A Partition Analysis Method to Increase Program Reliability

Debra J. Richardson
Lori A. Clarke

COINS Technical Report 80-12
May 1980

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

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

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

Field	Content
1. REPORT NUMBER	University of Massachusetts COINS Technical Report 80-12
2. GOVT ACCESSION NO.	
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle)	A Partition Analysis Method to Increase Program Reliability
5. TYPE OF REPORT & PERIOD COVERED	Final
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s)	Debra J. Richardson Lori A. Clarke
8. CONTRACT OR GRANT NUMBER(s)	AFOSR 77-3287 NSFMCS 77-02101
9. PERFORMING ORGANIZATION NAME AND ADDRESS	Computer and Information Science Department University of Massachusetts Amherst, Massachusetts 01003
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
11. CONTROLLING OFFICE NAME AND ADDRESS	Air Force Office of Scientific Research Washington, D.C.
12. REPORT DATE	May 1980
13. NUMBER OF PAGES	48
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	
15. SECURITY CLASS. (of this report)	Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited

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

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

error detection, program specification, program testing, program verification, symbolic evaluation, symbolic testing, test data selection

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

A major drawback of most program testing tools 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. The partition analysis method employs symbolic evaluation techniques to partition the set of input data into subdomains, where

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20.

the elements of each subdomain are treated uniformly by the specification and processed uniformly by the implementation. This partition divides the problem domain into more manageable units. Information related to each subdomain is used to guide in the selection of test data that either reveals errors in the implementation or provides confidence in its correctness for the whole subdomain. 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, and thus increase program reliability.

Abstract

A major drawback of most program testing tools 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. The partition analysis method employs symbolic evaluation techniques to partition the set of input data into subdomains, where the elements of each subdomain are treated uniformly by the specification and processed uniformly by the implementation. This partition divides the problem domain into more manageable units. Information related to each subdomain is used to guide in the selection of test data that either reveals errors in the implementation or provides confidence in its correctness for the whole subdomain. 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, and thus increase program reliability.

Keywords: error detection, program specification, program testing, program verification, symbolic evaluation, symbolic testing, test data selection.

1. Introduction

A major drawback of most program testing tools is that they ignore program specifications, and instead base their analysis solely on the information provided in the implementation. Thus, established testing methods that select test data from an understanding of the problem specification are not being utilized. Although these methods, which are sometimes referred to as black box testing, have traditionally been intuitive, they are beneficial in that they may direct attention to aspects of the problem that were neglected in the implementation. Goodenough and Gerhart [GOO76] have demonstrated the value of employing specifications in the test data selection process. Howden [HOW79] has studied the effectiveness of several program validation techniques and proposed a functional testing method [HOW80], which requires that a program's implementation be examined along with its internal documentation. Weyuker and Ostrand [WEY80] proposed a testing strategy based on a partition of the set of input data that is determined by analyzing both the specification and the implementation. None of these methods, however, have exploited formal specifications, which are becoming more available as their value in program development is recognized.

We are exploring a method, called _partition analysis_, that assists in program testing and program verification by

incorporating information from both the specification and the implementation. In this method, symbolic evaluation techniques are employed to partition the set of input data for a problem into subdomains, where the elements of each subdomain are treated uniformly by the specification and processed uniformly by the implementation. By forming these subdomains, we are dividing the problem domain into more manageable units. Information related to each subdomain is used to guide in the selection of test data that reveals errors in the implementation or provides confidence in its correctness for the whole subdomain. 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, which determines and analyzes these subdomains. To facilitate the presentation, we assume that the given specification is correct; thus we are considering the correctness of an implementation with respect to this specification. The next section describes the form of program specification we consider here. We present common

representations for program specifications and implementations and define the subdomains into which the set of input data is partitioned. The third section defines consistency properties between the program specification and implementation, which are based on this partition. The fourth section outlines a technique for finding the subdomains and describes 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 subdomain. In the conclusion, the potential for automating the partition analysis method and several areas of future research are discussed.

2. Problem Representation

During the development of a program, the problem to be solved is represented by successively more elaborate descriptions of the proposed program. The initial representation of the problem is a requirements document, an informal description of the desired program behavior. From this document, an initial program specification is developed. The specification is refined until finally resulting in an implementation, a representation in a programming language. 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. To facilitate this analysis, common representations of the specification and implementation are desired. This section describes the type of specification we consider for analysis, the common representations of the implementation and specification, and the symbolic evaluation technique that is used to generate these common representations. In this section, we also define the subdomains that partition the set of input data, upon which the partition analysis method is based.

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

When a procedure is described by an input-output specification, the relationships between the input values and the output values are provided by pairs of assertions. Whenever the input values satisfy an initial assertion, a correct implementation of the procedure will compute output values satisfying the corresponding final assertion. The technique of input-output specification has been used extensively in the inductive assertion method of program verification [LON75].

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

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

<u>Description</u>

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.

<u>Interface</u>

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

<u>Operation</u>

```
TRAP(F, A, B, N, AREA, ERROR) is
     if N<1 then
        ERROR := true;
        AREA := 0.0;
     else
        ERROR := false;
        AREA :=  Σ_{i=1}^{N} ((F(A+(i-1)*H) + F(A+i*H)) / 2)
                   * abs(H);
     endif;
```

<u>Abbreviations</u>

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

Figure 1

Operational Specification of 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
        --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 TRAP

this paper to demonstrate the partition analysis method.

An operational specification defines the intended function of a procedure. This function is usually composed of partial functions, where each partial function is defined over a subset of the problem domain. An operational specification S, therefore, can be represented as a set of subspecifications $\{S_1, S_2, \ldots, S_M \mid 1 \le M \le \infty\}$.* For each subspecification S_I, the subspecification domain $D[S_I]$ is the set of input data for which the subspecification is applicable and the subspecification computation $C[S_I]$ is the computation specified for those input values. The specification domain D[S] is the union of the subspecification domains, $D[S] = \cup_{I=1}^{M} D[S_I]$.

Similarly, the implementation of a procedure defines a function, which again is composed of partial functions. Each partial function corresponds to a path - that is, a sequence of statements through the procedure. Thus, a procedure implementation P defines a set of paths $\{P_1, P_2, \ldots, P_N \mid 1 \le N \le \infty\}$.** Associated with each path P_J is the path domain $D[P_J]$, which is the set of input data that

---

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

**Likewise, there may be an infinite number of paths due to program loops. As will be explained shortly, we also allow a path to define a class of related paths through the implemenation that differ only by the number of iterations of some loops.

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, $D[P] = \cup_{J=1}^{N} D[P_J]$.

Symbolic evaluation [BIC79, BOY75, CHE79, CLA76, DAR78, HOW77, HUA75, KIN76, MIL75, RAM76, VOG80] is used to provide representations of the domains and the computations for both the specification and the implementation. Symbolic evaluation of an implementation assigns symbolic names to the input values and "executes" a path. During symbolic evaluation, 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. These expressions may then be converted to a canonical form [RIC78a]. 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 represented by the conjunct of these constraints, thus defining the subset of the domain that executes this path. This conjunct of constraints can be translated into some canonical form, such as a simplified, conjunctive normal form. Symbolic evaluation of an implementation can be extended [CHE79, CLA80] to represent a class of paths in which the paths 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 while loop in procedure TRAP and the complete symbolic evaluation of procedure TRAP are shown in the appendix. Figure 3 provides the domains and computations for the classes of paths derived by symbolic evaluation of the implementation of procedure TRAP.

Symbolic evaluation can also be applied to a subspecification, thus constructing representations of the computation and domain. In addition to using iteration constructs like those in programming languages, a specification language may represent the repetition of a transformation by a closed form expression, such as summation or product notation or a set of recurrence relations. Note that summation notation was used in the specification of procedure TRAP. Symbolic evaluation must be modified to handle these constructs. The subspecification domains and computations, which were derived by symbolic evaluation of the specification of procedure TRAP, are given in Figure 4.

The specification is a more abstract representation of a procedure than the implemenation. In some cases, the specification domain and implementation domain may not be the same. Since we assume the specification correctly represents the procedure, the specification domain must be

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

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

$D[P_3] = \{(a,b,n) \mid (n \geq 1) \text{ and } (a>b)\}$
$C[P_3] =$ AREA: 0.0
ERROR: false

$D[P_4] = \{(a,b,n) \mid (n \geq 1) \text{ and } (a<b)\}$
$C[P_4] =$ 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

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

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

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

Subspecification Domains and Computations
for the Specification of Procedure TRAP

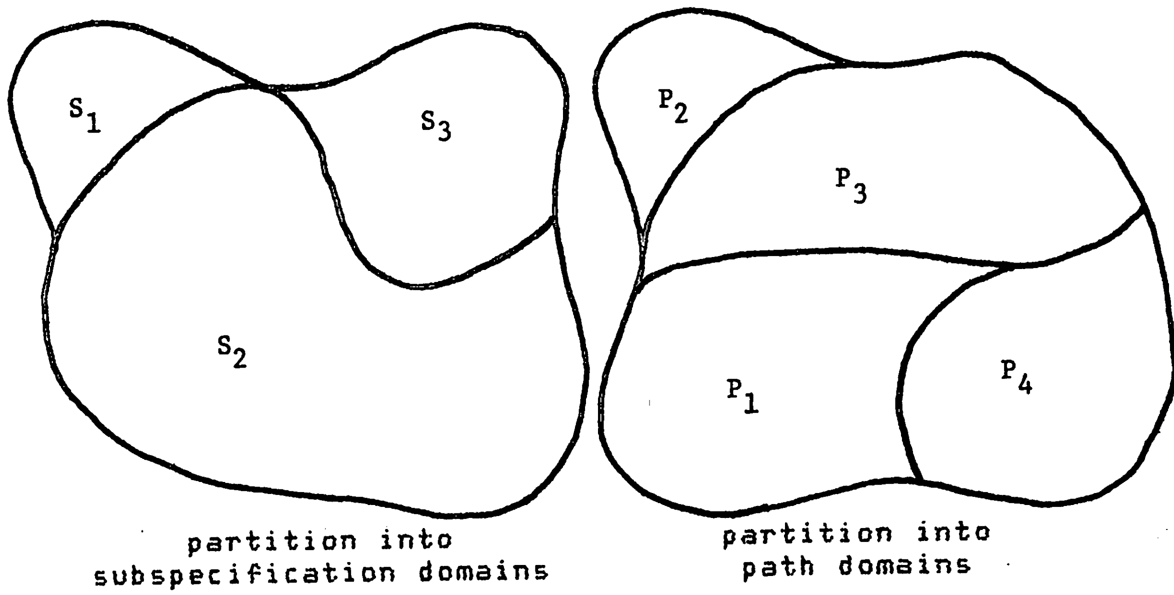
Note: a, b, and n are the symbolic names for the input values A, B, and N, respectively.

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 partitions imposed by a specification and an implementation represent two ways in which a problem may be divided. A specification partitions its domain so that all elements of a subspecification domain are treated uniformly by the specification, as described by the corresponding subspecification computation. Similarly, the implementation partitions its domain so that all elements of a path domain are processed uniformly by the implementation, as described by the corresponding path computation. It would not be surprising to find a subspecification 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 subspecification 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 or these differences may be due to domain or missing path errors [GOO76, HOW76a]. Each intersection, therefore, is a subdomain that should be tested separately. Each such intersection, called a

procedure subdomain, is treated by a single subspecification and a single path, thus only the two associated computations need be examined to verify consistency or to select test data to check for computation errors.

The partition imposed by these procedure subdomains can be visualized by overlaying the subspecification domains and path domains. Figure 5 shows a hypothetical example of the procedure subdomains that would result from overlaying partitions of the specification and implementation domains. A procedure subdomain D_{IJ} is the set of input data for which the subspecification S_I and the path P_J are applicable – that is, $D_{IJ} = D[S_I] \cap D[P_J]$. In addition, for each subspecification S_I there may be input data in its domain $D[S_I]$ that are not treated by any path; this set, $D_{I0} = D[S_I] - \cup_{J=1}^{N} D_{IJ}$, 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 subspecification; this set, $D_{0J} = D[P_J] - \cup_{I=1}^{M} D_{IJ}$, is a procedure subdomain as well. The representations of the procedure subdomains can be constructed by using the representations of the subspecification domains and the path domains, which are created by symbolic evaluation. 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.



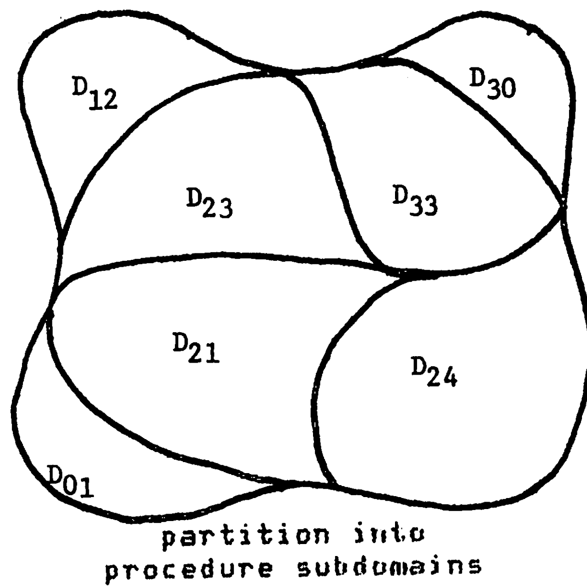


Figure 5

Development of Procedure Subdomains

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

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

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

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

Figure 6

Procedure Subdomains of Procedure TRAP

Note: D_{12}, D_{13}, D_{14}, D_{21}, D_{23}, D_{31}, D_{32}, and D_{34} are empty

3. Consistency Between Procedure Specifications and Implementations

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

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 subspecification domain are treated by some path and

all elements of each path domain are treated by some subspecification. Hence, all D_{IO} and D_{OJ} 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. An implementation P is equivalent to a specification S if for all $x \in D[S]$, $P(x)=S(x)$, where $P(x)$ is the output vector resulting from execution of P on x, and $S(x)$ is the output vector specified by S for x. Equivalence between a procedure implementation and a specification implies the implementation is correct with respect to the specification.

This property of equivalence can be stated in terms of the relationships between the subspecifications and paths over the procedure subdomains. For any input vector x, a particular path, say P_J, is executed – $x \in D[P_J]$ – and a particular subspecification, say S_I, is applicable –

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

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

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

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

Equivalence is sometimes very difficult, if not impossible, to determine. A restricted form of equivalence between a specification and an implementation is isomorphism, which is often an easier property to determine for those cases in which it applies. When isomorphism holds

each subspecification is uniquely associated with an equivalent path. A subspecification and a path are equivalent if their domains are equal and their computations are equal over that domain.

<u>Definition</u>: A subspecification S_I and a path P_J are equivalent if $D[S_I]=D[P_J](=D_{IJ})$ and $C[S_I] \underset{D_{IJ}}{=} C[P_J]$. This is denoted by $S_I \equiv P_J$.

This relationship between subspecifications and paths gives rise to a definition of an isomorphism between a specification and an implementation of a procedure.

<u>Definition</u>: An implementation P is <u>isomorphic</u> to a specification S if there exists a bijective mapping $B{:}S\text{->}P$ such that $B(S_I)=P_J$ if and only if $S_I \equiv P_J$.

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

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 to 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 subspecifications 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 subspecifications and paths can often be determined. Equivalence will then be considered for the remaining subdomains. Testing as well as verification techniques will be used in the partition analysis method.

4. Partition Analysis Verification

Partition analysis verification, a method for demonstrating consistency between a specification and an implementation, is described in this section. This method employs several established verification and validation techniques. 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. Several problems are described and some solutions and areas of future investigation are proposed. A more detailed explanation of the approach is given in [RIC78b].

Determining compatibility between an implementation and a specification is similar to determining uniformity between procedure interfaces in an implementation [GAN78, RAM75, OST76] or between levels of design [CAI75, ROB77, TEI77]. Determining compatibility is facilitated if the specification and programming languages have similar constructs for declaring parameters and global variables. By comparing such declarations, it is possible to determine if the implementation and the specification have the same number and type of parameters and global variables. If the languages do not support explicit declarations on how these variables are used, then data flow analysis methods [OST76] may be utilized to determine 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 [ABR79], 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. In our example, the compatibility of the implementation and specification of procedure TRAP is clear.

Once compatibility is established, partition analysis verification can proceed with the demonstration of additional consistency by comparing the subspecification and path domains and the subspecification and path computations. Since both the specification and the implementation are unambiguous, the subspecification domains are mutually disjoint as are the path domains. No such restriction, however, has been made on the computations; neither the subspecification computations nor the path computations must be distinct. For example, two paths might perform the same computation for their respective domain. With this in mind, the comparison of a specification and an implementation is driven by the relationships between the subspecification

domains and the path domains.

The partition analysis method requires the symbolic representations of the subspecification domains and computations and the path domains and computations provided by symbolic evaluation. Partition analysis verification is then applied to determine isomorphism or equivalence. This method first constructs the procedure subdomains and then compares the symbolic representations of the subspecification and path computations associated with each subdomain. These two steps and some related problems will be described in the remainder of this section. The application of partition analysis verification for procedure TRAP, which is shown in Figure 7, is used to illustrate this method.

In constructing the procedure subdomains, the subspecification domains and path domains are first compared for equality. 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 $D[S_K] \cap \sim D[P_L]$ is empty (where $\sim D[P_L]$ is the complement of $D[P_L]$). For each subspecification domain and path domain that are equal, one procedure subdomain is provided. For

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

$$C[S_1] = C[P_1] = \text{AREA: } 0.0$$

$$\text{ERROR: true}$$

$$\Rightarrow \quad S_1 \equiv P_1$$

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

$$C[S_3] - C[P_3] = \text{AREA: } (a*F(a) + a*F(b) - b*F(a) - b*F(b) + 2*a* \textstyle\sum_{i=1}^{n-1} (F(((n-i)*a+i*b)/n)) - 2*b* \sum_{i=1}^{n-1} (F(((n-i)*a+i*b)/n))) / 2*n - 0.0$$

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

$$\text{ERROR: false}$$

$$\text{Solution set: } (a=b) \text{ or } (F(a)+F(b)+ \textstyle\sum_{i=1}^{n-1} (F(((n-i)*a+i*b)/n)) = 0.0)$$

$$\Rightarrow \quad C[S_3] \underset{D_{33}}{\neq} C[P_3]$$

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

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

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

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

$$\text{ERROR: false}$$

$$\text{Solution set: } (a=b) \text{ or } (F(a)+F(b)+ \textstyle\sum_{i=1}^{n-1} (F(((n-i)*a+i*b)/n)) = 0.0)$$

$$\Rightarrow \quad C[S_2] \underset{D_{22}}{=} C[P_2]$$

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

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

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

$$\text{ERROR: false}$$

$$\Rightarrow \quad C[S_2] \underset{D_{24}}{=} C[P_4]$$

Figure 7

Demonstration of Equivalence of the
Specification and Implementation of Procedure TRAP

each remaining subspecification S_I and path P_J, the intersection can be constructed by conjoining the representations of the subspecification domain and path domain. If this intersection is empty, then no input data exists that causes execution of the path and for which the subspecification 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 $D[S_2] \cap D[P_4]$.

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. This method is subject to the limitations of automatic theorem proving [ELS72]. 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. Several algebraic techniques, such as a linear programming or a gradient hill climbing algorithm, can be used to solve a system of constraints. No method, however, can solve any arbitrary system of constraints [DAV73].

After determining the procedure subdomains, the equality of the subspecification computation and the path

computation over each subdomain must be determined. Often, a term-by-term comparison of the symbolic representations of the subspecification computation $C[S_I]$ and the path computation $C[P_J]$ reveals that the two computations are symbolically identical, 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 $C[S_I] - C[P_J] = 0$. 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, a mathematical package for finding the zeroes of a function can be used. 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 $C[S_2] - C[P_2] = 0$. It can sometimes be determined that the subspecification and path computations are not equal over the associated procedure subdomain. This is the case when the elements of the procedure subdomain do not lie in the

set for which the symbolic difference of the computations is zero.

Partition analysis verification enabled the detection of a fairly subtle error in the implementation of procedure TRAP. In implementing the <u>while</u> 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 $C[S_3] - C[P_3] = 0$. 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 subspecification 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. Several other approaches to deciding computation equality are proposed in [RIC78b], although, in general, this is undecidable.

Partition analysis verification, which compares the symbolic representations of the domains and computations, is a variation on symbolic testing [HOW77]. Other symbolic testing techniques involves merely examining the symbolic

representations of the path domains and computations. Our method has an advantage, however, in that it compares these representations with the symbolic representations of the domains and computations of a specification, which is presumably correct. 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 subspecification 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 subspecifications and paths, then the implementation and specification are isomorphic. Note that a subset of the paths and subspecifications may be in a one-to-one correspondence, as in procedure TRAP. Since the subspecifications and paths in this subset are often similar in structure, it is sometimes easy to determine their equivalence. Partition analysis verification capitalizes on this by determining the equivalent subspecifications and paths first and then concentrates on those parts of the implementation that deviate from the specification.

5. 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, the demonstration of consistency must be complemented by the actual execution of the implementation. Moreover, when consistency cannot be shown, testing can often demonstrate the equality of computations or provide counter examples.

The partition analysis method provides the basis for a test data selection strategy that incorporates both the specification and the implementation. The set of input data is partitioned into procedure subdomains, where the elements of each subdomain are treated uniformly by the specification and processed uniformly by the implementation. Each subdomain is a conceptual unit that should be examined carefully and tested independently.

The concept of dividing the program into smaller units and concentrating on selecting test data for those units is not new. Myers [MYE79] 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 [CLA76, CLA78, HOW75, HOW76a, WHI80] 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 [HOW80] that divides the program into a sequence of transformations. This method symbolically evaluates each transformation and compares the symbolic representation to the internal documentation as well as selects test data for each transformation. Weyuker and Ostrand [WEY80] have proposed a testing method that is based on a partition of the program that is similar to procedure subdomains. Their method assumes that the specification partition is provided. We propose a method that partitions the program 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 symbolic representations of a subdomain and its computations are employed to direct the testing for the

subdomain. The test data selected to exercise the implementation should include typical data points in the subdomain, as well as data chosen to explicitly uncover computation or domain errors.

By examining the representations of the subspecification 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 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. The reader is referred to [HOW78, MYE79] for further guidelines on selecting computationally sensitive test data. 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 [DEM77, HOW76b].

Test data that is apt to reveal domain errors can be chosen by examining the representation of the procedure subdomain. White and Cohen [WHI80] have proposed a strategy for selecting test data to verify path domains, which can be applied to procedure subdomains. By selecting test data near the borders of a procedure subdomain, errors in the

path domain are likely to be detected. Furthermore, missing path errors, which occur when a subspecification is neglected in the implementation, will in all probability be detected by partition analysis testing. This is because each subspecification domain imposes the construction of at least one procedure subdomain; thus data will be chosen that should exercise the missing path.

The test data selected for procedure subdomain D_{24} of procedure TRAP using the strategies outlined above are shown in Figure 8.

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.

<u>TEST CASES</u>

n=1	test cases to
n=2	detect domain
n>>1	errors
$a=b-\epsilon$	
a=0, b=1	
a=-1, b=0	
a=-1, b=1	
a=-k, b=-1	test cases 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 large positive real value

Figure 8

Test Data Selected for Procedure
Subdomain D_{24} of Procedure TRAP

6. Conclusion

The partition analysis method integrates the evaluation of specifications and implementations to assist in program testing and program verification. When a formal specification is available, the partition analysis method can be applied and greatly enhance the reliability of software. This method relies on the development of procedure subdomains, which partition the set of input data based on 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 yet to be seriously considered.

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

As progress in the design of formal specification languages is made, the feasibility of a mechanical means of using new types of specification in program analysis must be evaluated.

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

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 here uses a loop analysis technique [CHE79] 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 grouped together as a class of paths. 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. This loop analysis technique is described in general in [CLA80].

Figure A.1 shows the loop analysis for the <u>while</u> 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 LEC_k, 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 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 and provide the values on exit from the loop, must be added to the loop representation. The final iteration count, call it 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 $AREA(k_e)$.

Recurrence Relations $(k \geq 2)$

$X_k = X_{k-1} + H$

$AREA_k = AREA_{k-1} + F(X_k)$

Loop Exit Condition $(k \geq 2)$

$LEC_k = \sim(X_k < B)$

Solved Recurrence Relations $(k \geq 2)$

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

$AREA(k) = AREA_1 + \Sigma_{i=2}^{k} (F(X_1+(i-1)*H))$

Solved Loop Exit Condition $(k \geq 2)$

$LEC(k) = X_1 + (k-1)*H \geq B$

Final Loop Iteration Count

$\exists k_e: integer \varepsilon [2, \infty),$
$k_e = minimum \{k \mid (X_1+(k-1)*H \geq B)\}$

$= \exists k_e: integer \varepsilon [2, \infty),$
$(\forall k: integer \varepsilon [2, k_e-1], (X_1+(k-1)*H < B))$
$and\ (X_1+(k_e-1)*H \geq B)$

Closed Form Representation

if $X_1 \geq B$ then
--exit loop before first iteration
$X = X_1$
$AREA = AREA_1$
else if $(X_1 < B)$ and $(\exists k_e: integer \varepsilon [2, \infty),$
$(\forall k: integer \varepsilon [2, k_e-1], (X_1+(k-1)*H < B))$
$and\ (X_1+(k_e-1)*H \geq B))$ then
$X = X_1 + (k_e-1)*H$
$AREA = AREA_1 + \Sigma_{i=2}^{k_e} (F(X_1+(i-1)*H)$
endif

Figure A.1

Loop Analysis of <u>while</u> loop in Procedure TRAP

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 symbolic evaluation of procedure TRAP appears in figure A.2. The <u>while</u> 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 <u>while</u> loop.

Path P_1

statement or edge	condition defining path domain	changes in path computation
0	TRUE	(A=a. B=b, N=n, AREA=λ, ERROR=λ, H=λ, X=λ)
(0, 1)	TRUE and (n<1) =(n<1)	
1		ERROR=true
2		AREA=0.0
12		end

Path P_2

statement or edge	condition defining path domain	changes in path computation
0	TRUE	(A=a, B=b, N=n, AREA=λ, ERROR=λ, H=λ, X=λ)
(0, 3)	TRUE and ~(n<1) =(n$\geq$1)	
3		ERROR=false
(3, 4)	(n$\geq$1) and (a=b)	
4		AREA=0.0
12		end

Figure A.2

Symbolic Evaluation of Procedure TRAP

Path P_3

statement or edge	condition defining path domain	changes in path computation
0	TRUE	(A=a, B=b, N=n, AREA=λ, ERROR=λ, H=λ, X=λ)
(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
(7, 10)	(n≥1) and (a≠b) and ~(a<b) =(n≥1) and (a>b)	
		X=a
		AREA=F(a)/2.0
10		AREA=(F(a)/2.0-F(a)/2.0) *(b-a)/n
(10, 11)	(n≥1) and (a>b) and (a>b) =(n≥1) and (a>b)	
11		AREA=-(F(a)/2.0-F(a)/2.0) *(b-a)/n
12		end

Figure A.2

(continued)

Path P_4

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

Figure A.2

(continued)

Note: Elementary algebraic techniques were used to solve for k_e.

REFERENCES

ABR79 P. 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

HOW78 W. E. Howden, "A Survey of Dynamic Analysis Methods," Tutorial: Software Testing and Validation Techniques, IEEE Computer Society, Long Beach, CA, 1978, 184-206.

HOW79 W. E. Howden, "An Analysis of Software Validation Techniques for Scientific Programs," University of Victoria, Victoria, British Columbia, DM-171-IR, March 1979.

HOW80 W. E. Howden, "Functional Program Testing," IEEE Trans. on Software Engineering, SE-6, 2, March 1980, 162-169.

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

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

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

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

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

MYE79 G. J. Myers, The Art of Software Testing, John Wiley and Sons, New York, 1979.

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

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

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

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

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

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

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

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

TEI77 D. Teichrow and E. A. Hershey, III, "PSL/PSA: A

Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Trans. on Software Engineering, SE-3, 1, January 1977, 41-48.

VOG80 U. Voges, L. Gmeiner, and A. Amschler von Mayrhauser, "SADAT - An Automated Testing Tool," IEEE Trans. on Software Engineering, SE-6, 3, May 1980, 286-290.

WEY80 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.

WHI80 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.