. If C_{IJ} is a multivariate polynomial in k variables (where a variable in this context is the symbolic name for an input value) of maximal degree t, C_{IJ} must be zero for $O(t^k)$ elements in order to determine

that $C_{IJ}|D_{IJ} = 0$ [HOWD78]. Probablistic arguments have been made for limiting the number of elements that must be selected without sacrificing too much in the way of accurate results [DEMI78a]. Selecting test data to demonstrate that $C_{IJ}|D_{IJ} = 0$ is a useful feature to employ when partition analysis verification is not able to determine whether $D_{IJ} \Rightarrow C_{IJ} = 0$.

The computations are also analyzed in an attempt to characterize potential errors and run-time behavior. In general, a path computation may contain arithmetic manipulations or data manipulations, which are inherently sensitive to different classes of computation errors. Computations containing predominately arithmetic manipulations are sensitive to errors relating to the use of numeric values and operators in arithmetic expressions. Test data is selected for which each symbolic name in the computation difference C_{IJ} take on distinct values and for which multipliers, divisors, exponents, terms, and repetition counts take on special values and extremal [2] and nonextremal values. Test data is also selected to force the specification and implementation computations to take on special values and extremal and nonextremal values. Computations containing data manipulation typically maintain compound data structures and as a result are sensitive to errors that involve data movement operations rather than arithmetic operations. Test data is selected for which component selectors and components take on both distinct values and identical values, as well as extremal and nonextremal values. Test data is selected for which the size of a compound structure is extremal and nonextremal and for which a compound structure is

[2] A value of large magnitude often serves the purpose of an extremal value.

both full and empty. The guidelines that are applied within partition analysis testing for computations containing either arithmetic or data manipulations are more fully described elsewhere [CLAR83].

It is important to note that the guidelines may not all be satisfiable due to the condition defining D_{IJ} or the representation of C_{IJ}. In addition, not all of the guidelines need be satisfied independently, so the amount of test data is not as large as it might appear. For example, test data to characterize errors may satisfy the polynomial testing requirements. The selection of computationally-sensitive test data often results in erroneous or inaccurate output. If no such errors are uncovered, assurance in the correctness of the computations has been provided.

Domain testing, as traditionally described, is a path testing strategy that concentrates on detecting path selection errors. These strategies advocate the selection of test points near path domain boundaries. The boundary of a path domain is composed of borders with adjacent path domains. Each border results from a relational expression in a branch predicate constraint. For each closed border, the strategy selects *on* test points, which lie on the border and thus in the path domain being tested, and *off* test points, which lie on the open side of the border and thus in an adjacent path domain. (An open border of a path domain is tested when testing the adjacent domain.) A border shift can remain undetected only if the on test points and the off test points lie on opposite sides of the correct border. The undetectable border shifts are kept "small" by choosing the off test points as close to the border being tested as possible. A thorough description of domain testing, including

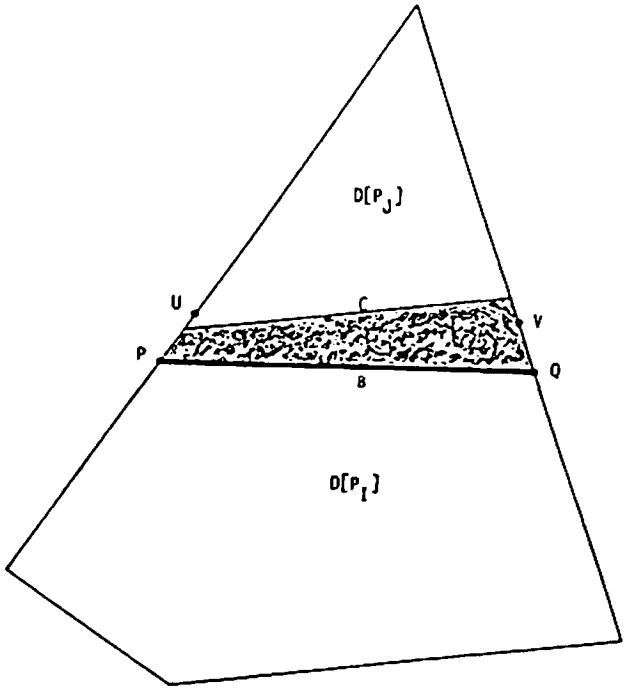

Figure 8: Domain Testing

the number of test points that should be selected to minimize the potential undetected border shifts and its effectiveness in detecting path domain errors is provided in [CLAR82]. Figure 8 illustrates a border shift where B is the border being tested and C is the correct border. The border shift is detected by the selection of on points P and Q and off points U and V, since V is in the wrong domain. The existence of a border shift implies that a path selection error has occured, and thus path selection errors are likely to be caught by domain testing criteria.

While domain testing directly addresses path selection errors, missing path errors will not be detected unless the missing path's domain is close to a domain border. Suppose, for instance, that M and N represent missing borders and thus a missing path error exists, as shown in Figure 9. Domain testing, as described, will not necessarily force selection

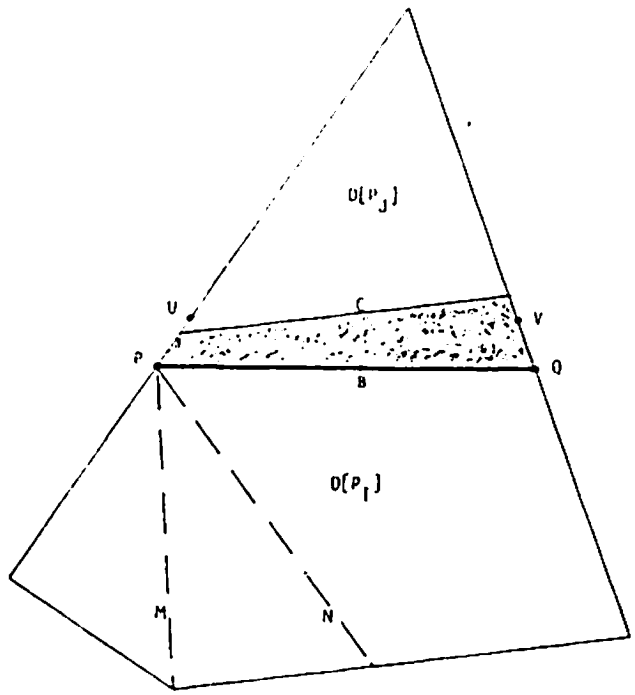

Figure 9: Missing Path Error

of data within the missing subdomain. The error will go undetected unless other testing criteria (such as the computation testing criteria) cause such data to be selected. In fact, missing path errors cannot be found systematically unless a specification is employed by the test data selection strategy, as is done by partition analysis testing.

The application of domain testing based on the procedure partition involves analyzing the procedure subdomain and selecting test data on and slightly off the procedure subdomain boundaries. In partition analysis testing, the correct borders are essentially those in the specification partition — that is, the borders of the subspec domains. As discussed previously, however, the implementation may be correct, even though the implementation and specification partitions are different. Any border of a procedure subdomain is either part of a path domain boundary or part of a subspec domain boundary or coincides in

a path domain boundary and a subspec domain boundary. It is important to test each type of border. A border that coincides in a path domain boundary and a subspec domain boundary is tested to substantiate the correctness of the subspec domain, and hence the path domain as well. A procedure subdomain border that corresponds only to a path domain boundary separates two path domains, say $D[P_J]$ and $D[P_K]$, that both have elements in one subspec domain. Perhaps the implementation has split a subspec domain into two path domains so as to perform a more efficient computation for some of the elements. Thus, the path computations $C[P_J]$ and $C[P_K]$ are likely to be different. By testing the border with on and off test points, it can be determined whether the border separates the path domains correctly or if a border shift, and thus a path selection error, has occurred. A procedure subdomain border that corresponds only to a subspec domain boundary, separates two subspec domains, say $D[S_I]$ and $D[S_H]$, that both have elements in one path domain. In this case, the subspec computation $C[S_I]$ and $C[S_H]$ are probably different, yet the implementation has grouped two subspec domains together. This may have been done for efficiency or may indicate a missing path error. Testing this border with both on and off test points helps determine whether the implementation computation has adequately captured both subspec computations. In Figure 9, the borders M and N are part of a procedure subdomain boundary resulting from a subspec domain. Partition analysis testing will force the selection of test points on and near the border M and N, and thus the corresponding missing path error will be detected. By testing the boundary of each procedure subdomain, both the similarities and differences between the subspec

domains and the path domains are tested.

Domain testing is a relatively new test data selection strategy for which much further research is needed. The strategy has been well defined for domains that are continuous, linear convex polyhedra. This assumes that the input space is continuous, that none of the interpreted branch predicates contain a disjunction, and that all relational expressions are linear. Adequate modifications have been proposed for both nonconvex and discrete domains, although several problems remain to be addressed [CLAR82,WHIT80]. Modifications have been proposed that require the selection of on and off test points near each of the local minima and maxima of a nonlinear border. Unfortunately, the practical applicability of domain testing is limited to interpreted branch predicates of low degree. Even in these cases, complications arise when branch predicates contain compound data structures and their component selectors depend on input values. Due to the dependencies among components of a compound structure and the component selectors, it may not be possible to find good on and off test points for a particular border. In such cases, the intuitive concepts underlying domain testing can be used as heuristics to test the borders of a path domain.

Figure 10 shows the test data selected for procedure subdomain D_{21} and computation difference C_{21} of *PRIME*. For each test datum selected, the reason for its selection is noted. Note that since partition analysis verification divided procedure subdomain D_{21} into subsets to prove that the associated computations are equal, partition analysis testing applies domain and computation testing to select test data from both subsets

D_{21}: $(n \geq 3)$ and $((trunc(n/2) * 2 - n = 0)$ or $(trunc(n/3) * 3 - n = 0))$
C_{21}: $(\text{forall}\{i := 2..n-1 \mid (trunc(n/i) * i - n \neq 0)\}) - (n < 4)$

Domain Testing Criteria:

N = 3 $\Rightarrow$ on $(n = 3)$ of D_{21a}, off $(n \geq 4)$ of D_{21b}
N = 2 $\Rightarrow$ off $(n = 3)$ of D_{21a}
N = 4 $\Rightarrow$ on $(n \geq 4)$ of D_{21b}, on $(trunc(n/2) * 2 - n = 0)$
N = 5 $\Rightarrow$ off $(trunc(n/2) * 2 - n = 0)$, off $(trunc(n/3) * 3 - n = 0)$
N = 6 $\Rightarrow$ on $(trunc(n/2) * 2 - n = 0)$, on $(trunc(n/3) * 3 - n = 0)$
N = 7 $\Rightarrow$ off $(trunc(n/2) * 2 - n = 0)$, off $(trunc(n/3) * 3 - n = 0)$
N = 9 $\Rightarrow$ on $(trunc(n/3) * 3 - n = 0)$
N = 1000 $\Rightarrow$ on $(trunc(n/2) * 2 - n = 0)$, off $(trunc(n/3) * 3 - n = 0)$
N = 999 $\Rightarrow$ on $(trunc(n/3) * 3 - n = 0)$, off $(trunc(n/2) * 2 - n = 0)$

Computation Testing Criteria:

N = 3 $\Rightarrow$ $C[S_3] = C[P_2] = true$, forall iterated minimum times
N = 4 $\Rightarrow$ $C[S_3] = C[P_2] = false$
N = 1000 $\Rightarrow$ forall iterated maximum times

Figure 10: Partition Analysis Testing of PRIME (Procedure Subdomain D_{21})

D_{21a} and D_{21b}. The domain testing strategy proposes the selection of boundary points of the procedure subdomain. Thus, test data are selected on and slightly off the borders of D_{21a} and D_{21b}. Due to the simplicity of the computations, most of the guidelines for computation testing are trivial to apply. For this example, computation testing requires that test data be selected for which both true and false values of $PRIME$ result and for which the iteration involved in the forall construct be done a minimum (once) and a maximum (1000 is assumed to be the maximum) number of times. The data selected by partition analysis to completely test $PRIME$ are shown in Figure 11.

Combining the domain and computation testing strategies on the basis of the procedure partition results in the selection of data that more rigorously test a procedure than other proposed testing strategies. Most programmers would admit that the test set for $PRIME$ is a more comprehensive set than they would have selected, yet upon examination it is clear that each datum is selected to test a particular feature of the implementation or the specification of $PRIME$.

The domain and computation testing criteria used within partition analysis testing subsume most proposed intuitive guidelines. The computation testing criteria subsume the special value testing and extremal output value testing proposed by Howden [HOWD80,HOWD81], as well as the engineering approach proposed by Redwine [REDW83]. Domain testing subsumes both the boundary value testing and condition coverage guidelines proposed by Myers [MYER79] and the extremal input value testing proposed by Howden [HOWD80]. The combination of computation and domain test-

D_{11} : Domain Testing Criteria:
$N = 1, N = 2, N = 3$
Computation Testing Criteria:
$N = 2$

D_{21} : Domain Testing Criteria:
$N = 2, N = 3, N = 4, N = 5, N = 6, N = 7, N = 9, N = 999, N = 1000$
Computation Testing Criteria:
$N = 3, N = 4, N = 1000$

D_{22} : Domain Testing Criteria:
$N = 2, N = 3$
Computation Testing Criteria:
$N = 5, N = 23$

D_{23} : Domain Testing Criteria:
$N = 24, N = 25, N = 26, N = 49, N = 50, N = 973, N = 995$
Computation Testing Criteria:
$N = 25, N = 995$

D_{24} : Domain Testing Criteria:
$N = 24, N = 27$
Computation Testing Criteria:
$N = 27, N = 113$

D_{25} : Domain Testing Criteria:
$N = 120, N = 121, N = 122, N = 168, N = 169,$
$N = 170, N = 288, N = 289, N = 961, N = 989$
Computation Testing Criteria:
$N = 121, N = 989$

D_{26} : Domain Testing Criteria:
$N = 120, N = 127, N = 288, N = 293$
Computation Testing Criteria:
$N = 127, N = 997$

Figure 11: Partition Analysis Testing of PRIME

ing covers Foster's Error-Sensitive Test Case Analysis [FOST80,FOST83,FOST84] and Weyuker's error-guessing technique [WEYU81]. Moreover, the computation and domain testing criteria are integrated within partition analysis testing so as to exploit the overlap among criteria in an attempt to reduce the total number of test points.

The testing and verification processes are integrated within partition analysis so that they might complement and enhance one another. Partition analysis testing not only substantiates the verification process, but may in fact assist in that process. Partition analysis verification often provides insight into the test data selection process as well. If partition analysis verification disproves equivalence by detecting counterexamples, these values are selected as test data. When partition analysis verification is unable to complete a proof, partition analysis testing attempts to complete the task of determining computation equality. Although the verification process did not succeed in proving or disproving computation equality, it may have reached some conclusion about elements of the procedure subdomain that are crucial to the prevalence of computation equality. In this case, the test data set is augmented with such elements. Then, partition analysis testing either shows these as counterexamples or provides assurance in the computation equality. Another situation in which partition analysis verification directs the selection of test data occurs when the proof of computation equality is contingent on the further division of the procedure subdomain. This further decomposition implies that it is important to test elements in each subset of the procedure subdomain and partition analysis testing is directed accordingly. Thus in partition analysis, the verification and testing are complementary techniques that are

employed to enhance each other, thereby providing a method that is stronger than either technique alone.

5. AN EVALUATION OF PARTITION ANALYSIS

An evaluation of the partition analysis method was undertaken that involved applying partition analysis to thirty-four procedures. These procedures were taken from the program testing and verification literature and from several programming textbooks. The major difficulty posed by this choice was the unavailability of accompanying formal specifications. We often had to write specifications for these procedures based on the implementation and the English descriptions of their intended function. In the first part of this evaluation, partition analysis was considered to be successful if it detected all the errors in an erroneous implementation or if it demonstrated consistency between a correct implementation and its specification. Since many of the programs were correct or contained only a few, well-documented errors, we used mutation analysis [DEMI78b] to systematically seed large numbers of errors into the implementation. For the second part of the evaluation, partition analysis testing was considered successful if it detected all these seeded errors. The results from both parts of this evaluation are discussed in this section and summarized in Figure 12.

For the first part of this evaluation, eleven erroneous programs and program fragments were selected from the "Common Blunders" section of *The Elements of Programming Style* [KERN74]. Partition analysis detected all of the blunders in procedures derived

PROCEDURE	ERRORS	PARTITION ANALYSIS	MUTANTS	KILLED
SIN	3 *ce*, 1 *pse*	errors detected	266	249
CURRENT	2 *ce*, 1 *pse*, 1 *mpe*	errors detected	384	362
SUM	1 *ce*	errors detected	82	76
ACCOUNT	1 *pse*	error detected	207	193
FIRSTMIN	2 *pse*	errors detected	87	85
LOAN	1 *ce*, 1 *pse*	errors detected	256	221
BINSEARCH	4 *ce*, 1 *pse*, 2 *mpe*	errors detected	292	273
GRADES	1 *ce*	error detected	133	121
MEANVAR	1 *ce*	error detected	130	127
TRAP	1 *ce*	error detected	337	325
RIGHTTRI	1 *pse*	error detected	272	227
KING1	correct	consistency demonstrated	121	102
KING2	correct	consistency demonstrated	186	167
KING3	correct	loop analysis failed	93	85
KING4	correct	consistency demonstrated	171	157
KING5	correct	consistency demonstrated	210	203
KING6	correct	loop analysis failed	156	139
KING7	correct	loop analysis failed	149	132
KING8	correct	consistency demonstrated	198	177
FIND	correct	loop analysis failed	525	503
BUGGYFIND	1 *ce*	error detected	513	491
CALENDAR	1 *pse*	error detected	99	87
TRIANG	1 *mpe*	error detected	353	315
LADERMAN	correct	consistency demonstrated	3991	3991
PRIME	correct	consistency demonstrated	363	295
INTEGRAL	correct	consistency demonstrated	647	607
PAL	correct	consistency demonstrated	183	161
HORNERS	correct	consistency demonstrated	162	154
COSINE	correct	consistency demonstrated	129	101
TRANSACT	correct	consistency demonstrated	226	219
DOCKING	correct	consistency demonstrated	307	289
QUAD	correct	consistency demonstrated	293	286
NEARP	correct	consistency demonstrated	321	309
BISECTION	correct	consistency demonstrated	592	569

NOTE: *ce* stands for computation error,
pse stands for path selection error,
mpe stands for missing path error.

Figure 12: Evaluation of Partition Analysis

from these programs and program fragments. For the binary search procedure, the loop expression for the implementation could not be created and thus the procedure partition could not be constructed. A partial application of partition analysis, in which symbolic representations are created for paths that perform zero, one, and two iterations of the loop, is capable of detecting all of the errors in this implementation. This partial application of partition analysis is often helpful when the loop analysis technique fails. The eleven procedures contained a total of fourteen computation errors and eleven domain errors (eight path selection errors and three missing path errors). Symbolic evaluation was all that was needed to reveal eight of the errors. Two of the errors are precision errors; these errors were not reflected in the implementation partition and thus were not detected by partition analysis verification, but were detected by partition analysis testing. The other fifteen errors were initially detected by partition analysis verification and also were detected by partition analysis testing. This group of procedures is of special interest since Howden used the same procedures to evaluate two testing methods [HOWD76,HOWD77] — symbolic testing and an approximation to path analysis testing. In Howden's study, symbolic testing was found to be somewhat more effective than path analysis testing, and a combination of the two was conjectured to be more effective than either method used alone. For this set of procedures, the partition analysis method performs better than either of these methods or their combination. This is not surprising, however, since our integrated testing strategies include a more formal version of symbolic testing and path analysis. The methods evaluated by Howden were ineffective for domain errors, but the

partition analysis method, which includes a domain testing strategy and considers both the specification and the implementation, was effective at detecting these errors.

In addition to the "Common Blunders" procedures, nine correct procedures were taken from the verification literature [DEUT73,HOAR71,KING69]. Partition analysis demonstrated the reliability of five of the nine. The failure of the method on the others was due to loop structures that are too complex for the loop analysis technique. The FIND procedure [HOAR71], for example, has four loops; the three inner loops can be represented by a loop expression but the loop analysis technique fails on the outer-most loop. As noted, partition analysis can be partially applied in an attempt to detect errors in the implementation, but consistency cannot be demonstrated. Hence, since each of these four procedures is correct, the partition analysis method failed to come to a conclusion, although no errors were found in the partial application, thus providing some assurance of the reliability of the implementation.

In addition, four procedures from the literature on program testing have been used to evaluate partition analysis. One such program is BUGGYFIND [BOYE75,DEMI78c], which is an erroneous version of the FIND program discussed above. BUGGYFIND contains one computation error, which has been repeatedly demonstrated as very difficult to detect. Partition analysis testing does reveal the existence of this error. The CALENDAR procedure has been studied by several authors [GELL78,WEYU80] and has undergone the successful application of partition analysis. This procedure contains one domain error that is detected by both the verification and testing processes of partition analysis. TRI-

ANG [RAMA76,DEMI78c,WEYU80] contains one missing path error, which is detected by partition analysis verification and testing. LADERMAN [RICH78], which was used to evaluate mutation analysis, contains no errors and was shown to be correct.

The final group of procedures to which partition analysis was applied, were ten procedures found in introductory programming and program development texts [GEAR78,GROG79,KOFF81,WEGN80,WELS79]. Most of these procedures were accompanied by relatively precise, although not formal, specifications that were easily written in the SPA language. The *PRIME* example that appears in this paper is one such example. These procedures contained no errors and the partition analysis method was successful in demonstrating consistency between the specification and implementation.

Although it provided interesting feedback, the first part of our evaluation had some limitations. First, the application of the verification and test data selection techniques was done manually. While efforts were made to remain objective, there is always some doubt as to whether the errors that have been detected would also have been detected if they were not known to exist. Second, the effectiveness of partition analysis testing is unclear when the implementation is correct. (Likewise, just because all known errors were detected does not mean that all errors were detected.) The method's effectiveness must be evaluated by some other effectiveness measure in addition to the evaluation in terms of its ability to detect a few known errors.

To address both of these problems, we undertook the second part of this evaluation in which we used mutation analysis [DEMI78b] to measure the adequacy of the selected test

data. Mutation analysis is based on the assumption that a program is either correct or almost correct (differs from a correct program by some simple error). Mutation analysis seeds a large number of simple errors into each statement in the implementation to produce "mutants". The original and mutant programs are executed on the selected set of test data. If a mutant program gives correct results on all of the test data, it is said to be live, otherwise it has been killed. If all mutants are killed, then the test data is adequate since it distinguishes the original program from all those containing simple errors. For any live mutant either the mutant is equivalent to the original program (no problem), the implementation is incorrect (an error is detected), or the test data set is inadequate.

Using the test data from partition analysis testing, the mutation analysis system demonstrated the effectiveness of the method. For each program analyzed, all of the nonequivalent mutants were killed. For the program *PRIME*, for instance, the mutation analysis system produced 363 mutant programs from the original. Of these, 295 gave incorrect results on the data selected by partition analysis. All of the remaining mutants were equivalent to the original program. Thus the test data set selected by partition analysis for *PRIME* is adequate.

One limitation of the second part of this study is that all errors introduced by mutation analysis are either computation errors or path selection errors. The structure of the program is not mutated, and hence no missing path errors are introduced. Thus checking the adequacy of partition analysis in this way does not exercise one of its most unique features.

6. CONCLUSION

Partition analysis attempts to integrate testing and verification. Our evaluation of the method demonstrates that it can be very effective at detecting errors. When no errors are detected in a program, it provides a reasonable level of assurance in the reliability of that program. This section describes some of the disadvantages and limitations of the method and of our evaluation and discusses some areas of current and future research.

Partition analysis relies on the development of procedure subdomains, which partition the set of input data based on the implementation and the specification. Procedure subdomains appear to be the largest units of input data that can be analyzed independently; yet they provide a practical decomposition of the testing and verification process. Further, the procedure partition appears to overcome many of the problems encountered with alternative decompositions that are suggested by other verification and testing methods [FLOY67,GELL78,HOWD76,LOND75,WEYU80]. There are several problems encountered, however, in forming the procedure partition. Determining constraint consistency, creating closed-form expressions for loops, and finding the nonempty intersection between subdomains are all unsolvable problems in general. In practice, formulating a closed-form expression for loops seems to be the major stumbling block. While closed-form expressions were found in most of the examples we examined, the insufficiency of the loop analysis technique was still the major cause of failure. We are currently investigating more powerful loop analysis techniques. Since no technique will always be successful, we are also reexamining the heuristics we use when loop analysis fails.

Partition analysis verification addresses some of the limitations of other verification methods. Although this paper has described the method for a design specification and implementation, it can be applied to any two descriptions and thus can be used throughout the software development process. Some verification methods have also been proposed for use throughout software development [SILV79,GRIE76], but these have been restricted to very particular languages. Because partition analysis is applicable to different kinds of languages (forming the procedure partition is actually a translation to a common functional representation), it can more naturally be applied with actual software development efforts. In particular, PDL-type languages appear to be the most widely used type of pre-implementation specification, yet other verification methods are not applicable with procedural languages of this sort. Despite these advantages, partition analysis verification has some of the same drawbacks as other verification techniques. In general, it can not be proven that two descriptions are equivalent. Also verification is usually based upon assumptions about a postulated environment; the proof or the assumptions could be wrong [GERH76,DEMI79]. It is for these reasons that partition analysis proposes that verification be complemented by testing.

Partition analysis testing has been shown to be a powerful testing strategy. The reasons for this are two-fold. First, it *integrates* several complementary testing strategies. Second, the selected test data appropriately *characterize* the procedure based on *both* the implementation and the specification. As such, it is one of the few testing strategies to address missing path errors. There are several problems with partition analysis testing that must

be addressed. In particular, the error-sensitive testing proposed for partition analysis may result in an excessive number of test points. It is important to note, however, that many of the test data satisfy more than one selection criteria. This overlap occurs within both the domain and computation strategies as well as between the two. Thus, various testing strategies must be more tightly integrated so as to exploit the overlap among criteria in an attempt to reduce the total number of selected test points. Before this can be accomplished, however, more work must be done to understand the strengths and weaknesses of these techniques and how they relate. We have already started to address some of these issues [CLAR85b] and intend to pursue this work further. In addition to developing better integration schemes, it is also necessary that the testing analysis be automated. Most of the testing strategies being used are difficult to apply and, although it can be shown how they relate to intuitive guidelines, they are not necessarily intuitive to apply. Finally, we are also considering using partition analysis testing to direct the testing of specifications. Even specifications for single modules can be difficult to write or understand, so designing specification languages that are amenable to testing and developing appropriate testing methods is an area of current research [KEMM85,BALZ81,GOGU79].

The evaluation that we did of partition analysis demonstrated some of the strengths and weaknesses of the method. The evaluation was of a limited scale, however, due to the fact that we were applying the method manually and to the unavailability of program and specification pairs. The method is intended to be applied to single modules so the size of the evaluated modules would not be significantly smaller than a "real world" sample.

It would be interesting and worthwhile, however, to apply the method to the modules of somewhat larger programs to see what problems are encountered and to evaluate how the method scales up.

REFERENCES

[ABRA79] P. Abrahams and L.A. Clarke, "Compile-Time Analysis of Data List - Format List Correspondences," *IEEE Transactions on Software Engineering*, SE-5, 6, November 1979, 612–617.

[BALZ81] R.M. Balzer, "Final Report," Information Sciences Institute, University of Southern California, February 1981.

[BOYE75] R.S. Boyer, B. Elspas, and K.N. Levitt, "SELECT — A Formal System for Testing and Debugging Programs by Symbolic Execution," *Proceedings of the International Conference on Reliable Software*, April 1975, 234–244.

[CHEA79] T.E. Cheatham, G.H. Holloway, and J.A. Townley, "Symbolic Evaluation and the Analysis of Programs," *IEEE Transactions on Software Engineering*, SE-5, 4, July 1979, 402–417.

[CLAR81] L.A. Clarke and D.J. Richardson, "Symbolic Evaluation Methods — Implementations and Applications," *Computer Program Testing*, editors B. Chandrasekaran and S. Radicchi, North Holland Publishing Co., 1981, 65–102.

[CLAR82] L.A. Clarke, J. Hassell, and D.J. Richardson, "A Close Look at Domain Testing," *IEEE Transactions on Software Engineering*, July 1982, 380–390.

[CLAR83] L.A. Clarke and D.J. Richardson, "A Rigorous Approach To Error-Sensitive Testing," *Sixteenth Annual Hawaii Conference on System Sciences*, January 1983, 197–206.

[CLAR85a] L.A. Clarke and D.J. Richardson, "Applications of Symbolic Evaluation," *Journal of Systems and Software*, Vol.5, No.1, January 1985.

[CLAR85b] L.A. Clarke, A. Podgurski, and D.J. Richardson, "A Comparison of Data Flow Path Selection Criteria," Software Development Laboratory, Department of Computer and Information Science, University of Massachusetts.

[DEMI78a] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *IEEE Computer*, 11, 4, April 1978, 34–41.

[DEMI78b] R.A. DeMillo, R.J. Lipton, and 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.

[DEMI78c] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, 11,4, April 1978, 34–41.

[DEMI79] R.A. DeMillo, R.J. Lipton, and A.J. Perlis, "Social Processes and Proofs of Theorems," *Communications of the ACM*, 22, 5, May 1979, 271–280.

[DEUT73] L.P. Deutsch, "An Interactive Program Verifier," Ph.D. Dissertation, University of California, Berkeley, May 1973.

[DIJK70] E.W. Dijkstra, "Structured Programming," *Software Engineering Principles*, editors J.N. Buxton and B. Randall, Brussels, Belgium, NATO Science Committee, 1970.

[FLOY67] R.W. Floyd, "Assigning Meaning to Programs," *Proceedings of a Symposium in Applied Mathematics*, 19, American Mathematical Society, 1967, 19–32.

[FOST80] K.A. Foster, "Error Sensitive Test Case Analysis (ESTCA)," *IEEE Transactions on Software Engineering*, SE-6, 3, May 1980, 258–264.

[FOST83] K.A. Foster, "Comment on The Application of Error-Sensitive Testing Strategies to Debugging," *ACM SIGSOFT Software Engineering Notes*, Vol.8, No.5, October 1983, 40–42

[FOST84] K.A. Foster, "Sensitive Test Data for Logical Expressions," *ACM SIGSOFT Software Engineering Notes*, Vol.9, No.3, July 1984.

[GEAR78] C.W. Gear, *Programming and Languages*, Science Research Associates, Inc., 1978.

[GELL78] A. Geller, "Test Data as an Aid to Proving Program Correctness," *Communications of the ACM*, 21, 5, May 1978, 368–375.

[GERH76] S.L. Gerhart and L. Yelowitz, "Observations of Fallibility in Applications of Modern Programming Methodologies," *IEEE Transactions on Software Engineering*, SE-2, 3, September 1976, 195–207.

[GOGU79] J.A.Goguen and J.J. Tardo, "An Introduction to OBJ," *Proceedings of the Conference on Specifications of Reliable Software*, 1979, 170–189.

[GRIE76] D. Gries, "An Illustration of Current Ideas on the Derivation of Correctness Proofs and Correct Programs," *IEEE Transactions on Software Engineering* SE-2,4, December 1976, 106–112.

[GROG79] P. Grogono, *Programming in PASCAL*, Addison-Wesley, Inc., 1979.

[HOAR71] C.A.R. Hoare, "Proof of a Program: FIND" *Communications of the ACM*, 14, 1, January 1971, 39–45.

[HOWD76] W.E. Howden, "Reliability of the Path Analysis Testing Strategy," *IEEE Transactions on Software Engineering*, SE-2, 3, September 1976, 208–215.

[HOWD77] W.E. Howden, "Symbolic Testing and the DISSECT Symbolic Evaluation System," *IEEE Transactions on Software Engineering*, SE-3, 4, July 1977, 266–278.

[HOWD78] W.E. Howden, "Algebraic Program Testing," *ACTA Informatica*, 10, 1978.

[HOWD80] W.E. Howden, "Functional Program Testing," *IEEE Transactions on Software Engineering*, SE-6, 2, March 1980.

[HOWD81] W.E. Howden, "Completeness Criteria for Testing Elementary Program Functions," *Fifth International Conference on Software Engineering*, March 1981, 235–243.

[KEMM85] R.A. Kemmerer, "Testing Formal Sepcifications to Detect Design Errors," *IEEE Transactions on Software Engineering*, SE-11,1, January 1985.

[KERN74] B.W. Kernighan and P.J. Plauger, *The Elements of Programming Style*, McGraw-Hill Book Company, 1974.

[KERN83] J.S. Kerner, "Design Methodology Subcommittee Chairperson's Letter and Matrix," *Ada Letters*, 2,6, May/June 1983, 110–115.

[KING69] J.C. King, "A Program Verifier," Ph.D. Dissertation, Carnegie-Mellon University, Pittsburgh, PA, September, 1969.

[KOFF81] E.B. Koffman, *Problem Solving and Structured Programming in Pascal*, Addison-Wesley, Inc., 1981.

[LOND75] R.L. London, "A View of Program Verification," *Proceedings International Conference on Reliable Software*, April 1975, 534–545.

[MYER79] G.J. Myers, *The Art of Software Testing*, John Wiley and Sons, 1979.

[RAMA76] C.V. Ramamorthy, S.F. Ho, and W.T. Chen, "On the Automated Generation of Program Test Data," *IEEE Transactions on Software Engineering*, SE-2, 4, December 1976, 293–300.

[REDW83] S.T. Redwine, "An Engineering Approach to Test Data Design," *IEEE Transactions on Software Engineering.*

[RICH78] D.J. Richardson, "Theoretical Considerations in Testing Programs by Demonstrating Consistency with Specifications," *Digest of the Workshop on Software Testing and Test Documentation*, December 1978, 19–56.

[RICH81a] D.J. Richardson, L.A. Clarke, "A Partition Analysis Method to Increase Program Reliability," *Fifth International Conference on Software Engineering*, March 1981, 244–253.

[RICH81b] D.J. Richardson, "A Partition Analysis Method to Demonstate Program Reliability," Ph.D. Dissertation, University of Massachusetts, September 1981.

[ROWL81] J.H. Rowland and P.J. Davis, "On the Use of Transcendentals for Program Testing," *Journal of the Association for Computing Machinery* 28,1, January 1981, 181–190.

[SILV79] B.A. Silverburg, L. Robinson, and K.N. Levitt, "The HDM Handbook, Volume 1: The Languages and Tools of HDM," Stanford Research Institute Project 4828, June 1979.

[WEGN80] P. Wegner, *Programming with Ada: An Introduction by Means of Graduated Examples*, Prentice-Hall, Inc., 1980.

[WELS79] J. Welsh and J. Elder, *Introduction to Pascal*, Prentice-Hall, Inc., 1979.

[WEYU80] E.J. Weyuker and T.J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains," *IEEE Transactions on Software Engineering*, SE-6, 3, May 1980, 236–246.

[WEYU81] E.J. Weyuker, "An Error-Based Testing Strategy," Computer Science Department, New York University, New York, New York, Technical Report No.027, January 1981.

[WHIT80] L.J. White and E.I. Cohen, "A Domain Strategy for Computer Program Testing," *IEEE Transactions on Software Engineering*, SE-6, 3, May 1980, 247–257.