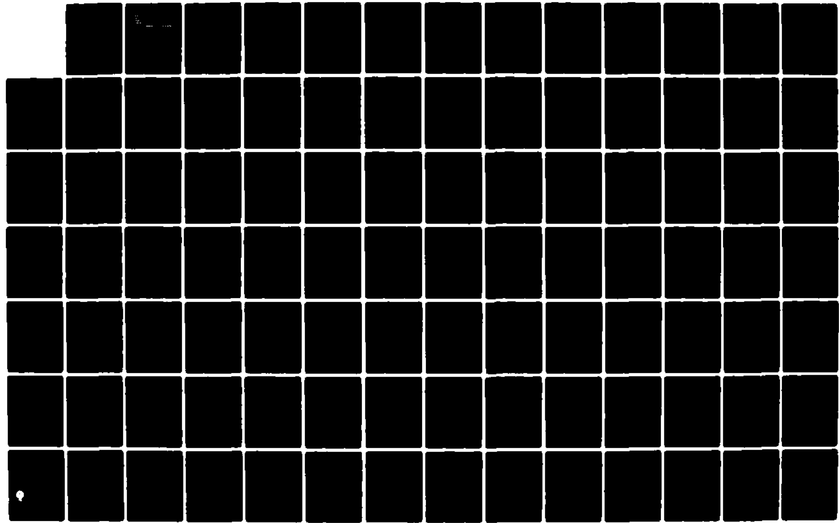
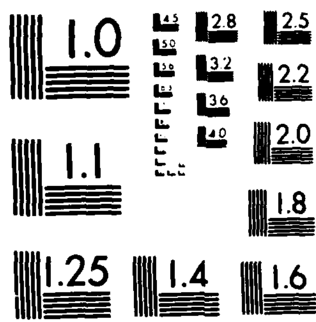
AD-A127 793 THEORETICAL FOUNDATIONS OF SOFTWARE TECHNOLOGY(U) OHIO STATE UNIV COLUMBUS DEPT OF COMPUTER AND INFORMATION SCIENCE B CHANDRASEKARAN ET AL. 14 FEB 83 1/2

UNCLASSIFIED AFOSR-TR-83-0333 F49620-79-C-0152 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AFOSR-TR- 83-0380

AD A127793

RF Project 761640/711991
Final Report

the
ohio
state
university
research foundation
1314 kinnear road
columbus, ohio
43212

THEORETICAL FOUNDATIONS OF SOFTWARE TECHNOLOGY

B. Chandrasekaran, Lee J. White, and H. W. Buttelmann
Department of Computer and Information Science

For the Period
July 1, 1979 - September 30, 1982

DTIC
MAY 9 1983
A

U.S. AIR FORCE
Office of Scientific Research
Bolling Air Force Base, D.C. 2033[illegible]

Contract No. F49620-79-C-0152

Approved for public release;
distribution unlimited.

February 14, 1983

DTIC FILE COPY

83 05 06 -202

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 83-0333	2. GOVT ACCESSION NO. AD-A127793	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) THEORETICAL FOUNDATIONS OF SOFTWARE TECHNOLOGY		5. TYPE OF REPORT & PERIOD COVERED FINAL, 1 JUL 79-30 SEP 82
		6. PERFORMING ORG. REPORT NUMBER 761640/711991
7. AUTHOR(s) B. Chandrasekaran, Lee J. White, and H.W. Buttelmann		8. CONTRACT OR GRANT NUMBER(s) F49620-79-C-0152
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer & Information Science Ohio State University Columbus OH 43212		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS PE61102F; 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Mathematical & Information Sciences Directorate Air Force Office of Scientific Research Bolling AFB DC 20332		12. REPORT DATE 14 FEB 83
		13. NUMBER OF PAGES 132
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)

Approved for public release; distribution unlimited.

17. DISTRIBUTION ST[ATEM]ENT (of [th]e abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY [NO]TES

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

Program testing; domain testing; module testing; integration testing; domain strategy; knowledge-based systems; program synthesis; natural language understanding; problem solving; translator generation; expert systems; diagnostic reasoning.

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

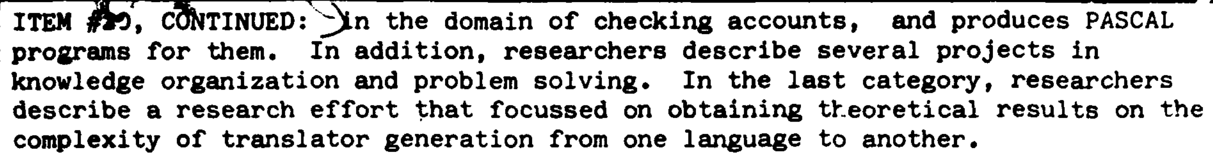
This is the final scientific report of research performed under the contract in various aspects of software technology. The research efforts can be categorized under three topics: (1) computer program testing, (2) knowledge-based systems for program construction, and (3) theory of translator generation. In the first category researchers describe a number of research results relating to various aspects of domain testing strategy and integration testing of modules. In the second category, researchers describe a program called LLULL, which understands programming problems stated in natural language (CONTINUED)

DD FORM 1 JAN 73 1473

83 05 06 -202

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)



ITEM #20, CONTINUED: in the domain of checking accounts, and produces PASCAL programs for them. In addition, researchers describe several projects in knowledge organization and problem solving. In the last category, researchers describe a research effort that focussed on obtaining theoretical results on the complexity of translator generation from one language to another.

AFOSR CONTRACT F49620-79-C-01[illegible]

THEORETICAL FOUNDATIONS OF SOFTWARE TECHNOLOGY

FINAL SCIENTIFIC REPORT
FOR RESEARCH PERIOD
1 JULY 1979 - 30 SEPTEMBER 1982

1. INTRODUCTION

This Contract was a continuation of earlier grants AFOSR 77-3416 to Professors B. Chandrasekaran and Lee J. White, and AFOSR 75-2811 to Prof. H. W. Buttelmann. In view of the many common interests of the two grants, they were merged into this new award. Over the three years, the research activities supported by this Contract can be classified into three broad categories:

1. Computer Program Testing, directed by Professor B. Chandrasekaran and Professor Lee J. White, with the assistance of Faculty Associate, Professor Stuart H. Zweben. This category of research dealt with : the theory of Domain Testing Strategy; a theory of sufficient testing; investigation of how modules ought to be tested and integrated so that the final testing effort does not suffer a combinatorial increase in test effort.

2. Knowledge-Based Systems for Automatic Program Synthesis, directed by Professor B. Chandrasekaran. In this research, Research Associate Fernando Gomez investigated the design of a system to understand programming problems stated in natural language and to produce PASCAL language code for the problems. Because of its relationship to many issues in Artificial Intelligence, we also investigated problems of knowledge-based problem solving in general, and issues in natural language understanding.

3. Automated Translation of Computer Programs, directed by Professor H. William Buttelmann. The major thrust of this work was on getting some theoretical results concerning the computability and complexity of translator generation.

Our approach in this Final Scientific Report will be as follows. We will present the main technical results in each area as a sequence of technical articles or reports. These reports by no means exhaust all the publications that have resulted from the grant, but will be chosen to cover all the results in a concise manner. In a final section we list other publications and describe professional activity by researchers supported by this grant, including Ph. D dissertations.

AIR FORCE [illegible] SCIENTIFIC [illegible]
NOTICE OF [illegible] TO DTIC
This [illegible]
appro[illegible]
Distr[illegible] AFR 190-12.
MATTHE[illegible]
Chief, [illegible] Information Division

TABLE OF CONTENTS

1. INTRODUCTION 0

2. TECHNICAL SUMMARY 1

2.1. Computer Program Testing 1
2.1.1. Domain Testing Strategy 1
2.1.2. Sufficient Testing 24
2.1.3. Module Integration Testing 54
2.1.4. Other Issues in Testing 75
2.2. Knowledge-Based Program Synthesis and Problem Solving 79
2.2.1. Understanding Programming Problems Stated in Natural Language 79
2.2.2. Natural and Social System Metaphors in Distributed Problem Solving: Introduction to the Issue 115
2.3. Automated Translation of Computer Programs 121

3. PUBLICATIONS AND OTHER ACTIVITY 127

3.1. List of Publications 127
3.2. Dissertations supported by the Grant 128

Accession For
NTIS GRA&I ☑
DTIC TAB ☐
Unannounced ☐
Justification
Distribution/
Availability Codes
Avail and/or
Dist Special
A

2. TECHNICAL SUMMARY

2.1. Computer Program Testing

2.1.1. Domain Testing Strategy

Introduction

Several years ago a research group at Ohio State University developed an automated testing approach called the Domain Testing Strategy. The original research in this area was supported by AFOSR Grant 75-2811, and we have reported in a series of papers the main ideas behind the approach. Several new testing approaches have been motivated by this initial work. Much of this new work was done here with support from the AFOSR Contract.

In this section, we present two papers that summarize our research relating to the Domain Testing Strategy. The first one, by Lee White, Edward Cohen and Steve Zeil, appeared in Computer Programe Testing, edited by Chandrasekaran and Radicchi (North Holland), and gives a presentation of the approach. The second one is by Lee White, and it appears in Workshop Digest, Workshop on Effectiveness of Testing and Proving Methods, Avalon, Calif., May 11-13, 1982. This paper discusses the impact of the domain testing approach on further research activities in the field.

COMPUTER PROGRAM TESTING
B. Chandrasekaran, S. Radicchi (eds.)
North-Holland Publishing Company
© SOGESTA, 1981

A DOMAIN STRATEGY FOR COMPUTER PROGRAM TESTING[1]

Lee J. White, Edward I. Cohen, and Steven J. Zeil

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio
U.S.A.

This paper presents a testing strategy designed to detect errors in the control flow of a computer program, and the conditions under which this strategy is reliable are given and characterized. The testing strategy generates test points to examine the boundaries of a path domain to detect whether a domain error has occurred; the number of test points required to test each domain grows only linearly with both the dimensionality of the input space and the number of predicates along the path being tested. A new method is described to decide whether an additional path should be tested when a number of paths have already been tested, or whether no additional information can be gained by testing this path, i.e., that the program has been "sufficiently tested".

1. INTRODUCTION

Computer programs contain two types of errors which have been identified as computation errors and domain errors by Howden [4]. A _domain error_ occurs when a specific input follows the wrong path due to an error in the control flow of the program. A path contains a _computation error_ when a specific input follows the correct path, but an error in some assignment statement causes the wrong function to be computed for one or more of the output variables. A testing strategy has been designed to detect domain errors, and the conditions under which this strategy is reliable are given and characterized. A byproduct of this domain strategy is a partial ability to detect computation errors. This study and proposed methodology are described in greater detail in Cohen [3] and in White and Cohen [5,6].

There are limitations inherent to any testing strategy, and these also constrain the proposed domain strategy. One such limitation might be termed _coincidental correctness_, which can occur when a specific test point follows an incorrect path, and yet the output variables coincidentally are the same as if that test point were to follow the correct path. This test point would then be of no assistance in the detection of the domain error which caused the control flow change. No path-oriented test generation strategy can circumvent this problem.

Another inherent testing limitation has been previously identified by Howden [4] as a _missing path error_, in which a required predicate does not appear in the given program to be tested. Especially if this predicate were an equality, Howden has indicated that no path-oriented testing strategy could systematically determine that such a predicate should be present.

An important assumption in our work is that the user or an "oracle" is available who can decide unequivocally if the output is correct for the specific input processed. The oracle decides only if the output values are correct, and not whether they are computed correctly. If they are incorrect, the oracle does not provide any information about the error and does not give the correct output values.

2. PREDICATE INTERPRETATIONS

Every branch point of a computer program is associated with a predicate which evaluates to true or false, and its value determines which outcome of the branch will be followed. The path condition is the compound condition which must be satisfied by the input data point in order that the control path be executed. It is the conjunction of the individual predicate conditions which are generated at each branch point along the control path. Not all the control paths that exist syntactically within the program are executable. If input data exist which satisfy the path condition, the control path is also an execution path and can be used in testing the program. If the path condition is not satisfied by any input value, the path is said to be infeasible, and is of no interest in testing the program.

A simple predicate is said to be linear in variables V_1, V_2 ....., V_n if it is of the form

$$A_1V_1 + A_2V_2 + ... + A_nV_n \text{ ROP } K,$$

where K and the A_i are constants, and ROP represents one of the relational operators ($<,>,=,\leq,\geq,\neq$). A compound predicate is linear when each of its component simple predicates is linear.

In general, predicates can be expressed in terms of both program variables and input variables. However, in generating input data to satisfy the path condition we must work with constraints in terms of only input variables. If we replace each program variable appearing in the predicate by its symbolic value in terms of input variables, we get an equivalent constraint which we call the predicate interpretation. A particular interpretation is equivalent to the original predicate in that input variable values satisfying the interpretation will lead to the computation of program variables which also satisfy the original predicate. A single predicate can appear on many different execution paths. Since each of these paths will in general consist of a different sequence of assignment statements, a single predicate can have many different interpretations. The following program segment provides example predicates and interpretations.

```
READ A,B;

IF A > B
   THEN C = B + 1;
   ELSE C = B - 1;

D = 2*A + B;
IF C ≤ 0
   THEN E = 0;
   ELSE
        DO I = 1,B;
           E = E + 2*I;
        END;

IF D = 2
   THEN F = E + A;
   ELSE F = E - A;


WRITE F;
```

In the first predicate, A > B, both A and B are input variables, so there is only one interpretation. The second predicate, $C \leq 0$, will have two interpretations depending on which branch was taken in the first IF construct. For paths on which the THEN C = B + 1 clause is executed, the interpretation is $B + 1 \leq 0$ or

equivalently $B \le -1$. When the ELSE C = B - 1 branch is taken, the interpretation is $B - 1 \le 0$, or equivalently $B \le 1$. Within the second IF-THEN-ELSE clause, a nested DO-loop appears. The DO-loop is executed:

no times if $B < 1$
once if $1 \le B < 2$
twice if $2 \le B < 3$
etc.

Thus the selection of a path will require a specification of the number of times that the DO-loop is executed, and a corresponding predicate is applied which selects those input points which will follow that particular path. Even though the third predicate, D = 2, appears on four different paths, it only has one interpretation, 2*A + B = 2, since D is assigned the value 2*A + B in the same statement in each of the four paths.

3. INPUT SPACE STRUCTURE

An _input space domain_ is defined as a set of input data points satisfying a path condition, consisting of a conjunction of predicates along the path. For simplicity in this discussion, each of these predicates is assumed to be simple. The input space is partitioned into a set of domains. Each domain corresponds to a particular executable path in the program and consists of the input data points which cause the path to be executed.

The boundary of each domain is determined by the predicates in the path condition and consists of _border segments_, where each segment is the section of the boundary determined by a single simple predicate in the path condition. Each border segment can be open or closed depending on the relational operator in the predicate. A _closed border segment_ is actually part of the domain and is formed by predicates with $\le$, $\ge$, or = operators. An _open border segment_ forms part of the domain boundary but does not constitute part of the domain, and is formed by <, >, and $\ne$ predicates.

The general form of a simple linear predicate interpretation is

$$A_1X_1 + A_2X_2 + \dots + A_NX_N \text{ ROP } k$$

where ROP is the relational operator, X_i are input variables, and A_i, K are constants. However, the border segment which any of these predicates defines is a section of the surface defined by the equality

$$A_1X_1 + A_2X_2 + \dots + A_NX_N = K,$$

since this is the limiting condition for the points satisfying the predicate. In an N-dimensional space this linear equality defines a hyperplane which is the N-dimensional generalization of a plane.

Consider a path condition composed of a conjunction of simple predicates. These predicates can be of three basic types: equalities (=), inequalities (<, >, $\le$, $\ge$), and nonequalities ($\ne$). The use of each of the three types results in a markedly different effect on the domain boundary. Each equality constrains the domain to lie in a particular hyperplane, thus reducing the dimensionality of the domain by one. The set of inequality constraints then defines a region within the lower dimensional space defined by the equality predicates.

The nonequality linear constraints define hyperplanes which are not part of the domain, giving rise to open border segments as mentioned earlier. Observe that the constraint $A \ne B$ is equivalent to the compound predicate $(A < B)$ OR $(A > B)$. In this form it is clear that the addition of a nonequality predicate to a set of

inequalities can split the domain defined by those inequalities into two regions.

The foregoing definitions and the example allow us to characterize more precisely domains which correspond to simple linear predicate interpretations.

> For an execution path with a set of simple linear equality or inequality predicate interpretations, the input space *domain is a single convex* polyhedron. If one or more simple linear non-equality predicate interpretations are added to this set, then the input space domain consists of the union of a set of disjoint convex polyhedra.

4. THE DOMAIN TESTING STRATEGY

The domain testing strategy is designed to detect domain errors and will be effective in detecting errors in any type of domain border under certain conditions. Test points are generated for each border segment which, if processed correctly, determine that both the relational operator and the position of the border are correct. An error in the border operator occurs when an incorrect relational operator is used in the corresponding predicate, and an error in the position of the border occurs when one or more incorrect coefficients are computed for the particular predicate interpretation. The strategy is based on a geometrical analysis of the domain boundary and takes advantage of the fact that points on or near the border are most sensitive to domain errors. A number of authors have made this observation, e.g., Boyer et al [1] and Clarke [2].

It should be emphasized that the domain strategy does not require that the correct program be given for the selection of test points, since only information obtained from the given program is needed. However, it will be convenient to be able to refer to a "correct border", although it will not be necessary to have any knowledge about this border. Define the given border as that corresponding to the predicate interpretation for the given program being tested, and the correct border as that border which would be calculated in some correct program.

The domain testing strategy will be developed and validated under a set of simplifying assumptions:

(1) Coincidental correctness does not occur for any test case.
(2) A missing path error is not associated with the path being tested.
(3) Each border is produced by a simple predicate.
(4) The path corresponding to each adjacent domain computes a different function than the path being tested.
(5) The given border is linear, and if it is incorrect, the correct border is also linear.
(6) The input space is continuous rather than discrete.

Assumptions (1) and (2) have been shown to be inherent to the testing process, and cannot be entirely eliminated. However, recognition of these potential problems can lead to improved testing techniques. Assumptions (3) and (4) considerably simplify the testing strategy, for with them no more than one domain need be examined at one time in order to select test points, and as will be indicated shortly, a reduced number of test points will be required. As for the linearity assumption (5), the domain testing method has been shown to be applicable for nonlinear boundaries, but the number of required test points may become inordinate and there are complex problems associated with processing nonlinear boundaries in higher dimensions. The continuous input space assumption (6) is not really a limitation of the proposed testing method, but allows points to be chosen arbitrarily close to the border to be tested. An error analysis has shown that pathological cases do exist in discrete spaces for which the testing strategy cannot be used, but these occur only when domain size is on the order of the resolution of the discrete space itself.

5. TWO-DIMENSIONAL LINEAR INEQUALITIES

Test-Point Selection

The test points selected will be of two types, defined by their position with respect to the given border. An ON test point lies on the given border, while an OFF test point is a small distance ε from, and lies on the open side of, the given border. Therefore, we observe that when testing a closed border, the ON test points are in the domain being tested, and each OFF test point is in some adjacent domain. Conversely, when testing an open border, each ON test point is in some adjacent domain, while the OFF test points are in the domain being tested.

Figure 1 shows the selection of three test points A, B, and C for a closed inequality border segment. The three points must be selected in an ON-OFF-ON sequence. Specifically, if test point C is projected down on line AB, then the projected point must lie strictly between A and B on this line segment. Also point C is selected a distance ε from the given border segment, and will be chosen so that it satisfies all the inequalities defining domain D except for the inequality being tested.

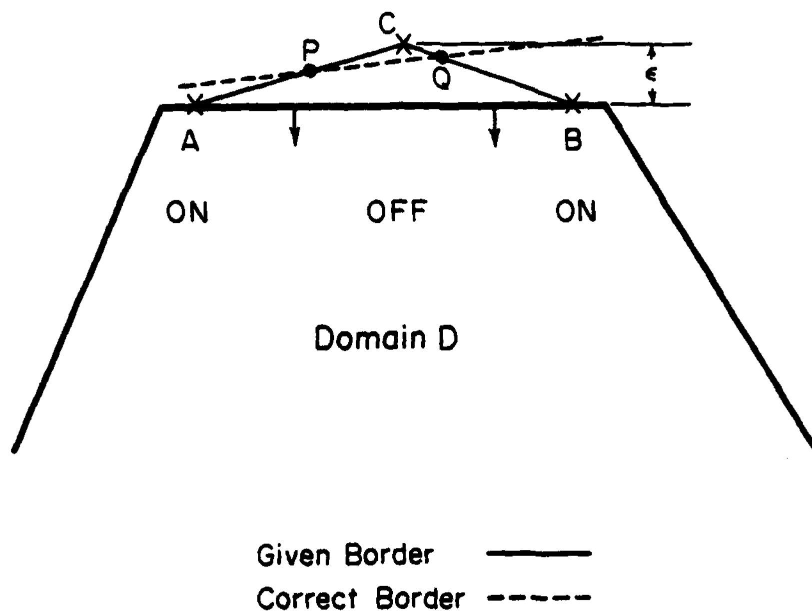


Fig. 1. Test Points for a Two-Dimensional Linear Border

Proof of Reliable Test Selection

It must be shown that test points selected in this way will reliably detect domain errors due to boundary shifts. If any of the test points lead to an incorrect output, then clearly there is an error. On the other hand, if the outputs of all these points are correct, then either the given border is correct, or if it is incorrect, Figure 1 shows that the correct border must lie on or above points A and B, and must lie below point C, for by assumptions (1) and (4), each of these test points must lie in its assumed domain. So if the given border is incorrect, then the correct border can only belong to a class of line segments which intersect both closed line segments AC and BC.

Figure 1 indicates a specific correct border from this class which intersects line segments AC and BC at P and Q respectively. Define the domain error magnitude for this correct border to be the maximum of the distances from P and from Q to the given border. Then it is clear that the chosen test points have detected domain errors due to border shifts except for a *class of domain errors of magnitude less* than ε. In a continuous space ε can be chosen arbitrarily small, and as ε approaches zero, the line segments AC and BC become arbitrarily close to the given border, and in the limit, we can conclude that the given border is identical to the correct border.

Figure 2 shows the three general types of border shifts, and will allow us to see how the ON-OFF-ON sequence of test points works in each case. In Figure 2(a), the border shift has effectively reduced domain D_1. Test points A and B yield correct outputs, for they remain in the correct domain D_1 despite the shifted border. However, the border has shifted past test point C, causing it to be in domain D_2 instead of domain D_1. Since the program will now follow the wrong path when executing input C, incorrect results will be produced. In Figure 2(b), the domain D_1 has been enlarged due to the border shift. Here test point C will be processed correctly since it is still in domain D_2, but both A and B will detect the shift since they should also be in domain D_2. Finally in Figure 2(c), only test point B will be incorrect since the border shift causes it to be in D_1 instead of D_2. Therefore, the ON-OFF-ON sequence is effective since at least one of the three points must be in the wrong domain as long as the border shift is of a magnitude greater than ε.

We must also demonstrate the reliability of the method for domain errors in which the predicate operator is incorrect. If the direction of the inequality is wrong, e.g., $\le$ is used instead of $\ge$, the domains on either side of the border are interchanged, and any point in either domain will detect the error. A more subtle error occurs when just the border itself is in the wrong domain, e.g., $\le$ is used instead of $<$. In this case the only points affected lie on the border, and since we always test ON points, this type of error will always be detected. If the correct predicate is an equality, the OFF point will detect the error.

Complexity of the Test Strategy

The domain testing strategy requires at most 3*P test points for a domain, where P, the number of border segments on this boundary, is bounded by the number of predicates encountered on the path. However, we can reduce this cost by sharing test points between adjacent borders of the domain. The requirement for sharing an ON point is that it is an extreme point for two adjacent borders which are both closed or both open. The number of ON points needed to test the entire domain boundary can be reduced by as much as one half, i.e., the number of test points, TP, required to test the complete domain boundary lies in the following range:

$$2*P \le TP \le 3*P.$$

Even more significant savings are possible by sharing the test points for a common border between two adjacent domains. If both domains are tested independently, the common border between them is tested twice, using a total of six test points. If this border has shifted, both domains must be affected, and the error will be detected by testing either domain.

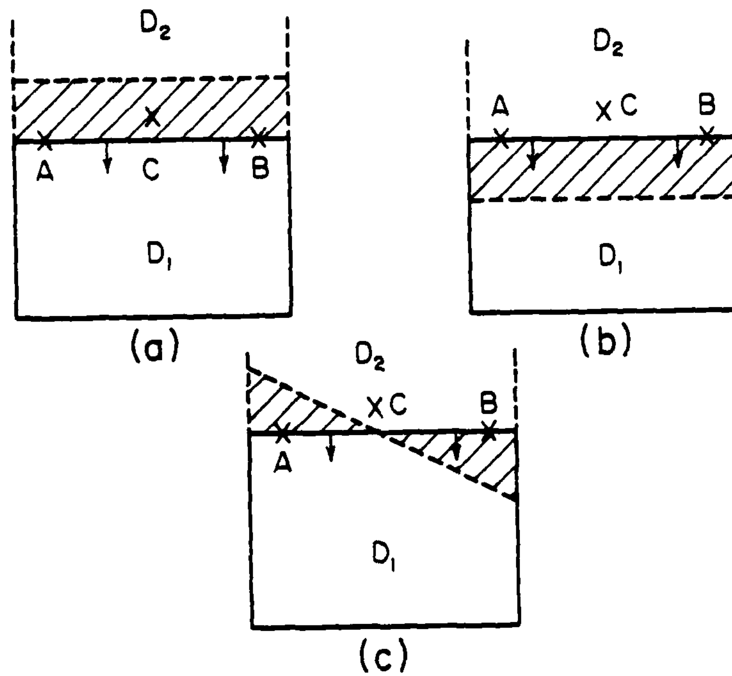


Fig. 2. The Three Types of Border Shifts

6. N-DIMENSIONAL LINEAR INEQUALITIES

The domain testing strategy developed for the two-dimensional case can be extended to the general N-dimensional case in a straightforward manner. The central property used in the previous analysis was the fact that a line is uniquely determined by two points. We can easily generalize this property since an N-dimensional hyperplane is determined by N linearly independent points. So, whereas in the two-dimensional case we had to identify only two points on the correct border, in general we have to identify N points on the correct border, and in addition, these points must be guaranteed to be linearly independent.

The validation of domain testing for the general linear case is based on the same geometric arguments used in the two-dimensional case. The key to the methodology is that the correct border must intersect every OFF-ON line segment, assuming that the test points are all correct. Since we must identify a total of N points on the correct border, N OFF-ON line segments are needed, and we can achieve this by testing N linearly independent ON test points on the given border and a single OFF test point whose projection on the given border is a convex combination of these N points. In addition, as in the two-dimensional case, the OFF point must also satisfy the inequality constraints corresponding to all adjacent borders.

Even though we do not know these specific points at which the correct border intersects the ON-OFF segments, we do know that these points must be linearly independent since the ON points are linearly independent. The OFF point is a distance ϵ from the given border, and in the limit as ϵ approaches zero, each OFF-ON line segment becomes arbitrarily close to the given border. However, as in the two-dimensional case, the ϵ-limitation means that only border shifts of magnitude greater than ϵ will be detected.

The domain testing strategy requires at most (N+1)*P test points per domain, where N is the dimensionality of the input space in which the domain is defined and P is the number of border segments in the boundary of the specific domain. However, we again can reduce this testing cost by using extreme points as ON test points, and by sharing test points between adjacent domains.

7. EQUALITY AND NONEQUALITY PREDICATES

Equality predicates constrain the domain to lie in a lower dimensional space. If we have an N-dimensional input space and the domain is constrained by L independent equalities, the remaining inequality and nonequality predicates then define the domain within the (N-L)-dimensional subspace defined by the set of equality predicates.

The test points for both equality and nonequality predicates can be chosen much as for the inequality case, but there is a technical problem which requires (N+3) test points for the dimensional case. This technical problem and its resolution is described in detail in references [3], [5], and [6]. The following proposition summarizes the results of our investigation:

> Given assumptions (1) through (6), with each OFF point chosen a distance ϵ from the corresponding border, the domain testing strategy is guaranteed to detect all domain errors of magnitude greater than ϵ using no more than P*(N+3) test points per domain, where N indicates the dimensionality of the input space and P is the number of predicates along the path to be tested.

Notice that the number of required points grows only linearly with both parameters N and P, which is about the best one could hope for.

8. SUFFICIENT TEST SETS FOR PATH-ORIENTED TESTING

Stopping Criteria for Testing

Although the number of required test points for each path in the Domain Strategy grows only linearly with the number of input variables and predicates along the path, the problem with this approach is that the number of paths grows in a highly combinatorial fashion and is potentially infinite. Moreover, any path-oriented strategy suffers from this basic problem.

In the definition of any automated path selection strategy, the questions which naturally arise are, "When does testing stop? At what point is it possible to point to a particular program construct and say that it has been sufficiently tested, i.e., no errors remain undetected?" In general, we know that this problem can be proven undecidable, but a programmer's intuition suggests that such claims should be possible after the selection of a small number of test paths, especially if we possess a strategy in which we have specific confidence in terms of its ability to detect certain types of errors in some construct along that path.

In references [7] and [8], Zeil and White have developed a vector space model for programs which satisfy conditions (1) through (6) of Section 6, and this model has indicated substantive answers to these questions for this class of programs. For convenience, let us define such programs satisfying these conditions as <u>linearly domained programs</u>. It should be emphasized that this research and these results are essentially independent of the Domain Strategy, and only require a testing strategy which will reliably detect domain errors associated with a specific predicate when conditions (1) through (6) of Section 6 are satisfied. However, the existence of the Domain Strategy with this degree of reliability allows us to investigate these issues.

Sufficient Testing Sets

In order to state these results more precisely, let us define these questions and concepts carefully. A set of paths is a <u>sufficient set</u> for a program construct if the failure to detect some error in that construct, using a reliable method of selecting data points along those paths, implies that this error would go undetected for <u>any path</u> through the program. In this definition, and throughout this discussion, we might be considering any program construct, but a most concrete construct for which we have such reliable methods is that of a predicate. We can then restate the questions more rigorously as:

a) After a number of paths have been tested which pass through the construct, what is the marginal advantage of testing another path?
b) Is there a point (before nearly all paths have been tested) at which we may say that no more paths need be chosen and tested through some program construct, i.e., that this construct has been <u>sufficiently tested</u>?

Types of Testing Blindness

In order for us to characterize the minimal number of paths which must be tested, we first must clearly understand why multiple paths might be needed in order to detect an error in a construct (such as a predicate). The following examples show three different reasons why a single path may not detect an erroneous predicate. These are termed <u>assignment blindness</u>, <u>equality blindness</u>, and <u>self-blindness</u>, and represent a seemingly pathological set of values for variables along the path so that both the correct and the incorrect predicate evaluate to equal values.

Assignment Blindness

Correct	Incorrect
A = 1	A = 1
⋮	⋮
IF B > 0 THEN	IF B+A > 1 THEN
⋮	⋮

Equality Blindness

Correct	Incorrect
IF D = 2 THEN	IF D = 2 THEN
⋮	⋮
IF C + D > 3 THEN	IF C > 1 THEN
⋮	⋮

Self-Blindness

Correct	Incorrect
X = A	X = A
⋮	⋮
IF X-1 > 0	IF X+A-2 > 0
⋮	⋮

Results from a Vector Space Model

By studying examples of this type, the vector space model examined in references [7] and [8] has yielded an insight as to how multiple paths through a single predicate can resolve these ambiguities due to various types of blindness. This vector space is composed of:

- one vector for each assigned program variable, for a total of n;
- one vector for each equality restriction on the path domain, at most m total, where m is the number of input variables.

The results of this research which provide answers to questions a) and b) posed earlier in this section can be stated as follows:

> For any predicate in a linearly domained program, the smallest sufficient set of test paths will contain at most (m+n+1) paths, where m is the number of input variables and n the number of program variables. Moreover, if a set of paths have been tested which pass through the predicate of interest, a simple vector criterion described in references [7] and [8] will determine whether a proposed additional path is required to detect an error in that predicate.

This is a most satisfying result in that it is consistent with a programmer's intuition that only a reasonably small number of paths should be sufficient to reliably test any construct in a given computer program. We have assumed, however, linearly domained programs in order to obtain this result. The greatest difficulty with this approach is that paths which constitute the smallest sufficient set cannot be generated easily or efficiently. Rather, it is only after a

set of paths are selected that the vector criterion can be applied. Research is continuing on this problem in order to devise heuristic methods to select the set of paths which are based upon ongoing experiments using the vector criterion.

FOOTNOTES

1. This research was supported in part by Air Force Office of Scientific Research Grant F49620-79-0152.

REFERENCES

[1] Boyer, R. S., Elspas, B., and Levitt, K. N., SELECT--A formal system for testing and debugging programs by symbolic execution, Proceedings - 1975 International Conference on Reliable Software, Los Angeles, CA, April 1975, 234-245.

[2] Clarke, L. A., A system to generate test data and symbolically execute programs, IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, Sept. 1976, 215-222.

[3] Cohen, E. I., A Finite Domain - Testing Strategy for Computer Program Testing, Ph.D. Dissertation, Dept. of Comp. Sc., Ohio State Univ. (June 1978).

[4] Howden, W. E., Reliability of the path analysis testing strategy, IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, Sept. 1976, 208-215.

[5] White, L. J., Cohen, E. I., and Chandrasekaran, B., A Domain Strategy for Computer Program Testing, Technical Report 78-4, Comp. and Infor. Sc. Research Center, Ohio State Univ. (August 1978).

[6] White, L. J. and Cohen, E. I., A domain strategy for computer program testing, IEEE Transactions on Software Engineering, Vol. SE-6, No. 3, May 1980, 247-257.

[7] Zeil, S. J. and White, L. J., Sufficient Test Sets for Path Analysis Testing Strategies, Technical Report TR-80-6, Comp. and Infor. Sc. Research Center, Ohio State Univ. (July 1980).

[8] Zeil, S. J. and White, L. J., Sufficient test sets for path analysis testing strategies, Proceedings-5th International Conference on Software Engineering, San Diego, CA, March 9-12, 1981, 184-191.

Some Research Approaches Motivated by the Domain Testing Strategy*

Lee J. White

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210

Abstract

Several years ago a research group at Ohio State University developed an automated testing approach called the Domain Testing Strategy. This paper examines some broader implications of the results of that research, together with several new testing research approaches which have been motivated by this work. For example, recently some new results which characterize a set of paths which are sufficient for path oriented testing have been obtained, motivated to a great extent by domain testing. This approach, in turn, has led to some positive and exciting results in the area of reliable module integration testing. Currently several researchers are examining the issue of specification testing, combining information from the program specification with a structural testing approach; they have found domain testing concepts to be helpful in this regard.

*The research described in this paper was supported in part by the Air Force Office for Scientific Research, Contract F49620-79-C-0152.

1. A Domain Strategy for Computer Program Testing

For the past five years, a research group at Ohio State University has been working in the area of reliable software in general, and program testing in particular. We have developed an automated testing approach called the Domain Testing Strategy [2,13] which appears to be promising for a large class of data processing programs. This method is a form of a path-oriented testing approach, where the process of testing a computer program is treated as two operations [7]:

1) selection of a path or set of paths along which testing is to be conducted, and

2) selection of input data to serve as test cases which will cause the chosen paths to be executed.

For general programs, the problem of generation of reliable test data is known to be unsolvable, e.g., see Howden [6]. For certain classes of programs, however, the domain testing strategy research has shown that it is possible to implement reliable methods of selecting test data for a given path to detect certain types of errors.

Computer programs contain two types of errors which have been identified as computation errors and domain errors by Howden [7]. A domain error occurs when a specific input follows the wrong path due to an error in the control flow of the program. A path contains a computation error when a specific input follows the correct path, but an error in some assignment statement causes the wrong function to be computed for one or more of the output variables. A testing strategy has been designed to detect domain errors, and the conditions under which this strategy is reliable are given and characterized. A byproduct of this domain strategy is a partial ability to detect computation errors. This study and proposed methodology are described in greater detail in Cohen [2] and in White and Cohen [12-13].

There are limitations inherent to any testing strategy, and these also constrain the proposed domain strategy. One such limitation might be termed coincidental correctness, which can occur when a specific test point follows an incorrect path, and yet the output variables coincidentally are the same as if that test point were to follow the correct path. This test point would then be of no assistance in the detection of the domain error which caused the control flow change. No path-oriented test generation strategy can circumvent this problem.

Another inherent testing limitation has been previously identified by Howden [7] as a missing path error, in which a required predicate does not appear in the given program to be tested. Especially if this predicate were an equality, no path-oriented testing strategy could systematically determine that such a predicate should be present.

An important assumption in our work is that the user or an "oracle" is available who can decide unequivocally if the output is correct for the specific input processed. The oracle decides only if the output values are correct, and not whether they are computed correctly. If they are incorrect, the oracle does not provide any information about the error and does not give the correct output values.

The control flow statements in a computer program partition the input space into a set of mutually exclusive domains, each of which corresponds to a particular program path and consists of input data points which cause that path to be executed. The testing strategy generates test points to examine the boundaries of a domain to detect whether a domain error has occurred, as either one or more of these boundaries will have shifted or else the corresponding predicate relational operator has changed. If test points can be chosen within ε of each boundary, the strategy is shown to be reliable in detecting domain errors of magnitude greater than ε, subject to the following assumptions:

(1) coincidental correctness does not occur;

(2) missing path errors do not occur;

(3) predicates are linear in the input variables;

(4) the input space is continuous.

Assumptions (1) and (2) have been shown to be inherent to the testing process, and cannot be entirely eliminated. However, recognition of these potential problems can lead to improved testing techniques. The domain testing method has been shown to be applicable for nonlinear boundaries, but the number of required test points may become inordinate and there are complex problems associated with processing nonlinear boundaries in higher dimensions. The continuous input space assumption is not really a limitation of the proposed testing method, but allows the parameter ε to be chosen arbitrarily small. An error analysis for discrete spaces is available [14], and the testing strategy has been proved viable as long as the size of the domain is not comparable to the discrete resolution of the space.

Now let us consider two further assumptions:

(5) predicates are simple; and

(6) adjacent domains compute different functions.

If assumptions (5) and (6) are imposed, the testing strategy is considerably simplified, as no more than one domain need be examined at one time in order to select test points. Moreover, the number of test points required to test each domain grows linearly with both the dimensionality of the input space and the number of predicates along the path being tested. Any program which satisfies these six constraints will be referred to as a <u>linearly</u> <u>domained</u> program.

2. Some Broader Issues Derived from Domain Testing

One of the major results of domain testing is that, subject to the assumption of a linearly domained program, reliable detection of domain errors requires a reasonable number of test points for a single path. This number of test points grows only linearly with the number of predicates along the path and the number of input variables. However, the total cost is unacceptable for any practical program, as it will routinely contain an excessive number of paths. Thus there has been a significant research effort to substantially reduce the number of paths required for domain testing; Sections 3 and 4 of this paper briefly describe several such research efforts.

One way to view the results from domain testing is to observe that the number of test points required is a minimum for reliable detection of domain errors, and if coincidental correctness should occur, even more test points would be required. However, in many places in the research testing literature, one finds reference to choosing only one test data point per path when a path-oriented strategy is utilized. This work shows clearly that in general this is inadequate for even a modest attempt at reliable testing.

Although we know that the problem of reliable test data generation is unsolvable, the domain testing research has shown that if attention is focused upon specific types of errors and a characterized subset of programs, reliable testing conditions can be obtained. Indeed, the problem here was to find the minimum set of conditions so that domain errors could be reliably detected. Another related example of this approach is seen in the problem of feasible paths. The problem of path feasibility is in general undecidable, but if the collection of predicates along that path can be shown to be linear in the input variables, then this problem is decidable (using linear programming).

I believe the undecidability issue for general reliable test data generation is manifested in the coincidental correctness condition for domain testing in linearly domained programs. There is no way to decide whether a test point is "coincidentally correct" (in that the input point has been affected by an existing domain error) or that test point is indeed correct. Thus, all we can do with the undecidability problem for reliable testing is to reduce it to a simpler concept of "coincidental correctness" (which is still undecidable).

Domain testing is an example of a structural approach, which uses only information from the program to be tested. Thus it is clear why only domain errors can be reliably detected, since they are intimately related to the structure of the given program. In order to detect computation errors or missing path errors, we must obtain additional information, e.g., from the program specifications. This is precisely the approach of the researchers described in Section 5.

We have explicitly assumed that an "oracle" exists which can always determine whether a specific test case has been computed correctly or not. In reality, the programmer (or user) must make this determination, and the time spent examining and analyzing these test cases is a major factor in the high cost of software development. Weyuker [9-11] has recently criticized this "oracle" assumption on both theoretical and practical grounds, proposing several alternative approaches. We believe that it was partially because of our explicit emphasis of the oracle assumption as an essential component of the problem paradigm that this has emerged as a research issue.

3. Sufficient Test Sets for Path Oriented Testing

Although the number of required test points for each path in the domain strategy grows only linearly with the number of input variables and predicates along the path, the problem with this approach is that the number of paths grows in a highly combinatorial fashion and is potentially infinite. Moreover, any path-oriented strategy suffers from this basic problem.

In the definition of any automated path selection strategy, the questions which naturally arise are, "When does testing stop? At what point is it possible to point to a particular program construct and say that it has been sufficiently tested, i.e., no errors remain undetected?" In general, we know that this problem can be proven undecidable, but a programmer's intuition suggests that such claims should be possible after the selection of a small number of test paths, especially if we possess a strategy in which we have specific confidence in terms of its ability to detect certain types of errors in some construct along that path.

In references [15-17], Zeil and White have developed a vector space model for linearly domained programs, and this model has indicated substantive answers to these questions for this class of programs. It should be emphasized that this research and these results are essentially independent of the domain strategy, and only require a testing strategy which will reliably detect domain errors for linearly domained programs. However, the existence of the domain strategy with this degree of reliability allows us to investigate these issues.

In order to state these results more precisely, let us define these questions and concepts carefully. A set of paths is a sufficient set for a program construct if the failure to detect some error in that construct, using a reliable method of selecting data points along those paths, implies that this error would go undetected for any path through the program. In this definition, and throughout this discussion, we might be considering any program construct, but a most concrete construct for which we have such reliable methods is that of a predicate. We can then restate the questions more rigorously as:

a) After a number of paths have been tested which pass through the construct, what is the marginal advantage of testing another path?

b) Is there a point (before nearly all paths have been tested) at which we may say that no more paths need be chosen and tested through some program construct, i.e., that this construct has been sufficiently tested?

In order for us to characterize the minimal number of paths which must be tested, we first must clearly understand why multiple paths might be needed in order to detect an error in a construct (such as a predicate). The following examples show three different reasons why a single path may not detect an erroneous predicate. These are termed assignment blindness, equality blindness, and self-blindness, and represent a seemingly pathological set of values for variables along the path so that both the correct and the incorrect predicate evaluate to equal values.

By studying examples of this type, the vector space model examined in references [15-17] has yielded an insight as to how multiple paths through a single predicate can resolve these ambiguities due to various types of blindness. This vector space is composed of:

- one vector for each assigned program variable, for a total of n;
- one vector for each equality restriction on the path domain, at most m total, where m is the number of input variables.

Assigment Blindness

Correct	Incorrect
A = 1	A = 1
⋮	⋮
IF B > 0 THEN	IF B+A > 1 THEN
⋮	⋮

Equality Blindness

Correct	Incorrect
IF D = 2 THEN	IF D = 2 THEN
⋮	⋮
IF C + D > 3 THEN	IF C > 1 THEN
⋮	⋮

Self-Blindness

Correct	Incorrect
X = A	X = A
⋮	⋮
IF X-1 > 0	IF X+A-2 > 0
⋮	⋮

The results of this research which provide answers to questions a) and b) posed earlier in this section can be stated as follows:

> For any predicate in a linearly domained program, the smallest sufficient set of test paths will contain at most (m+n+1) paths, where m is the number of input variables and n the number of program variables. Moreover, if a set of paths have been tested which pass through the predicate of interest, a simple vector criterion described in references [15-17] will determine whether a proposed additional path is required to detect an error in that predicate.

This is a most satisfying result in that it is consistent with a programmer's intuition that only a reasonably small number of paths should be sufficient to reliably test any construct in a given computer program. We have assumed, however, linearly domained programs in order to obtain this result. The greatest difficulty with this approach is that paths which constitute the smallest sufficient set cannot be generated easily or efficiently. Rather, it is only after a set of paths are selected that the vector criterion can be applied. Research is continuing on this problem in order to devise heuristic methods to select the set of paths which are based upon ongoing experiments using the vector criterion.

4. Module Integration Testing

In references [4-5], Haley and Zweben have investigated the issues involved when a "correct" module which has been thoroughly validated is integrated into a larger program context. It is desired to maximally utilize the information that this module is correct in designing the integration testing strategy.

Two approaches to this question have been examined by Haley and Zweben. Since the goal is to detect errors in the module's input, one could simply require that input values to the module be examined, together with the normal output of the calling program. This technique is not new, as programmers often point out values of intermediate or temporary variables. However, it is difficult to know whether an intermediate program value is correct, and the programmer would usually be more interested in examining the final outputs of the calling program. This is actually a more complicated version of the "oracle assumption" discussed in Section 2, and illustrates the problem with this approach.

The second approach involves addressing the following two problems:

1) we may have failed to retest a predicate in the module that would have shifted for a particular error in the calling program (an integration domain error); or

2) an error in the calling program that produces an error in the module's input might not be passed to the module's outputs (and hence to the program's outputs) along these paths that are executed in the module during integration testing.

The solution proposed by Haley and Zweben is to do a limited amount of "retesting" during integration testing using a set of paths through the module which are sensitive to the two problems identified above. They refer to this act as the Integration Test Set for the module. This integration test set should meet two important criteria. First, it should be capable of detecting all of the integration testing errors which have been identified. Second, it should contain as few of the module's paths as possible to meet the first criterion.

For the detailed description of how these integration test sets are constructed by Haley and Zweben, see references [4-5]. However, they were strongly influenced in their work by the results of Zeil [15]. The key idea used in the investigation of these integration test sets is that if a module has m input variables with no inherent relationship among them, then there are only m independent ways in which an error can occur. It can be shown that for integration domain errors there are at most (m+1) "different" errors that can occur (see Haley [4]). For integration computation errors there are at most ((m*n)+1) "different errors", where n represents the number of module output variables.

This result is explored more fully by Haley and Zweben, but it illustrates the symbiotic effect when a research group is working on several related problems, and can make contributions to various research problems.

5. Specification Testing

One of the primary limitations of the domain testing strategy is that it is a structural approach, using only the program itself. A number of researchers are actively examining the possibility of generating test data from program specifications, especially to complement structural approaches such as path testing. Cartwright [1] has developed a very high level language with which to express program specifications, and since it is procedural, allows him to generate test data from the specifications. John Gourlay [3] has shown that specifications can be written using the flexibility and power of predicate calculus, and yet test data can be generated from specifications in this form. Richardson and Clarke [8] have also chosen to use a very high level language for program specification, and execute a path analysis of the specification which is then used to refine the program path testing partition.

Each of these research efforts promised to make a contribution to specification testing. Richardson and Clarke possess the distinct advantage of having constructed a working system, with which they can conduct experiments to evaluate their approach to specification testing. They have utilized several concepts from domain testing in their research, primarily for structural test data selection. John Gourlay is now a faculty member at Ohio State University, and we believe that some of the domain testing concepts will be useful to his research.

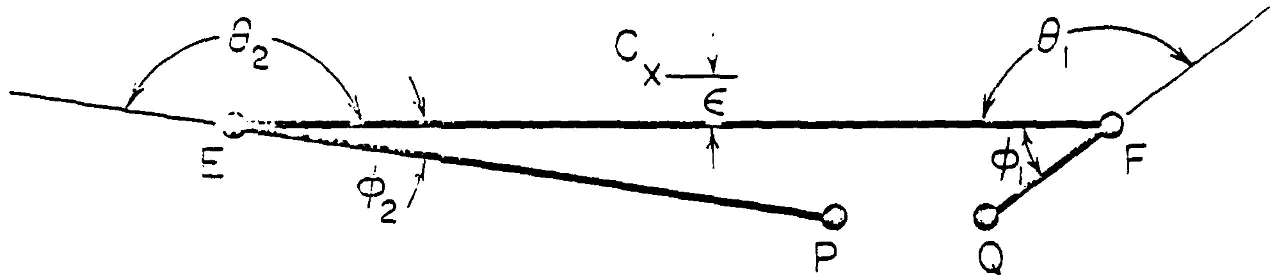
6. Examples

Many researchers have noted that some test data should be generated near boundaries of input domains defined by selected paths in the program. In domain testing we had simply worked out the details as to how this could be systematically implemented. Thus, examples could be generated illustrating how effectively domain errors can be detected by this approach.

An error analysis of domain testing was documented in reference [14]. It is interesting to observe that one extreme situation may cause problems for test point selection, sensitivity to potential errors in other borders, or applying domain testing in a discrete space rather than a continuous space. This extreme situation is encountered when two adjacent borders of the same domain are nearly parallel. Figure 1 shows this effect geometrically. In Figure 1a), we see very "sharp corners" being formed by adjacent borders which are nearly parallel. In Figure 1b), it will be very difficult to test border EF, since the OFF test point should be placed inside triangle EFT in order to satisfy all other inequalities.

Another interesting example can be seen from the various types of blindness shown in Section 3. The example for assignment blindness:

Correct	Incorrect
A = 1	A = 1
⋮	⋮
IF B > 0 THEN	IF (B+A) > 1 THEN
⋮	⋮



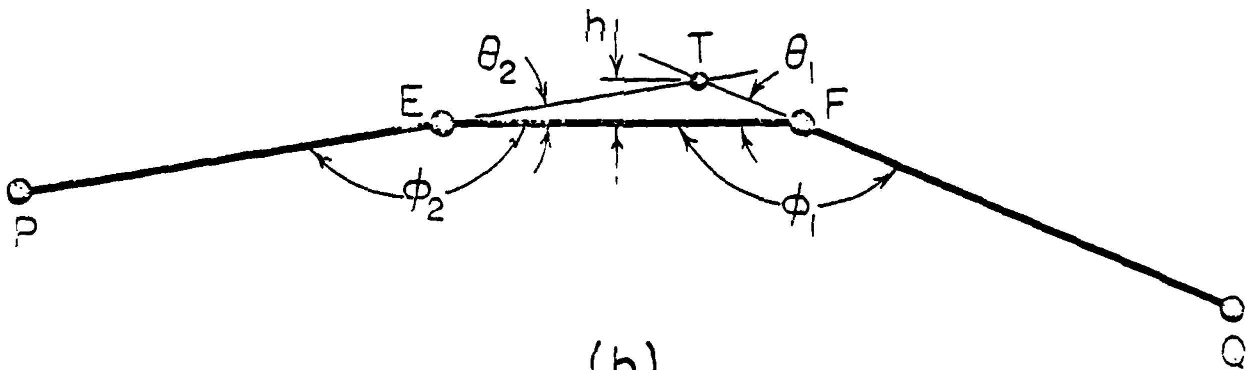


FIGURE 1 Adjacent Border Segments Which are Nearly Parallel

This shows how path testing may easily miss the detection of a predicate error due to the assignment A=1 along this path. Of course, we would expect another path to traverse through the predicate for which the assignment A=1 is not encountered, and thus the predicate error can be easily detected. Note that if all paths which contain the predicate also contain the assignment A=1, then the error cannot be detected at all; however, in this case we must decide if this "error" is of any consequence, since the specifications will always be met!

Another example of a similar type is encountered when the "correct" predicate

$$A + X > 0$$

is replaced by the "incorrect" predicate

$$2*A + 2*X > 0,$$

which is called "self-blindness" by Zeil [15]. Again, there is some question as to whether this is an error at all. The same effect is generated along each path containing this predicate, and thus the program specifications are met.

References

[1] Cartwright, Robert, "Formal Program Testing", Eighth Annual ACM Symposium on Principles of Programming Languages, 1981.

[2] Cohen, E.I., A Finite Domain-Testing Strategy for Computer Program Testing, Ph.D. Dissertation, Department of Computer and Information Science, The Ohio State University, June, 1978.

[3] Gourlay, John S., Theory of Testing Computer Programs, Ph.D. Dissertation, Department of Computer and Communication Sciences, The University of Michigan, 1981.

[4] Haley, A. and Zweben, S., "An Approach to Reliable Integration Testing", Technical Report TR-81-5, Computer and Information Science Research Center, The Ohio State University, May 1981.

[5] Haley, A. and Zweben, S., "Module Integration Testing", Computer Program Testing, B. Chandrasekaran and S. Radicchi, Eds., North-Holland Publishing Co., Amsterdam, 1981.

[6] Howden, William E., "Introduction to the Theory of Testing" in: Miller and Howden (eds), Tutorial: Software & Validation Techniques (IEEE Computer Society, Catalog No. EHO 138-8, 1978, 16-19.

[7] Howden, W.E., "Reliability of the Path Analysis Testing Strategy", IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, Sept. 1976, 208-215.

[8] Richardson, Debra J. and Clarke, Lori, "A Partition Analysis Method to Increase Program Reliability", Proceedings 5th International Conference on Software Engineering, San Diego, California, March 9-12, 1981.

[9] Weyuker, Elaine, "The Oracle Assumption of Program Testing", Proceedings of the Thirteenth International Conference on System Sciences, Honolulu, Hawaii, January 1980.

[10] Weyuker, Elaine and Danis, M., "Pseudo-Oracles for Non-Testable Programs", Proceedings ACM National Conference, Los Angeles, November 1981.

[11] Weyuker, Elaine, "On Testing Nontestable Programs", Department of Computer Science Technical Report 025, Courant Institute of Mathematical Sciences, New York University, New York, New York, October 1980.

[12] White, L.J., Cohen, E.I., and Chandrasekaran, B., "A Domain Testing Strategy for Computer Program Testing", Technical Report TR-78-4, Computer and Information Science Research Center, The Ohio State University, August 1978.

[13] White, L.J. and Cohen, E.I., "A Domain Strategy for Computer Program Testing", IEEE Transactions on Software Engineering, Vol. SE-6, No. 3, May 1980, 247-257.

[14] White, L.J., Teng, F.C., Kuo, H.C., and Coleman, D.W., "An Error Analysis of the Domain Testing Strategy", Technical Report 78-2, Computer and Information Science Research Center, The Ohio State University, August 1978.

[15] Zeil, S.J., Selecting Sufficient Sets of Test Paths for Program Testing, Ph.D. Dissertation, Department of Computer and Information Science, The Ohio State University, September 1981; Technical Report TR-81-10, Computer and Information Science Research Center, October 1981.

[16] Zeil, S.J. and White, L.J., "Sufficient Test Sets for Path Analysis Testing Strategies", Technical Report TR-80-6, Computer and Information Science Research Center, The Ohio State University, July 1980.

[17] Zeil, S.J. and White, L.J., "Sufficient Test Sets for Path Analysis Testing Strategies", Proceedings 5th International Conference on Software Engineering, San Diego, California, March 9-12, 1981, 184-191.

2.1.2. Sufficient Testing

As we mentioned earlier in this report, the issue here is one of deciding when a program or a construct has been tested sufficiently in order to enable a conclusion that no errors remain. The paper that follows is a technical report that presents work by Zeil and White relating to this question. A condensed version of this appeared in *Proc. 5th Intern. Conf. on Software Engineering*.

(OSU-CISRC-TR-80-6)

SUFFICIENT TEST SETS FOR PATH ANALYSIS TESTING STRATEGIES

By

Steven J. Zeil and Lee J. White

Work performed under
Contract F49620-79-0152, Air Force Office of Scientific Research

The Computer and Information Science Research Center
The Ohio State University
Columbus, Ohio 43210

July 1980

ABSTRACT

This report presents a new method for selecting paths to test when path analysis testing strategies are employed. This method carefully analyzes the types of errors which can be detected by testing along a single path, and what type of errors might escape detection. This research provides an approach to decide whether an additional path should be tested when a number of paths have already been tested, or whether no additional information can be gained by testing this path. Another result is to characterize the situation when no more paths need be chosen through some program construct, i.e., that it has been "sufficiently tested".

TABLE OF CONTENTS

PAGE

Abstract ii

Preface iii

I. Introduction 1

II. The Domain Testing Strategy 4

III. A Model of Linearly Domained Programs 7

IV. Sufficient Testing for Predicate Errors 13

V. Conclusions 24

References 26

Sufficient Test Sets for Path Analysis Testing Strategies

Steven J. Zeil and Lee J. White

I. Introduction

Recent years have witnessed the proposal of a number of methods for automating portions of the software testing effort. Many of these methods are forms of path analysis strategies, where the process of testing is treated as two operations [1,4,6,7]:

1. selection of a path or set of paths along which testing is to be conducted;
2. selection of input data to serve as test cases which will cause the chosen paths to be executed.

Work has proceeded on such methods despite the lack of a theoretical basis for the justification or evaluation of such methods. Little is known regarding the proper methods of selecting of test paths and data. Indeed, for general programs these problems are known to be unsolvable. For selected classes of programs, however, it is possible to implement reliable strategies of selecting test data for a given path to detect certain types of errors. Section II of this paper summarizes one such method, the domain testing strategy, which detects errors in program predicates for a large class of programs, referred to as "linearly domained programs" [7].

Even with a reliable method of selecting test data for a given path, the fact remains that certain errors may escape detection no matter what data is used along that path. A good testing strategy must therefore select a set of paths which collectively account for all possible errors.

Frequently an attempt is made to achieve one of the following measures of coverage [1,3,4]:

1. each statement is executed at least once;
2. each branch is executed at least once;
3. each path is executed at least once.

Examples can be easily constructed to show that the first two measures are insufficient to guarantee error detection, but to infer that a good testing strategy must execute all paths is hardly practical. The presence of a simple DO-WHILE construct may introduce an infinite number of feasible paths. Even if an arbitrary limit is placed on the number of loop iterations, the number of available paths tends to grow exponentially as program complexity increases.

All questions of practicality aside, such a claim runs counter to the intuition of the typical programmer who is quite willing to infer the correctness of his program from a small, finite number of test paths. It is the goal of this paper to show that, when testing for errors in program predicates, this confidence is not misplaced. Specifically, the questions to be addressed are:

1. "After a number of paths have been tested, what is the marginal advantage of choosing yet another test path?"
2. "Is there a point at which we may say that no more paths need be chosen through some program construct, i.e., that it has been sufficiently tested?"

A set of paths shall be considered a <u>sufficient set</u> for a program construct if the failure to detect some error in that construct, using a reliable

method of selecting data points along those paths, implies that this error would go undetected for any path through the program.

The domain strategy selects a reliable set of points at the cost of restricting the permitted functional forms for the program predicates. In this paper, it will be shown that under a similar restriction, direct answers can be supplied for the above questions. In Section III of this paper, a model of linearly domained programs is developed. This model will be employed in Section IV to investigate the effect of predicate errors on control flow. Expressions describing the value of a proposed test path will be developed and it will be shown that the number of test paths required to detect errors in a given predicate of a linearly domained program has a small, finite bound. This bound is linear in the number of program variables and inputs and is independent of the complexity of the program's control flow.

II. The Domain Testing Strategy

Computer programs contain two types of errors which have been described as computation errors and domain errors [5]. A domain error occurs when a specific input follows the wrong path due to an error in the control flow of the program. A path contains a computation error when a specific input follows the correct path, but an error in some assignment statement causes the wrong function to be computed for one or more output variables. The domain strategy has been designed to detect domain errors. Under the proper conditions, this strategy is reliable for any given path [2,7].

There are limitations inherent in any testing strategy, and these also constrain the proposed domain strategy. One such limitation might be termed "coincidental correctness", which occurs when a specific test point follows an incorrect path, and yet the output variables coincidentally are the same as if that test point were to follow the correct path. This test point would then be of no assistance in the detection of the domain error which caused the control flow change. No test generation strategy can circumvent this problem. Another inherent testing limitation has been previously identified as a missing path error, in which a required predicate does not appear in the given program to be tested [5]. Especially if this predicate were an equality, no testing strategy could systematically determine that such a predicate should be present.

The control flow statements in a computer program partition the input space into a set of mutually exclusive domains, each of which corresponds to a particular program path and consists of input data points which cause that path to be executed. The testing strategy generates test points to examine

the boundaries of a domain to detect whether a domain error has occurred, as either one or more of these boundaries will have shifted or else the corresponding predicate relational operator has changed. If test points can be chosen within ϵ of each boundary, the strategy has been shown to be reliable in detecting domain errors of magnitude greater than ϵ, subject to the following assumptions:

(1) coincidental correctness does not occur;
(2) missing path errors do not occur;
(3) predicate interpretations are linear in the input variables.

Assumptions (1) and (2) have been shown to be inherent to the testing process and cannot be completely eliminated. Although assumption (3) appears to be severely limiting, some evidence exists to indicate that it may hold for a surprisingly large class of programs. Besides indirect evidence from software metric studies, a study by Cohen of fifty COBOL programs taken from production data processing found only one predicate out of 1225 to be nonlinear [2]. The domain testing method has been shown to be applicable to nonlinear boundaries, but the number of test points may become inordinate and there are complex problems associated with processing nonlinear boundaries in higher dimensions.

Next let us consider three further assumptions:

(4) the input space is continuous;
(5) predicates are simple;
(6) adjacent domains compute different functions.

The continuous input space assumption is not really a limitation of the present testing method, but allows the parameter ϵ to be chosen arbitrarily small. An error analysis for discrete spaces has shown the strategy to be viable as long as the size of the domain is not comparable to the discrete resolution of the space.

If assumptions (5) and (6) are imposed, the testing strategy is considerably simplified, as no more than one domain need be examined at one time in order to select test points. Moreover, the number of test points required to test each domain grows linearly with the dimensionality of the input space.

Any program satisfying the six constraints given above will henceforth be referred to as a <u>linearly domained</u> program.

The analysis of linearly domained programs which follows this section is not dependent upon the domain testing strategy, although some form of reliable means of selecting test points for a given path is assumed. The domain strategy has been discussed here as an example, demonstrating that reliable strategies can be constructed.

III. A Model of Linearly Domained Programs

It is the goal of this paper to provide a mathematical justification for some of the intuitive arguments in the preceeding sections. Towards this end we now present a model for the behavior of linearly domained programs. In this model the program itself is represented as a static set of transformations and predicates, while the execution state is represented using the dynamic attributes of environment, path, and constraints.

The central element in this model is the environment. Properly speaking, the environment of a program represents the values of all variables at any point in the program's execution. However, since the subject of this analysis is the detection of domain errors, we shall restrict our representation of the environment to those variables and other factors which may affect the flow of control. Then the environment may be represented as the following vector:

$$\bar{v} = (1, x_1, \ldots, x_m, y_1, \ldots, y_n)^T$$

The y_i represent those program variables which may directly or indirectly affect the program control flow through their effects on the evaluation of program predicate expressions. The x_i represent the values of input data. It is convenient for purposes of illustration to treat these as special variables whose values are established prior to execution and held fixed thereafter, although in practice no such special variables need exist. The first element of the environment vector is held to the constant "1" as a notational convenience so that computations involving constants as well as variables might be expressed in a uniform manner. Initially, only this constant term and the x terms are considered to be defined. The program must initialize the program variables as functions of these terms.

The components of the program itself can then be described in terms of their interactions with the environment vector. A program is considered as a set of pairs of the form (C_i, T_i) where C_i is a computation or transformation to be applied to the current environment to generate the new environment and T_i is a predicate which is applied to the new environment. The next (C_i, T_i) pair to be used is determined by the result of the application of T_i.

The process of executing a program consists of determining a path $P = (p_o, p_1, \ldots, p_k)$ where the p_i are the indices of the (C_i, T_i) pairs which are to be successively applied to the environment. As a convention, we shall let $p_o = \emptyset$ designate the start of the program.

The term subpath will be used to designate a path which does not begin with $p_o = \emptyset$ or does not end at a valid HALT statement, that is, a path which does not describe a complete execution of the program. An initial subpath shall be defined as a subpath beginning at the start of the program, for which $p_o = \emptyset$.

For linearly domained programs, after any step along such a path, the old and new environments will be linearly related. The computations C_i may therefore be treated as linear transformations. Taking C_i as a matrix, the k<u>th</u> step along a path P causes the environment to undergo the transformation:

$$\overline{v}_k = C_{p_{k-1}} \overline{v}_{k-1}$$

The environment after k steps along path p is therefore given by:

$$\overline{v}_k = C_{p_k} \ldots C_{p_1} C_{p_o} \overline{v}_o$$

where $\overline{v}_o$ is the initial environment. It will often be convenient to represent this long string of matrices as a single matrix

$$C_P = C_{P_k} \cdots C_{P_1} C_{P_o}$$

representing the total transformation along subpath P.

Since the predicates in a linearly domained program must be linear expressions, the T_i may be treated as vectors such that the scalar product $\overline{T}_{P_k} \cdot \overline{v}_{k+1}$ is compared with zero to determine the next index p_{k+1}. (The mechanism by which the next index is selected has deliberately been left unspecified as it is not of importance to this analysis.)

Figure 1 shows a short program segment and its representation under this model. If we treat the variables A and B as restricted input variables in the sense described earlier, then the environment vector has six components $(1,x_1,x_2,y_1,y_2,y_3)^T$ corresponding to $(1,A,B,S,T,U)^T$. Let the values in the input stream for A and B be designated as "a" and "b". Then the initial environment $\overline{v}_o$ is $(1,a,b,?,?,?)^T$ where "?" indicates an undefined value.

Two initial subpaths are available up to location PRED depending on the result of the test for A>2, $P_A = (\emptyset, 1, 3)$ and $P_B = (\emptyset, 2, 3)$. After the first step along either path, the new environment would be

$$\overline{v}_1 = C_o\overline{v}_o = (1,a,b,?,1,a)^T$$

Then applying the predicate T_o involves comparing the values $\overline{T}_o \cdot \overline{v}_i = (-2,1,\emptyset,\emptyset,\emptyset,\emptyset)\ (1,a,b,?,1,a)^T = -2 + a$ to zero. Note that the values in $\overline{v}_1$ for S, T, and U and the expression for $\overline{T}_o \cdot \overline{v}_1$ do indeed correspond to the results expected of the program at this point.

Program	i	C_i	T_i
READ A,B; T = 1; U = A; IF A>2 THEN	Ø	$\begin{bmatrix} 1&0&0&0&0&0\\ 0&1&0&0&0&0\\ 0&0&1&0&0&0\\ 0&0&0&1&0&0\\ 1&0&0&0&0&0\\ 0&1&0&0&0&0 \end{bmatrix}$	$\begin{bmatrix} -2\\ 1\\ 0\\ 0\\ 0\\ 0 \end{bmatrix}$
T = 2 * U;	1	$\begin{bmatrix} 1&0&0&0&0&0\\ 0&1&0&0&0&0\\ 0&0&1&0&0&0\\ 0&0&0&1&0&0\\ 0&0&0&0&0&2\\ 0&0&0&0&0&1 \end{bmatrix}$	$\begin{bmatrix} 0\\ 0\\ 0\\ 0\\ 0\\ 0 \end{bmatrix}$

ELSE
T = 2*A + 2*B;
U = U + B;
END IF
	2	$\begin{bmatrix} 1&0&0&0&0&0\\ 0&1&0&0&0&0\\ 0&0&1&0&0&0\\ 0&0&0&1&0&0\\ 0&2&2&0&0&0\\ 0&0&1&0&0&1 \end{bmatrix}$	$\begin{bmatrix} 0\\ 0\\ 0\\ 0\\ 0\\ 0 \end{bmatrix}$

S = 1; PRED: IF U>B THEN ...	3	$\begin{bmatrix} 1&0&0&0&0&0\\ 0&1&0&0&0&0\\ 0&0&1&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&1&0\\ 0&0&0&0&0&1 \end{bmatrix}$	$\begin{bmatrix} 0\\ 0\\ -1\\ 0\\ 0\\ 1 \end{bmatrix}$

Figure 1: Model Representation of Sample Program

Correct Code	Incorrect Code
IF D=1 THEN	IF D=1 THEN
. . .	. . .
IF C+D > 1 THEN	IF C > Ø THEN
. . .	. .

Figure 2: Equality Blindness

Completing the execution along both subpaths, define C_A and C_B:

$$C_A = C_3C_1C_o = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad (A>2)$$

$$C_B = C_3C_2C_o = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 2 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \end{pmatrix} \qquad (A\le 2)$$

so that $C_A\bar{v}_o = (1,a,b,1,2a,a)^T$

$C_B\bar{v}_o = (1,a,b,1,2a+2b,a+b)^T$.

Taking the product of the transformation matrices along some path is equivalent to symbolically executing along that path. The total transformation matrix represents the equivalent assignments along that path. But not all such paths are valid. An initial subpath $P = (\emptyset,p_1,\ldots,p_k)$ shall be called a <u>testable subpath</u> if there exists a subpath $P' = (p_{k+1},\ldots,p_h)$ such that

1. P' ends with a HALT statement;
2. There exists some input value causing the path $P'' = (\emptyset,p_1,\ldots,p_k,p_{k+1},\ldots,p_h)$ to be executed;
3. The predicate $\bar{T}_{p_k}$ is not implied by the conjunction of other predicates on P".

One final item remains to be modeled. Every predicate encountered along a path places restrictions on the legal set of input values for that path. However the constraints imposed by equality predicates are qualitatively different from those imposed by inequalities, since a valid equality predicate

reduces the dimension of the space of legal inputs x_i. In recognition of this, there is associated with each testable subpath P_A a set of restriction vectors $\overline{r}_i^A$, for $\emptyset<i\leq k_A$, such that if $\overline{v}_o$ is an initial environment which might cause path P_A to be executed, then

$$\overline{r}_i^A \cdot \overline{v}_o = \emptyset, \ \emptyset<i\leq k_A$$

To summarize, this model represents linearly domained programs in terms of computation-predicate pairs $(C_i,\overline{T}_i)$ with execution being described in terms of environment $\overline{v}$, paths P, and equality restrictions $\overline{r}_i$. In the next section this model will be employed to investigate the detection of errors in program predicates.

IV. Sufficient Testing for Predicate Errors

In the definition of any automated path selection strategy, a question which must arise is, "When does testing stop? At what point is it possible to point to a particular program construct and say that it has been sufficiently tested - no errors can remain undetected?" A programmer's intuition suggests that such claims should be possible after the selection of a small number of test paths. This intuitive claim may be verified for predicates in a linearly domained program using the model presented in the previous section.

Even given a reliable method of selecting critical test data for a given path, certain predicate errors will escape detection. This is inherent in the nature of path analysis testing. The goal is to choose a combination of paths so as to collectively eliminate all such errors.

To characterize these undetected errors, consider a program where for some pair $(C_i,\overline{T}_i)$ the predicate is replaced by an erroneous predicate $\overline{T}_i'$ such that

$$\overline{T}_i' = \overline{T}_i + \alpha\hat{e} \qquad \alpha \neq \emptyset$$

$\hat{e}$ is a unit vector giving the "direction" of the error and α is a scalar giving the magnitude of the error. Let P_A be a testable subpath ending with $(C_i,\overline{T}_i')$. The environment after executing along P_A will be $\overline{v}_A = C_A\overline{v}_o$ where C_A is the total transformation along P_A. Assuming that adjacent domains compute different functions and coincidental correctness does not occur, a reliable strategy for selecting test data will be able to detect the erroneous predicate if and only if

$$\overline{T}_i \cdot \overline{v}_A \neq \overline{T}_i' \cdot \overline{v}_A$$

Expanding $\overline{T}_i'$ and $\overline{v}_A$ this becomes

$$\overline{T}_i \cdot \overline{v}_A \neq \overline{T}_i \cdot \overline{v}_A + \alpha\hat{e} \cdot \overline{v}_A$$

$$\hat{e}^T C_A \overline{v}_o \neq \emptyset$$

Therefore if $\hat{e}^T C_A \overline{v}_o = \emptyset$ for all $\overline{v}_o$ in the domain of the path P_A, then the error $\hat{e}$ will go undetected. Consider the various cases which may force this expression to zero:

1. $C_A \overline{v}_o = \overline{\emptyset}$
2. $\hat{e}^T C_A = \overline{\emptyset}^T$
3. $\hat{e}^T C_A \neq \overline{\emptyset}^T$, $\hat{e}^T C_A \overline{v}_o = \emptyset$ for all $\overline{v}_o$ in the *domain of* P_A.

The first case is clearly impossible since $\overline{v}_A = C_A \overline{v}_o$ will always have a constant "1" in its first position, as constants may not be reassigned new values.

If $\hat{e}^T C_A = \overline{\emptyset}^T$ then transposing to get $C_A^T \hat{e} = \overline{\emptyset}$ indicates that this is an eigenvalue problem $C_A^T \hat{e} = \lambda \hat{e}$ with $\lambda = \emptyset$. The solution to this problem can be found by examining the structure of the individual C_i.

Each matrix C_i may be partitioned into the form

$$C_i = \left(\begin{array}{c|c} Q & R \\ \hline S & T \end{array}\right)$$

where Q is (m+1) by (m+1) and T is n by n.

The matrix Q maps the inputs and constants from the old environment into the new environment, and so must be the identity matrix I. R maps the variables of the old environment onto the inputs and constants in the new environment.

Such assignments are forbidden and so R must be entirely zero. S, mapping the old inputs and constants into the new variables, may contain any real values. T maps the old variables into the new variables. This mapping is unrestricted for all C_i except C_o, the initial assignments where all variables are initialized in terms of inputs and constants. For C_o the component T must be entirely zero, so that

$$C_o = \left(\begin{array}{c|c} I & \emptyset \\ \hline S_o & \emptyset \end{array}\right)$$

$$C_i = \left(\begin{array}{c|c} I & \emptyset \\ \hline S_i & T_i \end{array}\right) \qquad i \neq \emptyset$$

and for any initial subpath P_A,

$$C_A = C_{p_k} C_{p_{k-1}} \cdots C_{p_1} C_o = \left(\begin{array}{c|c} I & \emptyset \\ \hline S_A & \emptyset \end{array}\right)$$

$$C_A^T = \left(\begin{array}{c|c} I & S_A^T \\ \hline \emptyset & \emptyset \end{array}\right)$$

Now the solution to the eigenvalue equation $C_A^T\bar{z} = \lambda\bar{z}$ is given by inspection. If $\hat{\mu}_i$ are the vectors forming the columns of the identity matrix of the same dimension as C_A, then

$$C_A^T\hat{\mu}_i = \hat{\mu}_i \qquad i = 1\ldots(m+1) \tag{1}$$

$$C_A^T\hat{\mu}_i = \bar{c}_i \qquad i = (m+2)\ldots(m+n+1) \tag{2}$$

where $\bar{c}_i$ is the ith row of C_A. But the $\bar{c}_i$ for $m+2 \leq i \leq m+n+1$ are linear combinations

of the $\hat{\beta}_i$ for $1 \le i \le m+1$. Therefore

$$C_A^T \hat{\beta}_i = \hat{\beta}_i \qquad 1 \le i \le m+1 \qquad \lambda = 1 \tag{3}$$

$$C_A^T (\bar{c}_i - \hat{\beta}_i) = \bar{\emptyset} \qquad m+1 < i \le m+n+1 \qquad \lambda = \emptyset \tag{4}$$

describes the eigenvalues and eigenvectors of C_A^T.

Thus an error $\hat{e}$ will go undetected if

$$\hat{e} \in \text{null-space}(C_A^T) = \underset{i}{\text{span}}(\bar{c}_i - \hat{\beta}_i) \qquad m+1 < i \le m+n+1 \tag{5}$$

This vector space has a simple interpretation in terms of symbolic evaluation. In Figure 1 consider the subpath leading to PRED for the case A>2. Since T=2a along this path, one error satisfying the above criterion is $\hat{e} = (\emptyset, 2, \emptyset, \emptyset, -1, \emptyset)^T$. Adding this to the vector representation of $\bar{T}_3$ would give a vector equivalent to the predicate "IF U+2*A-T>B". For any data which causes this path to execute, this erroneous form will be indistinguishable from "IF U>B" since the added term "2*A-T" will evaluate to zero. This behavior is termed "assignment blindness", because it results solely from the assignment statements encountered along the test path.

Finally if $\hat{e}^T C_A \ne \bar{\emptyset}^T$ but $\hat{e}^T C_A \bar{v}_o = \emptyset$ for all $\bar{v}_o$ in the domain of P_A, then the error still goes undetected but assignment blindness cannot be a factor since $\hat{e}^T C_A \ne \bar{\emptyset}^T$.

Let $\bar{u} = C_A^T \hat{e}$. Then $\bar{u} \cdot \bar{v}_o = \emptyset$, but neither $\bar{u}$ nor $\bar{v}_o$ can be a zero vector. If the set of legal $\bar{v}_o$ for the path P_A forms a space of dimension m, there will always exist some $\bar{v}_o$ in that domain such that $\bar{u} \cdot \bar{v}_o \ne \emptyset$. However, equalities restrict the $\bar{v}_o$ to a hyperplane of that space of inputs. If $\bar{u}$ is

orthogonal to that hyperplane, then $\bar{u} \cdot \bar{v}_o = \emptyset$ for all legal $\bar{v}_o$. This implies that some form of "equality blindness" exists to complement assignment blindness. Since $\bar{r}_i^A \cdot \bar{v}_o = \emptyset$, $i=1..k_A$ represent these equality restrictions, it is apparent that the necessary condition for the inability to detect an error due to equality blindness is

$$\bar{u} = C_A^T \hat{e} \in \underset{i}{\text{span}}(\bar{r}_i^A). \qquad (6)$$

An example of equality blindness is given in Figure 2, where the two expressions for the second predicate will be indistinguishable for any test path where the first predicate is true.

The above analysis of the conditions under which $\hat{e}^T C_A \bar{v}_o$ would go to zero has identified a number of isolated vectors representing undetectable errors for the path being tested. Clearly any linear combination of these errors will also go undetected, implying that the total undetected space is described by the span of these vectors. The total undetected space would then be given by the span of the null-space of C_A^T and of the set $\{\hat{e}: C_A^T \hat{e} \in \underset{i}{\text{span}}(\bar{r}_i^A)\}$.

Equation (5) describes assignment blindness directly in terms of the assignments performed along a path. In contrast, the description of equality blindness requires the solution of (6). Such a procedure would be, at best, awkward. This indirect method of specifying the characteristic vectors for equality blindness can be simplified. When the spaces for assignment and equality blindness are combined in this way, there may be some overlap. Choose any $\hat{e}$ which is subject to equality blindness. Decomposing $\hat{e}$ into its components,

$$\hat{e} = \sum_{i=1}^{m+n+1} \beta_i \, \Omega_i. \qquad (7)$$

Since we have specified that this $\hat{e}$ is subject to equality blindness,

$$C_A^T \hat{e} \in \operatorname*{span}_i(\bar{r}_i^A).$$

So there must exist a set $\{\gamma_i\}$ such that

$$C_A^T \hat{e} = \sum_i \gamma_i \bar{r}_i^A.$$

Referring back to the expressions for $C_A^T \Omega_i$ given in (1) and (2):

$$C_A^T \hat{e} = \sum_{i=1}^{m+1} \beta_i \Omega_i + \sum_{i=m+2}^{m+n+1} \beta_i \bar{c}_i = \sum_i \gamma_i \bar{r}_i^A$$

$$\sum_{i=1}^{m+1} \beta_i \Omega_i = \sum_i \gamma_i \bar{r}_i^A - \sum_{i=m+2}^{m+n+1} \beta_i \bar{c}_i$$

$$\sum_{i=1}^{m+1} \beta_i \Omega_i + \sum_{i=m+2}^{m+n+1} \beta_i \Omega_i = \sum_i \gamma_i \bar{r}_i^A - \sum_{i=m+2}^{m+n+1} \beta_i (\bar{c}_i - \Omega_i)$$

The expression on the left is the decomposition of $\hat{e}$ and that on the right is a linear combination of the equality restrictions and of the assignment blindness vectors. Thus $\hat{e}$ is subject to equality blindness if and only if

$$\hat{e} \in \text{span}[\text{null-space}(C_A^T), \{\bar{r}_i^A\}]$$

Clearly this also holds for errors subject to assignment blindness. This leads to the following theorem:

Theorem 1. Characterization of Undetected Predicate Errors

Let P_A be a testable subpath in a linearly domained program. Then an error $\hat{e}$ in the final predicate of P_A will be undetectable if and only if

$$\hat{e} \in \text{span}[\text{null-space}(C_A^T), \{\bar{r}_i^A\}].$$

Although the examples of assignment and equality blindness given above may seem trivial or may appear to involve awkward or unlikely errors, it is important to note that any linear combination of these errors will also go undetected. Such combinations can involve simple expressions, yet may not be apparent from an inspection of the program.

The existence of a characterization theorem suggests a return to the question posed earlier, "After testing several paths, what is the marginal advantage of testing still another path?" Since the undetected errors are described by a well-defined vector space, a new proposed path will form a useful test only if some portion of the (previously) untested space is detectable along the new path.

Theorem 2. Path Rejection Criteria

If a set $K = \{P_\xi\}$ of testable subpaths ending at $\overline{T}_j$ has been previously tested, then a proposed testable subpath P_A also ending with $\overline{T}_j$ need not be tested if and only if

$$\bigcap_{\xi \in K} \text{span}[\text{null-space}(C_\xi^T), \{\overline{r}_i^\xi\}] \subseteq \text{span}[\text{null-space}(C_A^T), \{\overline{r}_i^A\}].$$

A set of previously tested paths may leave a certain error space unchecked. If $\hat{e}$ is in that space, and $\hat{e}$ is detectable over subpath P_A, then any errors in the untested space which have $\hat{e}$ as a component will be detected. Testing along such a path will reduce the dimension of the undetected error space by at least one. This naturally suggests the following corollary:

Corollary 1. A set of testable subpaths $K = \{P_\xi\}$ all ending with $\overline{T}_j$ is sufficient for $\overline{T}_j$ if

$$\bigcap_{\xi \in K} \text{span}[\text{null-space}(C_\xi^T), \{\overline{r}_i^\xi\}] = \phi$$

Normally certain errors will remain undetectable no matter what test paths are used. These may occur because no options exist to certain assignments or equalities. For example, in Figure 1 the statement "S=1;" immediately before PRED means that no path can detect errors in PRED of the form $\hat{e} = (1,\emptyset,\emptyset,-1,\emptyset,\emptyset)^T$ which would result in predicates like "IF U+S>B+1". Alternatively, some errors will go undetected because some functional relationship is preserved along all paths. In Figure 1 the path for A>2 transforms the environment to $C_A^T \bar{v}_o =$ $(1,a,b,1,2a,a)^T$. Applying the rules for assignment blindness to this environment shows that testing along this path misses errors in PRED involving the expressions"A-U" and "2*A-T". The path for A≤2 with an environment $C_A^T \bar{v}_o =$ $(1,a,b,1,2a+2b,a+b)^T$ is blind to error expressions "A+B-U" and "2*A+2*B-T". But neither path will detect the expression "T-2*U" if it is added to the predicate, because for both paths that expression is a linear combination of the undetected errors.

Such errors are undetectable for any test path and hence for any input data. Consequently they have no real bearing upon the correctness of the program, but they do complicate the problem of judging when a predicate has been sufficiently tested. Although we may reasonably believe a set of paths to be nearly sufficient because they reduce the dimension of the untested space to a small number, the smallest possible dimension of that space may not be known.

In this context, the value of a proposed test path may be measured by the number of dimensions it would subtract from the total untested space.

The computations necessary to find the null space of C_A^T and the $\bar{r}_i^A$ are not as difficult as the notation might suggest. Both are directly derivable

from a symbolic execution without requiring an explicit construction of the C_i matrices.

Furthermore, in most cases where a subpath is rejected under Theorem 2, any extensions of that path may also be rejected in favor of extension of already tested paths. This rule applies whenever the extensions do not involve additional equalities and the extended paths remain testable.

Theorem 3. Concatenation Rule

Given a set of subpaths $K = \{P_\xi\}$ and a subpath P_A satisfying the terms of Theorem 2, define $K' = \{P'_\xi\}$ and P'_A as the initial subpaths formed by concatenating the P_ξ and P_A with subpath P_B. Suppose that $\{P'_\xi\}$ and P'_A are testable and no additional equalities are encountered in P_B.

Then if testing is performed on $\{P'_\xi\}$, $P_{A'}$ need not be tested.

Proof: Let $D(\xi)$ and $D(A)$ designate the domains of paths P_ξ and P_A. Then by Theorem 2,

$$[\forall \xi \in K, \forall \bar{v}_o \in D(\xi), \hat{e}^T C_\xi \bar{v}_o = \emptyset] \Rightarrow [\forall \bar{v}_o \in D(A), \hat{e}^T C_A^T \bar{v}_o = \emptyset] \qquad (8)$$

Assume by way of contradiction that P'_A must be tested, meaning that there exists some error $\hat{f}$ such that

$$\forall \xi' \in K', \ \forall \bar{v}_o \in D(\xi'), \ \hat{f}^T C_{\xi'} \bar{v}_o = \emptyset \qquad (9)$$

and

$$\exists \bar{v}_o \in D(A'), \ \hat{f}^T C_{A'} \bar{v}_o \neq \emptyset \qquad (10)$$

Noting that

$$D(A) = D(A') \qquad D(\xi) = D(\xi')$$

$$C_{A'} = C_B C_A \qquad C_{\xi'} = C_B C_\xi \ ,$$

let $\quad \bar{e} = C_B^T \hat{f}.$

Substituting for $\hat{f}$ and the A' and ξ' terms in (9) and (10):

$$\forall \xi \epsilon K, \ \forall \bar{v}_o \epsilon D(\xi), \ \bar{e}^T C_\xi \bar{v}_o = \emptyset$$

and

$$\exists \bar{v}_o \epsilon D(A), \ \bar{e}^T C_A \bar{v}_o \neq \emptyset$$

But this contradicts the statement of Theorem 2 in equation (8) which is given as true. Consequently the assumption that such an $\hat{f}$ exists fails and the theorem is proven.

At this point we have characterized those predicate errors which escape detection for a given test path and have shown that the value of a test path lies in its ability to reduce the space of potentially undetected errors. We have yet to justify the claim that a small, finite set of paths will be sufficient for detecting predicate errors. This is accomplished in the final theorem.

Theorem 4. Minimal Set for Sufficient Testing

A minimal set of subpaths sufficient for testing a given predicate in a linearly domained program will contain at most m+n+1 subpaths, where m is the number of input values and n the number of program variables.

After a single path has been tested, the untested error space due to assignment blindness is of dimension n, and the space due to equality blindness is at most dimension m. In constructing a minimal sufficient set of test paths, any subpath which fails to reduce the dimension of the total untested space by at least one would be rejected under Theorem 2. So after testing two paths the dimension of the total untested error space is at most m+n-1. Continuing in this fashion it is clear that a minimal sufficient test set can have at most m+n+1 paths.

The importance of this theorem is that it shows that a finite number of test paths will suffice for a wide class of programs. This limit is linear in the number of inputs and variables, so it should not grow inordinately large. Furthermore in most cases this limit should prove to be unnecessarily pessimistic, for several things may act to reduce the actual number of paths required. If the number of equalities is small, the dimension of the initial untested space will be reduced. If paths with widely different computations are used, the untested space due to assignment blindness can be reduced by far more than one dimension at a time. Even more important, however, are the implications of the concatenation rule. The chosen test subpaths, when extended to full paths from start to halt, should pass through a number of predicates. The concatenation rule then suggests that a sufficient or nearly sufficient set of paths for a predicate early in the program may also serve as a nearly sufficient set for later predicates, so that m+n+1 separate paths need not be formed for each individual predicate.

V. Conclusions

In linearly domained programs, the program predicates and the computations affecting control flow are linear in the input variables. Although linearity itself yields considerable simplification, another implication of this assumption is conceptually more important. Restricting predicate interpretations to a well-behaved functional class makes possible the description of the infinite set of possible predicate errors using a small finite set of linearly independent errors.

In the above sections we have used this approach to characterize those predicate errors which must escape detection for a given test path in a linearly domained program. This characterization has led to criteria for determining whether a proposed test path is capable of detecting any errors not already revealed by previous tests. These criteria are directly derivable from the assignments and equality predicates encountered along the test paths. The value of a test path is defined in terms of its ability to eliminate one or more of the characteristic errors which had escaped previous tests.

The number of test paths which may be selected under these criteria is limited by the finite number of independent errors. For linearly domained programs any predicate may be sufficiently tested using at most m+n+1 paths where m is the number of program input values and n is the number of program variables. This limit is independent of the complexity of the program control flow.

These results do not constitute a method for selecting paths for testing. The question of which paths are to be examined under this criterion has not

been addressed here. However it seems unlikely that the more general question of how to select paths for testing can be answered without some means of judging the path's value to the testing process. Such a means is provided here, together with the assurance that only a finite number of paths need to be selected.

The chief assumption throughout this analysis has been that predicates and computations affecting the control flow are restricted to a well-behaved class of functions, in this case linear, which permits the definition of a finite-dimensioned space of possible errors. It does not seem unreasonable to expect that similar results might be obtained for higher order functions. It is not clear whether the incidence of such functions is in practice sufficiently common to necessitate such an extension or to justify the additional number of test paths which might be required.

The model employed here does not take program structure into account. It remains to be seen what effects, if any, the use of well-structured control constructs might have on the selection of sufficient sets of test paths.

Work is continuing on this model, focusing on the extension of the analysis for linearly domained programs to domain errors caused by incorrect computations and on the applicability of these results to path selection strategies.

REFERENCES

1. L.A. Clarke, "Automatic Test Data Selection Techniques", in Infotech State of the Art Report on Software Testing, Vol. 2, 1979.

2. E.I. Cohen, A Finite Domain-Testing Strategy for Computer Program Testing, Ph.D. dissertation, 1978, Ohio State University.

3. J.B. Goodenough and S.L. Gerhart, "Toward a Theory of Test Data Selection", IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, pp. 156-173, June 1975.

4. W.E. Howden, "Methodology for the Generation of Program Test Data", IEEE Transactions on Computers, Vol. C-24, No. 5, pp. 554-559, May 1975.

5. W. E. Howden, "Reliability of the Path Analysis Testing Strategy", IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, pp. 208-215, September 1976.

6. C.V. Ramamoorthy, S.F. Ho, and W.T. Chen, "On the Automated Generation of Program Test Data", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, pp. 293-300, December 1976.

7. L.J. White and E.I. Cohen, "A Domain Strategy for Computer Program Testing", IEEE Transactions on Software Engineering, Vol. SE-6, No. 3, pp. 247-257, May 1980.

2.1.3. Module Integration Testing

Haley and Zweben have been investigating the issues involved when a "correct" module which has been thoroughly validated is integrated into a larger program context. It is desired to maximally utilize the information that this module is correct in designing the integration testing strategy. The paper that follows presents the results that have been obtained by our group, and has been submitted for publication to Journal of Systems and Software.

Development and Application of a White Box Approach to Integration Testing*

Allen Haley, Stuart Zweben
Dept. of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210

Abstract

Program testing techniques can be classified in many ways. One classification is that of "black box" vs "white box" testing. In black box testing, test data are selected according to the purpose of the program independent of the manner in which the program is actually coded. White box testing, on the other hand, makes use of the properties of the source code to guide the testing process. A white box testing strategy, which involves integrating a previously validated module into a software system is described. It is shown that, when doing the integration testing, it is not enough to treat the module as a "black box", for otherwise certain integration errors may go undetected. For example, an error in the calling program may cause an error in the module's input which only results in an error in the module's output along certain paths through the module. These errors can be classified as Integration Domain Errors, and Integration Computation Errors. The results indicate that such errors can be detected by the module by retesting a set of paths whose cardinality depends only on the dimensionality of the module's input for integration domain errors, and on the dimensionality of the module's inputs and outputs for integration computation errors. In both cases the number of paths that need be retested do not depend on the module's path complexity. An example of the strategy as applied to the path testing of a COBOL program is presented.

*This work supported in part by the Air Force Office of Scientific Research Grant AF F49620-79-C-0152 and by the National Science Foundation under grant MCS-8018769.

Development and Application of a White Box Approach to Integration Testing*

Allen Haley, Stuart Zweben
Dept. of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210

Introduction

Program testing techniques can be classified in many ways. One classification is that of "black box" vs "white box" testing. In black box testing, test data are selected according to the purpose of the program (as expressed, say, by a specification), independent of the manner in which the program is actually coded. Such approaches are described in [3] and [1]. Unfortunately, the insights needed to develop these ideas into an easily applied testing technique appear beyond the state of the art. White box testing, on the other hand, makes use of the properties of the source code to guide the testing process. Statement and decision coverage, and Domain Testing [7] are examples of white box strategies. While such techniques can be (and have been) automated, they suffer from either the inability to provide formal statements about the adequacy of testing (e.g. coverage approaches [3, 6]) or from impracticality due to the large amount of testing required (e.g. Domain Testing). What is needed are strategies which have some identifiable degree of reliability and yet do not require an inordinate amount of test data. For example, we intuitively believe that it is not necessary to require examination of every path in a program. But how can we

*Research supported in part by: AFSOR contract F49620-79C-0152
NSF grant MCS-8018769

determine which paths are "unnecessary"?

One possible approach to achieving this reduction is motivated by considering the problem of program development. In developing the solution to a large, complex problem, it is customary to form subdivisions which abstract interesting aspects of the total solution. These subdivisions might then be refined, implemented, and tested as independent units of the total system and then integrated to form a complete working solution to the original problem. When viewing the integrated program as the object to be tested, it may well be the case that the complexities are too great to make certain testing strategies practical. For example, consider a program P consisting of subprogram P1 containing m paths followed by subprogram P2 containing n paths. The integrated program can have a total of m*n paths, since any of the m paths in P1 can be followed by any of the n paths in P2. In the course of developing P however, it may well be the case that both P1 and P2 have been tested separately. It would be desirable if the correctness information obtained in unit testing P1 and P2 could be used in validating P. If the individual modules do not contain a large number of paths, it may in fact be possible to test all possible paths in each module. If the additional testing required at integration time was negligible compared to the unit testing overhead (for example, we might be able to ignore the internal control structure of a tested module when integrating it), the result would be a reduction of the magnitude of the testing problem from O(m*n) to O(m+n). While this represents in some sense an ideal

situation, it is clear that with such a potential for complexity reduction, even a less than ideal solution might represent a considerable improvement and yet provide a substantial degree of practicality.

Thus, the justification for the development of a method of integrating independently tested modules into the testing of a program is (1) to reduce the total testing complexity, and (2) to make the testing procedure conform to the way progra s are developed.

Integration Time Errors

In the remainder of the paper, we will explore the issues involved when a "correct" module (one which produces the appropriate output for any valid input) is integrated into a larger program context, with the goal of identifying testing strategies which are sensitive to integration time errors.

In order to be able to characterize the effectiveness of any testing approach, it is necessary to classify those kinds of errors that we might hope to detect. One proposal, due to Howden [3] distinguishes between domain errors and computation errors. A domain error occurs when a specific input follows the wrong path due to an error in the control flow of the program. A computation error exists when a specific input follows the correct path, but an error in some assignment statement causes the wrong function

to be computed for one or more of the output variables.

The notion of domain and computation errors turns out to be quite useful in characterizing certain types of integration problems. For example, consider a module M which has been thoroughly validated, say by some "Hypothetical Testing Strategy". so that it is free of both domain and computation errors. Module M is to be integrated into a program P. Assume that P has some computation whose result (call it C) is used in some predicate of M but is not used anywhere else in the program (see Figure 1).

```
                                                 .
              READ Ip                            .
              C = Ip                           IF C < 4
P             CALL M (.., C,..Om)        M     THEN Om = 1
              Op = Om                          ELSE Om = 2
              PRINT Op
```

Figure 1.
Program Containing a Computation Used Only in a Predicate of a Previously Tested Module.

Now suppose that the correct computation in P should have set C to Ip+1. In validating M, we may have ensured that M produces the correct output no matter which branch of the IF statement is taken, but P will still produce the wrong output if the initial value of Ip is such that $3 < Ip \leq 4$. However, if we do not happen to choose a value of Ip in this range we will not catch the error in the computation statement. Notice that, from the point of view of the program P, there is only one path to consider (READ Ip; C=Ip; CALL M (...); Op=Om; PRINT Op) if we ignore the internal structure of M at integration time and deal

only with P's structure. Yet this example shows that we must do more than just select a couple of values of Ip and examine the resulting values of Op. In this case, if we were to analyze the integrated program including the module's control structure, we would notice that the program contains a domain error, since values of Ip in the range $3 < Ip \leq 4$ follow the wrong path.

Computation errors cause another problem in ignoring the validated module's control structure at integration time. Assume that the program contains an incorrect computation whose result is passed to the validated module. Further assume that the only use of this result is by some computation in the validated module. As an example, suppose P is the same as in Figure 1, but M is changed as in Figure 2.

```
                  .
                  .
M       IF (condition)
        THEN Om = C
        ELSE Om = 2
```

Figure 2.
Module Which Transmits a Program Computation Error.

Assume once again that the computation in P should set C equal to Ip+1 instead of Ip. If integration test data were chosen which never exercised the true branch of the condition in M, then the resulting value of Om would always be 2 and the error in the computation of P would go undetected by simply examining the output of the program.

These two examples have elements in common. In both cases there is an error in the code preceding the call to the validated module. The error causes one of the module's inputs to have an incorrect (not invalid) value; it is possible for the error in the module's input to not be reflected as an error in the module's (and hence the program's) output, since transmission of the error to an output may be dependent upon the particular path chosen through the module. It is therefore clear that, when integrating a previously validated module, one needs to know more than just that the module is correct. If information relevant to the module's internal structure is ignored, it is possible for both domain and computation errors in the integrated program to go undetected. Therefore it is natural to ask at this stage "What, in addition to knowing that the module is correct, will allow effective integration testing to be done?". Furthermore, in view of the introductory remarks concerning black box testing, we are interested in knowing if this additional information can be obtained "automatically", by examining properties of the program structure.

Detecting Integration Errors

Two approaches to answering the question posed at the end of the previous section are suggested by the examples presented in that section. Since our goal is to detect errors in the module's input, we could simply require that all input values to the module be output (along with the normal output of the calling

program). This technique is not new, as programmers often print out values of intermediate/temporary variables. However it is often hard to know whether an intermediate program value is correct. More likely, the programmer is only interested in examining the final outputs of the (calling) program.

Therefore we consider a second approach. It would appear that the chief problems presented in the previous section are that 1) we may have failed to retest adequately a predicate in the module whose interpretation is affected by a particular error in the calling program (for integration time domain errors), or 2) an error in the calling program that produces an error in the module's input might not be passed to the module's outputs (and hence to the program's outputs) along those paths that are executed in the module during integration testing. The solution, therefore, seems to be a matter of "retesting" during integration testing, a set of paths through the module which are sensitive to these problems. We will refer to this set as the Integration Test Set for the module. The integration test set should meet two important criteria. First, it should be capable of detecting all of the integration testing errors identified in the previous section. Second, it should contain as few of the module's paths as is necessary to meet the first criterion.

In order to find an integration test set for a particular module it is first necessary to be able to determine all the possible "different" integration domain and computation errors that can occur in the module. Once this is done, it is necessary

to be able to tell, given a path through the module, which of the possible integration errors the particular path will detect. The details of how to perform these operations can be found in [2]. A key idea used in the derivations is that, if a module has m input variables then there are only m independent ways in which an input error can occur. That is, any input error can be expressed as a combination of the m independent error types (referred to as "error directions").

Example 1

Suppose m=2 so that the module has two inputs I1 and I2. Now I1 can be in error on a particular call to the module, so that the module is in fact called with I1′=I1+e1. Similarly, the module can be called with I2′=I2+e2. But any incorrect inputs can be expressed in terms of the correct input vector $\binom{I1}{I2}$ and an error vector $\binom{e1}{e2}=c_1\binom{1}{0}+c_2\binom{0}{1}$ where c_1 and c_2 are constants. In this sense, there are 2 linearly independent error directions.

Using the concept of error directions, and letting

m = the number of input variablies for the module,

and

n = the number of output variables for the module,

the number of potentially detectable integration domain and integration computation errors can be determined.

For integration domain errors, we are attempting to detect situations where an erroneous input to a previously tested module causes

1. some predicate in the module to have an erroneous interpretation, so that
2. an incorrect path is taken by this input, resulting in
3. a different computation to be performed from that which takes place on the correct path, so that
4. one or more of the program's outputs has an incorrect value.

The notion of an erroneous predicate interpretation can be illustrated as follows. Suppose the previously tested module has inputs I1 and I2 and contains a predicate of the form "IF P1≥0" whose interpretation along some path in the module is "IF I1+I2≥0". If both I1 and I2 are in error in such a way that their errors cancel (ie I1'=I1+e and I2'=I-e for some e≠0) then this predicate is "blind" to this error and therefore is incapable of detecting it. However, some other predicate in the module might have an interpretation like "IF I1≥0". This second predicate <u>is</u> capable of detecting the "canceling errors" (though it, too, is blind to certain errors, such as those which only involve I2). We must also be wary of a situation in which an error to the module's input causes a predicate to have an interpretation which is a multiple of the correct interpretation, for then both interpretations evaluate identically for any input, and the error will go undetected.

<u>Example 2</u>

Consider the predicate "IF P1≥0" whose interpretation, in terms of the module inputs I1 and I2, is "IF I1+I2≥0". Suppose the program which calls this module has inputs X, Y, and Z.

Further assume that the correct assignments to I1 and I2 by the calling program should be {I1=X+2*Y, I2=2*Z} while the incorrect assignments are {I1′=2*X+Z, I2′=3*Z+4*Y}. Then

$$I1+I2\geq 0 = X+2*Y+2*Z\geq 0$$

while

$$I1'+I2'\geq 0 = 2*X+4*Y+4*Z\geq 0$$

$$= 2*(X+2*Y+2*Z)\geq 0$$

Both interpretations evaluate identically for any triplet (X,Y,Z).

This discussion illustrates that there are at most m+1 "different" integration domain errors that can occur, and hints at the kind of analysis, based on algebraic properties of the predicate interpretations, that would be needed to identify a candidate *set of paths for the integration test set*. Analysis of integration computation errors, again using algebraic properties of the code, reveals that at most (m*n)+1 "different" errors are possible [2].

Given that there is a path through a module containing at least one predicate interpretation that is linear in terms of the module′s inputs, then that path will be able to detect at least one of the m+1 possible integration domain errors. If enough paths with linear predicate interpretations exist in the module, and we assume a path won′t be included in the integration test set unless it contributes to the detection of at least one new error, then for integration domain errors we need at most m+1

paths in the integration test set. Notice that for the above result there is no requirement that <u>all</u> the predicate interpretations in the module be linear.

A similar situation occurs for determining the maximum size of the integration test set for integration computation errors. In this case, if there exist enough paths through the module such that the computations along those paths are linear in terms of the module's inputs, then the integration test set for integration computation errors will contain at most $(m*n)+1$ paths.

The determination of whether any particular path should be included in the integration test set involves applying standard linear algebra techniques to the results of a symbolic evaluation of the path. The complexity of the computations involved in applying these techniques to any one path is no worse than $(m+1)^3$ for integration domain errors, and $(m*n+1)^3$ for integration computation errors.

Combined Integration Testing

If the desire is to do integration testing for both domain and computation errors at the same time the integration test set that is required is simply the union of the integration test sets needed for each case. Therefore, the upper bound on the number of paths in the combined integration test set is $(m+1)+(m*n+1)$. In general the integration test set will contain far fewer members,

for a number of reasons.

First, a path might contribute to the detection of more than one integration domain error because the path might contain more than one predicate. Second, some of the paths in the two integration test sets might be the same, resulting in the union of the integration test set being smaller than the upper bound. The third, and most significant reason, is related to the existence of sufficient linear paths in the module to detect all integration errors. Our experience has shown that for most modules it isn't possible to detect all the integration errors, even when all paths in the module are examined. The cause of this isn't a lack of linear paths, but is instead that many different paths have the ability to detect the same integration error. This has two effects on the integration testing strategy. First, since the number of possible errors that can be detected is a subset of all the possible integration errors, the maximum number of paths in the integration test set is reduced by the number of errors that can't be detected. In practice, this reduction can be substantial. Second, nonlinear paths might be capable of detecting some of the integration errors that aren't detected along linear paths.

Nonlinear Paths

A possible method of handling paths that contain nonlinear predicate interpretations and computations is to include any such

paths in the integration test set. While this simplifies the selection of the integration test set (only the linear paths need be analyzed to determine if they should be included in the test set), it is only a reasonable solution if the number of nonlinear paths in the module is relatively small.

In modules where the number of nonlinear paths is too large to employ the above solution it would be helpful to analyze the nonlinear paths to determine if they can detect integration errors not detected by the linear paths. This is possible if the type of nonlinearity is such that the predicates and computations can be represented in a canonical form as elements of a finite dimensional vector space. The class of multinomials, for example, satisfies the above condition, and could therefore be handled by the integration testing strategy, albeit at the expense of additional computational complexity.

Fundamental Limitations of White Box Integration Testing.

There are a few fundamental problems with this testing strategy which need to be addressed. The first problem only occurs with the detection of integration domain errors. For integration domain errors we have chosen paths through the module to guarantee that, if an input error exists, then some predicate in the integration test set will shift. A problem arises if the predicate that is shifting is redundant. This occurs when some other predicate along that same path through the integrated program supersedes the shifting predicate so that the shifting

predicate isn't part of the border of the path that is being tested. In this case an integration domain error might go undetected that would have been detected along some other path in the module. Since there is no requirement that the superseding predicate be in the module (it can be in the calling program either before or after the module), there is no way to avoid this problem by simply examining the structure of the module. In real programs and modules that we have examined, this problem in fact occurs. However its significance has not yet been thoroughly analyzed.

The second problem with the integration test set affects both integration domain and computation errors. This problem arises because the paths we have chosen for the integration test set might not be feasible with respect to the calling program. Again this problem might prevent us from detecting certain integration errors that would have been detectable along some other feasible path through the module. This problem, again, can't be solved through examination of the module because the infeasibility could be the result of predicates in the program outside of the module.

<u>An Example of Module Integration Testing</u>

The integration testing strategy has been applied to the testing of a production COBOL program for computing hourly payroll. The program is divided into six main modules, with three

of these modules (subsystem 1, 2, and 3 respectively) containing lower level submodules. The relevant structural properties of these subsystems are as follows:

1. Subsystem 1
   a) 2 individual modules
   b) 78,429 total paths
2. Subsystem 2
   a) 6 individual modules
   b) 2,904,545,988 total paths
3. Subsystem 3
   a) 2 individual modules
   b) $4.679*10^{24}$ total paths

Clearly the amount of work required to do a complete path test for these subsystems is unreasonably large. The hope is that by applying the integration testing strategy, the total number of paths that need to be examined for testing will be substantially reduced.

Because of the limited number of nonlinear predicates and computations in this program, we decided to include all nonlinear paths in the integration test set at each level. Furthermore, it should be noted that the above path counts for each module are theoretical paths, and may contain paths that are infeasible.

The integration testing strategy was applied to the program in the following manner. First, all paths in the lowest level modules were examined, and the integration test set for those modules was determined. Next, all paths in the next higher level of modules were examined in combination with the integration test set from the lower level modules. This process was continued

until the highest level modules were reached.

When choosing paths at each level the order in which the paths are chosen can affect the size of the integration test set, because each path can contribute to the detection of more than one error direction for integration domain errors. For the purposes of this example we always chose the paths that contained the most predicates first. In cases where the number of predicates along two paths were the same, the true branches were always chosen first. While we don't claim that this selection process gives a minimum integration test set, our experience has shown that it does tend to reduce the size of the integration test set.

The following table gives the total number of paths that were examined using the above method of applying the integration testing strategy.

1. Subsystem 1	15,621	integration paths
2. Subsystem 2	12,057	integration paths
3. Subsystem 3	$1.4*10^6$	integration paths

In addition to the reduction in the total number of paths that need be tested, the integration testing strategy offers the added benefit of using paths which are, on the average, shorter than those which must be tested when the program is considered as a single unit. This is because in integration testing many of the paths lie entirely within the lower level submodules, rather than

spanning the entire program (recall that in the integration testing strategy the modules are tested independently, and this testing is reflected in the above numbers). This should serve to reduce the complexity of the testing process since many testing strategies are dependent, at least in part, on the number of predicates and computations on a particular path (strategies which employ symbolic analysis fall into this category).

Final Remarks

We have shown that it is possible to detect integration domain and computation errors using a set of paths whose cardinality depends on the complexity of the module interfaces of the subsystem under test, rather than on the path complexity of that subsystem. Furthermore, these errors can be detected by examining the normal outputs of the subsystem, without requiring intermediate values or extraneous quantities to be examined. The reduction in the number of paths that need to be examined can be several orders of magnitude, and in certain situations might mean the difference between a practical and impractical testing plan. In other cases the number of paths required may still be too large to be practical, but in such situations, the integration testing strategy can provide information concerning the kinds of errors that remain untested after some subset of the integration test set has been chosen. The integration testing strategy is based solely upon properties of the program structure, thereby illustrating that white box testing need not be bound by the path complexity of the system under test. However, it is faced with

problems such as feasibility and redundancy which are fundamental to a technique which uses no information about the purpose of the code under test. We believe that a well thought out testing strategy has got to make use of specification information to be both practical and effective. In the absence of the required sophistication to employ this information, it is at least helpful to know the extent to which the code itself can guide the testing process.

References

[1] Gourlay, J. S., Theory of Testing Computer Programs, PhD Thesis, The University of Michigan, Computer and Communication Sciences, 1981.

[2] Haley, A. and Zweben, S., "An Approach to Reliable Integration Testing", Technical Report TR-81-5, Computer and Information Science Research Center, The Ohio State University, May 1981.

[3] Howden, W. "Reliability of the Path Analysis Testing Strategy", IEEE Trans. on Software Eng., SE-2, 3, September, 1976, p. 208-214.

[4] Howden, W., "Effectiveness of Software Validation Methods", Infotech State of the Art Report on Software Testing, Vol 2, Infotech International, 1979, p. 131-146.

[5] Howden, W., "Functional Program Testing", IEEE Trans. on Software Eng., SE-6, 2, March 1980, p. 162-169.

[6] Westley, A., ed., Infotech State of the Art Report on Software Testing, Vol 1, Infotech International, 1979.

[7] White, L. and Cohen, E., "A Domain Strategy for Computer Program Testing", IEEE Trans. on Software Eng., SE-6, 3, May 1980, p. 247-257.

[8] Zeil, S. and White, L., "Sufficient Test Sets for Path Analysis Testing Strategies", Proc. 5th Int'l. Conf. on Software Engineering, San Diego, CA. March 1981, p. 184-191.

2.1.4. Other Issues in Testing

Prof. Chandrasekaran edited a special issue of the IEEE Trans. Software Engineering on computer program testing. The editorial that follows is a technical discussion of several issues in program testing in the context of the papers in the Special Issue.

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. SE-6, NO. 3, MAY 1980

Guest Editorial

Special Collection on Program Testing

I. Introduction

THIS special collection had its genesis at the IEEE Computer Society Workshop on Software Testing and Test Documentation, held in Fort Lauderdale, Florida, during December 18-20, 1978. This collection is devoted to selected papers on program testing that were presented at the Workshop. It contains six papers whose concerns span a wide range: theoretical issues, practical experience with particular strategies, extension of a class of techniques to new classes of language constructs, and building integrated test tools. In the next section, we present an overview of the papers in the collection.

II. Overview of the Special Collection

The paper by Weyuker and Ostrand begins by examining the conceptual, definitional framework for test data selection presented by Goodenough and Gerhart [1]. After clarifying some of the definitions in [1], it is pointed out that this framework needs to be enriched in order to make the theory yield methodologies for *practical* test data selection. In particular, Weyuker and Ostrand attempt to lower the goal from one of proving program correctness by testing to designing testing methodologies which expose the presence of certain *specified types of errors.* This is a direction which, as we shall see, is also one taken by several other papers in this collection.

Weyuker and Ostrand proceed by introducing the notion of *revealing subdomains* of the input space. Intuitively, one looks for a partition of the domain of the program into subdomains such that correct or incorrect performance of the program for any element of the subdomain implies correct or incorrect performance of the program for all elements of the subdomain. The motivation behind seeking such subdomains is that often the specifications lead to a partition of the input space such that elements of each subdomain are in fact processed rather uniformly by the program. If one *could* find such a partition with a finite number of subdomains, one would be in very good shape, since a finite number of test data would suffice.

However, finding such partitions which are revealing for all errors is in general a tall order. But the framework enables the authors to talk about subdomains which are revealing for particular types of errors. That is, given a program and a candidate error, one will often be able to identify a subdomain as one that should be surely affected by that error if it were present. Such a subdomain is then revealing for that error.

This work was supported by the Air Force Office of Scientific Research under Grant 77-3416 and under Contract F-49620-79-c-0152.

Identification of such subdomains is still a creative act, based on studying the specifications and even the program structure. The authors demonstrate the approach on some example programs. They acknowledge that their paper outlines an *approach* rather than a concrete methodology. More work would need to be done on a wide class of problems and programs before the practical usefulness of the notion of revealing subdomains can be established. However, it is clear that the theoretical framework for testing is enlarged by the notions introduced in this paper.

White and Cohen describe a strategy which concentrates on the detection of a particular type of errors, viz., *domain errors* as defined by Howden [2]. These errors result from errors in the control flow of a program, which cause some inputs to follow an incorrect path, i.e., those inputs are in the wrong domain (or subdomain in the terminology of the previous paper). They note that essentially these errors result in a shifting of the boundaries of the domains. Thus the technical question becomes one of how and under what conditions can such boundary shifts be detected by test data. White and Cohen note that domain errors can be traced to errors in predicates in the program. Further, predicates which are linear in input variables produce domain boundaries which are hyperplanes. They show that in such cases, as long as both the incorrect and the correct (but unknown) predicates are both linear, then a finite number of test points suffice to test for such boundary shifts. They precisely specify the conditions under which this strategy can be guaranteed to detect all domain errors of given magnitudes. The strategy can be extended to the case of nonlinear predicates which are low degree polynomials in input variables. In any event, the authors argue persuasively for the practical importance of the case of linear predicates in its own right. They also provide an analysis of the complexity of their domain testing strategy, and give a useful discussion of the inherent limitations of their approach. It is worth pointing out that the testing for domain errors in this manner is still a path testing strategy and is subject to the difficulties caused by the rapid increase in the number of paths as the size of the program grows. Further research on this aspect of their strategy is needed to make this approach practical.

An interesting relationship between the papers of Weyuker and Ostrand on the one hand and White and Cohen on the other can be observed. Both are, as noted, interested in guaranteeing detection of specified (types of) errors. That leads the authors of both papers to look for subsets of the input space which are especially sensitive to the error; revealing subdomains in the case of the former, the region around the do-

0098-5589/80/0500-0233$00.75 © 1980 IEEE

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. SE-6, NO. 3, MAY 1980

main boundary in the case of the latter. It would be interesting to investigate the extent of the relationship, and whether the White-Cohen results can be cast in the framework of revealing subdomains.

Foster's paper on error sensitive test cases is an attempt to apply classical logic hardware testing techniques to the detection of "code errors" in software which survive compilation and assembly: errors such as reference to wrong variables, incorrect relational or arithmetic operators, etc. In hardware testing a collection of test patterns were determined that together would detect faults in any logic gate and that were minimal in some well-defined sense. Foster, by means of a combination of experimentation and theory, develops a set of rules for the generation of such test patterns, i.e., a set of test data sensitive to a sizable class of code errors. These rules are heuristic and, unlike in the hardware case, do not guarantee detection of all code errors of the given class. This is not surprising in view of the increased complexity of the operators in the software case. However, the rules are clearly very useful in that they represent a systematized way of generating test data sensitive to a class of errors. In any case, a complete set of rules which guarantee the detection of all code errors might be computationally too complex to be useful.

While current data flow analysis techniques can handle most single-process programs, there is a need for new analytic techniques for dealing with concurrent-process programs because of the complex data and control flow possibilities introduced by their synchronization constructs. The paper by Taylor and Osterweil is a response to this need, and the ideas contained in it arose in the context of production and testing concurrent-process flight software.

Static analysis is often effective in weeding out errors that are costlier to detect by dynamic testing techniques. Extension of data flow analysis to concurrent-process software requires more complex control flow models. The PAF—process augmented flowgraph—is a concept designed to capture the data and control flows in concurrent-process programs with *schedule* and *wait* statements as synchronization constructs. The PAF and associated algorithms are capable of detecting errors due to shared data items being referenced by one process before any other process defines them. In addition, certain anomalies in the PAF indicate the occurrence of poorly coordinated processes. A number of examples are given which illustrate these notions. It would seem fruitful for further work in this area to extend these notions to a broader class of constructs, such as *open*, *close*, and *signal* statements.

While a theoretical/conceptual infrastructure for program testing is slowly emerging, only experience in testing real programs of nontrivial size can determine the directions in which the theoretical framework needs to be extended for greater practical relevance. This is because the theoretical frameworks, with few exceptions, are somewhat detached from properties of programs (or classes of them) *as they are in fact written*. For instance, Woodward, Hedley, and Hennell report on the fact that, in the large class of numerical software that they studied, a surprisingly large fraction of paths were *infeasible*, i.e., no input data will ever execute those paths. Whether this is a much more general phenomenon, or whether some special characteristics of numerical computations (at least the way people organize them) result in this is an issue that would be useful to study in a theoretical fashion, now that this phenomenon has been spotlighted by the authors' experience in testing this class of programs.

While path testing strategies are theoretically very effective, the number of paths is often very large, if not infinite. Effective criteria for selecting a small number of paths to test, as well as reliable measures of the degree of testedness achieved at any given stage in testing, are badly needed. Woodward *et al.* propose a hierarchy of structural test metrics, which include and extend simpler measures such as "fraction of statements executed," and "fraction of branches executed." They are hierarchical in the particular (weak) sense that a value of unity for the metric of order n implies a value of unity for the metric of order $(n - 1)$, (just as "all branches executed at least once" implies, "all statements executed at least once"). It is not known whether these metrics are hierarchical in a stronger sense, i.e., a high value for order n metric implies a high value for order $(n - 1)$. Nevertheless, the notions have a reasonable heuristic content, and the authors discuss their experience in experimenting with these metrics for the class of numerical programs. It is in this context that the problem of infeasible paths discussed earlier becomes significant, in that their existence prevents the metrics from reaching high values.

The authors investigate the effectiveness of a technique called "allegations" (originally due to Osterweil [3]) to prevent the generation of at least some of the infeasible paths. These allegations can be viewed as any heuristic knowledge that the tester may have about the logical conditions that need to be satisfied by a path for it to be feasible. The idea is that automatic analyses—at least those currently available—are not "intelligent" enough to discover these by themselves, while a human tester's understanding of the program will be a good source of such heuristic knowledge. Once again the question of whether this heuristic technique is powerful enough to be useful in practice is an empirical one, which is answered by the authors in the affirmative for the class of numerical programs that was their concern. They wisely recommend, however, that the long-term solution lies in the design of languages with constructs which do not permit the generation of large numbers of infeasible paths in the first place.

Finally, we come to the paper by Voges *et al.* on an integrated testing tool that they have implemented. Called SADAT (Static and Dynamic Analysis and Testing), this tool is designed for testing Fortran programs that have been compiled error-free. The main modules of SADAT are static analyzer, dynamic analyzer, test case generator, and path predicate calculator. Static analysis is useful to detect certain forms of dead code, undeclared or unused labels and variables, and jumps into a loop. In addition, the output of the static analysis phase serves as a database for later analysis.

SADAT's dynamic analysis documents the execution of program test runs. Basically this consists of instrumentation for the execution count of various branch points. This dynamic analysis is useful for identification of dead code, determining correctness of loop iterations, and optimization.

The test generation subsystem automatically generates a subset of paths with almost complete C_1-coverage (i.e., each arc and each node is represented in at least one path). In addition to the automatically generated paths, the user can specify a path as a sequence of statements. The final module calculates path predicates by symbolic evaluation.

The authors report briefly on their experience in using this tool. While the static and dynamic analysis components were found to be useful and stable, the test data generation system suffers from problems associated with symbolic execution, in particular, in handling loops and subroutine calls.

III. Concluding Remarks

The papers in this special collection seem to me mainly to consolidate, refine, extend, and experiment with many existing notions and theories of testing. The thrust is towards approaches that can support the *heuristic components* of testing. There is increasing recognition that it is unlikely there will be a grand theory of testing which will lead to fully automated testing systems. Instead the tester will be called upon to use his intuition and problem-dependent knowledge in a disciplined manner to test for a variety of specified error types. But is is crucial that this less ambitious, heuristic activity be nevertheless firmly embedded in an underlying framework which is logical, rigorous, and well-understood. Supplying this framework, which will necessarily include properties of programs as they are in fact designed and written, will be the task of researchers in program testing.

Acknowledgment

My greatest debt of gratitude must go to the referees who provided professional, scholarly, critical yet sensitive reviews of the papers, sometimes through second and third revisions. If a scientific field can be measured by the quality of its reviews, I think that testing research is doing very well. I would like to acknowledge my indebtedness to F. E. Allen, L. A. Clarke, R. E. Fairley, S. L. Gerhart, J. B. Goodenough, R. Hamlet, R. C. Houghton, W. E. Howden, J. C. Huang, L. J. Osterweil, C. V. Ramamoorthy, J. Reif, V. Voges, L. Yelowitz, and S. H. Zweben. I thank the authors of the submitted papers for their infinite patience, cooperation and good humor, and willingness to play by the rules of the peer review game. E. Miller and D. Fife as the organizers of the original Workshop also deserve appreciation from our community.

References

[1] J. B. Goodenough and S. L. Gerhart, "Toward a theory of testing: Data selection criteria," *Current Trends in Programming Methodology*, vol. 2, R. T. Yeh, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1977, pp. 44–79.

[2] W. E. Howden, "Reliability of the path analysis testing strategy," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 208–215, Sept. 1976.

[3] L. J. Osterweil, "Allegations as aids to static program testing," Dep. Comput. Sci. Rep., Univ. Colorado, Boulder, CO.

B. Chandrasekaran
Guest Editor

B. Chandrasekaran was born in India on June 20, 1942. He received the B.Eng. degree with honors from Madras University, Madras, India, in 1963 on a National Merit Scholarship, and the Ph.D. degree from the Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia, in 1967 on a Moore Fellowship.

From 1967 to 1969 he was Research Scientist with Philco-Ford Corporation, Blue Bell, PA, working on problems of pattern recognition, and since 1969 he has been with The Ohio State University, Columbus, where he is currently Professor of Computer and Information Science. His current major research interests are software testing, aritificial intelligence, pattern recognition, and computer graphics.

Dr. Chandrasekaran is currently Associate Editor of the IEEE Transactions on Systems, Man, and Cybernetics, and Chairman of the IEEE Systems, Man, and Cybernetics Society's Technical Committee on Artificial Intelligence. He was the recipient of a 1976 citation from the Pattern Recognition Society for an "outstanding contribution." For the last two years he has been co-directing a research project at Ohio State on Software testing, and he was a member of the Program Committee of the 1978 IEEE Computer Society Workshop on Software Testing and Test Documentation. He is a member of the Association for Computing Machinery.

2.2. Knowledge-Based Program Synthesis and Problem Solving

Our research in this area is presented here in three sections. The first report is one by Gomez on a system called LLULL, which accepts programming problems (in the domain of checking accounts) stated in natural language and produces Pascal programs for them. The second paper in this group is a paper by Chandrasekaran on the general issues of distributed problem solving where knowledge sources cooperate to perform a complex problem solving activity. This was an editorial to a special issue of the IEEE Trans. System, Man and Cybernetics on distributed problem solving.

2.2.1. Understanding Programming Problems Stated in Natural Language

The following is a technical report by Fernando Gomez, issued as technical report OSU-CISR-TR-81-9.

(OSU-CISR-TR-81-9)

UNDERSTANDING PROGRAMMING PROBLEMS STATED IN NATURAL LANGUAGE

by

Fernando Gomez

Work performed under

Contract F49620-79-0152

Air Force Office
of
Scientific Research

Computer and Information Science Research Center

The Ohio State University

Columbus, OH 43210

February 1981

Table of Contents

Preface

1. Introduction 1

2. Background 3
   - 2.1 Concepts as Specialists 3
   - 2.2 Passive Frames 4
   - 2.3 Concepts as as Abstract Representation 5

3. Parser 5
   - 3.1 Noun Group 5
   - 3.2 Understanding the Concept Underlying the Noun Group 7
   - 3.3 Clauses Completing the Description 9
   - 3.4 Verbal Concepts 12
   - 3.5 Prepositions 13

4. Parsing into Knowledge Structures 15

5. A Checking-Account Programmer .26
   - 5.1 The Planner 27

6. Comparison with Related Approaches and Future Research 29

7. A Computer Run 30

References 31

FEB 26 1981

Understanding Programming Problems Stated in Natural Language

Fernando Gomez
Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210

ABSTRACT: A system to understand programming problems stated in natural language is described. Parsing is viewed as a process in which high level sources of knowledge override low level linguistic processes. Thus, the need of a low level parser with the necessary knowledge to determine the meaning of propositions, of verbs with many senses, of the noun group, etc. is recognized, and accordingly one has been built. But the function of this parser is not to produce an output to be interpreted by an interpreter or semantic routines, but to start the parsing and to proceed until a concept relevant to the theme of the text is recognized. Then the concept (in the form of a computer program) takes control of the parsing overriding the low level linguistic processes. The high level sources of knowledge parse the text directly into the relevant concepts that define a programming problem. The system has understood ten problems taken verbatim from introductory texts to programming as well as many variants on those problems. We have built a small system that takes the parser output for checking account problems and produces a PL/1 program. A very brief description (only two pages) of the problem solver is given.

KEYWORDS: Natural Language Understanding, Parsing Directly into Knowledge Structures, Descriptive Verbs, Understanding The Noun Group, Automatic Programming.

1. Introduction

Recently the emphasis on automatic programming research (AP) has shifted from the design of comprehensive systems intended to automate the totality of the programming activity (see [1], [2] and [3] for three excellent surveys of AP) to a mixture of AP and programming environments [4], [5], or formal specifications [6], [7], or natural language as a very high level programming language [8]. This paper addresses the programming activity in its totality. But it focusses on the communication aspect of AP. Only two pages are dedicated to the problem solving aspect.

From the natural language point of view, this paper belongs to a category that Novak [9] has called "natural language problem solvers". The earlier efforts by Bobrow [10] and Charniak [11] belong to this category. More recently are those by Heidorn [12], Hayes and Simon [13], Novak [9], and Ginsparg [14]. A characteristic that distinguishes our system from the last group is the way the understanding of the natural language text is done. Although there are important differences between these systems, the mapping of the text into a more or less syntactic structure is common to all of them. From there on, an interpreter or semantic routines transform the syntactic structure into the final internal representation. In our system, which we call LLULL, the text is directly parsed into the internal representation without intervening interpreters. But in contrast to the work of Schank and his

collaborator [15], we recognize the need of a parser that will be able to produce some kind of segmentation (a case meaning structure), the determination of the meaning of prepositions and verbs, the understanding of the noun group, the disambiguation of "and", etc.. When a sentence is going to be processed, the parser gets started and proceeds until it finds a concept that is recognized as being relevant to the theme of the text that is being parsed. From there on, the concept (in the form of a program) takes control <u>overriding</u> the parser. The concept continues supervising and guiding the parser until the sentence has been processed. Thus the basic idea is : in understanding natural language, the low level linguistic knowledge shared by the speakers of a language is <u>overriden</u> by high level sources of knowledge (concepts). Our program has understood ten programming problems on the topics of checking accounts, payroll, and exam scores. The problems have been taken verbatim from introductory texts to programming. A representative example is shown in fig. 1.

The problem solver receives as input a list containing the name of the concepts that the parser has built. It is a based knowledge system. For each conceptual step or level of abstraction [16] in which a programming problem decomposes, there is a specialist [17] that knows what kind of things must be present and what to do if they are missing. Each of these specialist has a list of things they have to do, called the task list. A <u>planner</u> will fill this list before the specialists are activated. We have implemented a system to

produce checking account programs. Presently the system has synthesized a PL/1 program for the example in fig.1 and other PL/1 programs for similar checking account programs. Sections 5 and 5.1 contain a brief description of the system.

2. Background

There are important differences between the language used to express human actions and the language of scientific books, text books, programming examples etc.. A striking difference is the predominance of descriptive verbs over action verbs in the latter. This is not surprising if we realize that the latter contexts deal essentially with the description, relations and illustrations of entities. In these contexts, *understanding* rests basically on the recognition of what is said of the nominal concepts that make up the sentences.

2.1 Concepts as Specialists.

Assume that a human programmer who is familiarized with checking-accounts programs reads the example of fig.1. As soon as he will begin to read the text, those concepts relevant (input, output etc.) to the example will be recognized by him. He will start to record knowledge around those concepts. We may say that the relevant concepts take control of the parsing (understanding) process overriding the low level linguistic knowledge.

We assume that this knowledge is stored under each concept under the form of production rules [18]. Concepts are taken as a cluster of production rules. This notion of concept as a system of rules governing the application of predicates to a concept is already found in Kant's Critique of Pure Reason . Kant also considered concepts as a kind of representation (vorstellung). According to this view, concepts are an abstract representation of the properties of an object. This later notion is akin to the notion of concept used in such knowledge representation languages as KLONE, KRL, FRL, and identical to the notion of concept of section 2.3.

2.2 Passive Frames

Understanding does not only depends on the concepts of section 2.1. The sentence, "The instructor records the grades on cards" is a perfect sentence, yet does not make any sense in the context of the example in fig.1. None of the concepts of the sentence belongs to those concepts associated with checking-accounts problems. The only way we can have made that decision is if we have knowledge ab[illegible] checking-accounts prestored in memory, and when we are exposed to the right context this knowledge is activated to direct our understanding. This corresponds to Minsky's notion of frame. For each kind of problem, LLULL has stored in LTM a passive frame. We have used the adjective "passive" to indicate that the content of the frame is not altered during processing. Each of them contains a list of the relevant concep[illegible] for that

problem, a list of the most usual input or output etc..

2.3 Concepts as an Abstract Representation

During parsing, the sentences are mapped into the relevant concepts that define the problem. Each concept is represented as a labelled structure describing its properties. The type of labels (slots) as well as what to put in them is decided by the specialists of sec. 2.1.

3. Parser

Although most people will be not able to have a deep understanding of a text about microbiology, yet they will be able to distinguish between the different senses of prepositions and verbs, to understand the noun group etc.. This type of linguistic knowledge is shared by all speakers of a language and it allows us to read even the most obstruse texts and still to get something out of them. We have built a parser that has that linguistic knowledge to a certain extent, of course.

3.1 Noun Group

The part of the parser that handles the noun group is called DESCRIPTION. It is based on the following assumptions. Any word that may form part of the description of an entity (nominal concept) is tagged in the dictionary with the marker DESCR (for description). Those words are articles, adjectives, pronouns and nouns . Articles, pronouns, and demonstrative,

distributive and quantitative adjectives are also tagged with the marker STD to indicate that they initiate a new description. Personal and impersonal pronouns (You, it, they) and are tagged with the marker ERU to indicate their exclusively referential use [19].

DESCRIPTION is invoked by any function that finds a word with the marker DESCR. If that word has also the marker ERU, DESCRIPTION exits. Otherwise it will be processing words until 1) one of them does not have the marker DESCR or 2) it has the marker STD or the marker ERU. 3) or it is a proper name and at least a noun has been found in that noun group. Some examples follow:

(1) (The blue apple) the gardener picked is made of metal.
(2) (The U.S forces) Germany out of France.
(3) (The U.S forces) Germany defeated were unprepared.
(4) (These books) belong to Peter.

We have used the brackets to indicate the segmentation produced by DESCRIPTION. In 1), the word following "apple" has the marker STD. In 2) and 3) the word following "forces" is a proper name and a noun has been already found (notice that when "U.S" is found no noun has been processed). In 4), the word following "books" has not the marker DESCR. DESCRIPTION groups the maximum number of words under the noun group (see [20] for a comparison). This is why it produces the wrong segmentation of 2). Notice that 3) has the same surrounding words as 2). In cases where there are not semantic clues to establish a boundary, humans as well as machines have to back up or look ahead, whatever you prefer. In our opinion, backing up is a

more natural phenomenon in this context, while looking ahead is more natural in determining the meaning of prepositions or disambiguating "and". There are cases in which semantic reasons decide where the noun group boundary is. For example, when a noun is followed by a noun that denotes citinzenship or a profession, it may be assumed that the preceding noun is not a modifying noun. For example, "The book Spaniards like is Don Quixote." In other contexts, only a deep understanding of the concept underlying the noun group will help in establishing a boundary. Consider : "The book people love is The Metamorphosis." Although one can imagine an interpretation that takes "book" as a modifying noun, it is unusual. Our knowledge of people does not include book as being one of its predicates. Thus this knowledge (in the form of a program stored in "people) can advise the parser to establish a boundary in "book". This idea is explained with more detail in the next section.

3.2 Understanding the concept underlying the noun group.

The segmentation produced by DESCRIPTION tells us only when a noun group ends. We need a function that will understand the meaning of the noun group while its components are being processed. An essential problem in uderstanding the noun group is posed by the complex nominals. The term "complex nominal" has been used by Levi [21] to refer to nominal compounds like "the program output", to nonpredicating adjectives like "electrical engineer" and nominalizations like

"film producer".

Essential to the problem of complex nominals is that in many instances different combinations of words denote the same concept. For example, "the old balance", "the balance at the beginning of the period", or "the output", "the program output". Whether we are dealing with a complex nominal or with a normal noun group like "the red apple", understanding depends on our ability to recognize the property being predicated of the noun as fitting our knowledge of the noun. A parser that parses "the blue apple" into (PHYSOBJ TYPE (APPLE) COLOR (BLUE)), and does not register the fact that "blue" is not a color of apples is missing something.

Our approach to the understanding of the noun group has been determined by the natural language context we are studying. The natural language understander has to solve the different ways of referring to the same concept, otherwise the problem solver will be in trouble. Thus our main concern has been to identify the concept underlying the noun group. We have stored the knowledge necessary to identify the underlying concept of the noun group in the head noun. What enables us to find something funny about the expression "the blue apple" is that our knowledge of apples does not include the blue color. This knowledge is stored into the concept <u>apple</u>. Similarly our knowledge about checking account balances is what makes us to produce the single concept <u>old-balance</u> when we read "the previous balance", "the old balance", "the past balance", or

"the balance at the starting of the period".

When DESCRIPTION is entered, the adjective or modifying noun is stored in the variable MODIF, and the head noun in H-NOUN. When both variables have a value, a function, whose name is stored in the dictionary definition of each word, is invoked. These functions are a collection of very simple production rules. A typical production rule of the function CN-BALACE (the function for the noun "balance") is:

If the marker PAST (a time marker for "old" and "past") or the marker PREVIOUS (a time marker for "previous", "beginning" etc) belongs to the adjective modifying "balance", return OLD-BAL.

Not all adjectives modifying a noun will activate the function for that noun. Only the non operational adjectives will do it. By "operational adjectives" we mean those adjectives whose meaning can be mapped to a computer program. For ex., "even", "divisible", and also all ordinals from "first" to "last". "The last account" will be parsed into (ACCOUNT ORDINAl (LAST)). All non operational adjectives are tagged in the dictionary with the marker NOPAD.

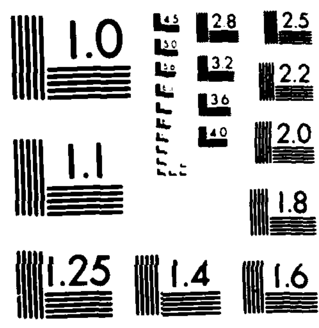
3.3 Clauses Completing the Description.

Relative clauses, -ing forms, two place adjectives or past participles and prepositions are used to complete the description of objects. By two place adjectives we mean those predicates that take two arguments, i.e, " - greater than -" or " - influenced by -". All these terms are tagged in the dictionary with the marker SPEC (for specification).

AD-A127 793 THEORETICAL FOUNDATIONS OF SOFTWARE TECHNOLOGY(U) OHIO STATE UNIV COLUMBUS DEPT OF COMPUTER AND INFORMATION SCIENCE B CHANDRASEKARAN ET AL. 14 FEB 83 2/2

UNCLASSIFIED AFOSR-TR-83-0333 F49620-79-C-0152 F/G 9/2 NL

END
DATE
FILMED
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

DESCRIPTION (before exiting) checks if the word it is examining has the marker SPEC. If so, it considers the possibility that it may be a clause completing the DESCRIPTION. There is no general mechanism to determine if a clause following a noun group is modifying the noun group or not. The classic example is : "The man with the broken leg killed Peter". Our parser uses semantic clues to make that decision. If the concept expressed by the noun group is an individual concept [22], the clause following the noun group can not be completing the description. By "individual concept" Carnap means those concepts that denote a single entity. For example, "John's wife with the broken leg killed Peter". Howerver funny the sentence may be, the "broken leg" is the instrumental case. In the case of -ing forms, if the object denoted by the noun group is inanimate, it can be safely assumed that the -ing clause is a part of the description. .

Phrases completing descriptions (PCD) present a similar problem to that of compound nouns, namely to identify the concept denoted by the description. The problem is more complex, because in the case of complex nominals we know we are dealing with a single concept. However when we deal with a description completed by a phrase, we do not know if the PCD forms part of the definition of the concept or is expressing an accidental property like time, location etc.. For instance

(1) The man on the street does not give a damm for politics
(2) The man on the street is from Ohio

In 1) "on the street" is obviously a location, but in (2) it is a predicate of the concept man. "The man on the street" in (2) refers to the concept ordinary man. A similar case we have in "the balance at the beginning of the period". It does not refer to a balance that is temporarily situated at the beginning of a period of time, but to the abstract notion of a previous-balance. In fact, many complex nominals have been formed by deletion of prepositions [21]. Our solution to this problem has been to ask the concept denoted by the noun group about the meaning of the clause completing the description. Our implementation applies only to descriptions completed by prepositional phrases. Thus, the routine associated with a preposition (see below) may decide to ask the concept denoted by the noun group about the meaning of the prepositional phrase as applied to that concept. Let us consider: "the balance at the beginning of the period". The routine for "at" asks DESCRIPTION to obtain the concept denoted by "the beginning of the period". DESCRIPTION returns with "ST-PERIOD" that means the beginning of a period of time. Then the routine "at" invokes CN-BALANCE with the argument "ST-PERIOD", and this one returns with "OLD-BAL". The conditions under which a prepositional routine invokes the preceding concept need futher study. We have come up with the following criterion: if the concept preceding the prepositional phrase is an abstract concept, and the the concept underlying the prepositional phrase has the semantic features of TIME or LOCATION, the prepositional routine will invoke the preceding concept. By

"abstract concept" we mean a concept that is not a picture-producer [23].

3.4 Verbal Concepts

We have used different levels of abstraction in tagging the verbs in the dictionary. We have marked the verbal surface forms much as a dictionary does it. Thus, we have used the marker SUPL to tag in the dictionary "supply", "provide", "furnish", but not "offer". From the highest level of abstraction, all of them, "offer" included, are tagged with the marker ATRANS [23]. All action verbs that may have an operational meaning are tagged with the marker OPER. The most obvious operational verbs are: add, substract etc.. Others are delete, store, move.

What level of abstraction a system must have present in order to "understand" a sentence in a given context, is a hard question. The highest level of abstraction will facilitate the matching, paraphrase and the understanding of verbs used in contexts somewhat inappropiate. On the other hand, a system that only "knows" about "walk" that is an instance of PTRANS [23], will not understand the second sentence in : "Peter walked 20 miles home. His feet were swollen".

Descriptive verbs (D-VERBS) are those used to describe. In the programming examples we have studied, we have found four semantic classes of D-VERBS. There are those that describe the constituents of an object. Among them are : consist of, show,

include, be given by, contain etc.. We refer to them as CONSIST-OF D-VERBS. A second class are those used to indicate that something is representing something. Represent, indicate, mean, describe etc. belong to this class. We refer to them as REPRESENT D-VERBS. A third class are those that fall under the notion of appear. To this class belong appear, belong, be given on etc. We refer to them as APPEAR D-VERBS. The fourth class are formed by those that express a spatial relation. Some of these are : follow, precede, be + any spatial verb. We refer to them as SPATIAL D-VERBS.

A routine called ACTION-VERB parses the action verbs. There are markers in the dictionary for the cases they take and the prepositions used for the cases other than the transitive case. For instance, in the dictionary definition of "move" is indicated that it is a transitive verb and that it also takes a destination case, and that the prepositions with the destination case may be "to" or "into". Similarly, the function DESCRIPTIVE-VERB parses the descriptive verbs.

3.5 Prepositions

For each preposition (also for each conjuction) in the lexicon there is a function. The name of the function is stored in the dictionary definition of each preposition. When the parser finds a preposition its function is activated. The function of these prepositional experts is to determine the meaning of the preposition.

The first thing they do is to test if the preposition introduces a case for the preceding word. For example, the expert for "of" will check if the preceding word has the feature CASE-OF, a feature associated with words such as : "number of", "abuse of", "collection of", "result of", "destruction of" etc.. The same check will be done by the function for "for". It will look for : "lust for", "pressure for" etc.. In the most interesting cases, the meaning of a preposition can not be determined by looking at adjacent words. Thus the prepostional experts suspend themselves and ask DESCRIPTION to parse for them the concept denoted by the prepositional phrase. DESCRIPTION returns to them the concept with all its semantic features. Two group of production rules follow. The first group is activated if the verb of the sentence has not been parsed, the second one otherwise. Obviously it is simpler to determine the meaning of a preposition in the first case. The decision is primarialy based on the semantic features of the concept denoted by the prepositional phrase. If the verb has been processed, the whole conceptualization underlying the sentence must be used by the expert in order to decide what is the meaning of the preposition. For example, "John takes wine for his depression". There is a rule in FOR-SP (the prepositional expert for "for") that says that if the feature ANIMATE belongs to the subject of the sentence and the verb is an instance of INGEST [23], , and the concept denoted by the prepositional phrase has the feature PHYPSY (a physical or mental state) then

"for" express the idea of doing something with the PURPOSE-OF-ALLEVIATING a physical or mental state. The meaning of the verb is taken in its highest level of abstraction. In most cases, the meaning of the preposition depends on the concept denoted by the prepositional phrase. Compare the above sentence with "John takes wine for lunch". When a context has been established, high level sources of knowledge can anticipate the meaning of prepositions. We will see some examples in the next section.

4. Parsing into Knowledge Structures

As we said in the introduction, the concepts relevant to a programming topic are grouped in a passive frame. We distinguish between those concepts which are relevant to a specific programming task, like balance to checking-account programs, and those relevant to any kind of program, like output, input end-of-data etc.. The former can be only recognized when the programming topic has been identified. A concept like output will not only be activated by the word "output" or by a noun group containing that word. The verb "print" will obviously activate that concept. Any verb that has the feature REQUEST, a semantic feature associated with such verbs as "like", "want", "need", etc., will activate also the concept output. Similarly nominal concepts like card and verbal concepts like record, a semantic feature for verbs like "record", "punch" etc., are just two examples of concepts that will activate the input specialist.

The recognition of concepts is as follows: Each time that a new sentence is going to be read, a global variable RECOG is initialized to NIL. Once a nominal or verbal concept in the sentence has been parsed, the function RECOGNIZE-CONCEPT is invoked (if the value of RECOG is NIL). This function checks if the concept that has been parsed is relevant to the programing task in general or (if the topic has been identified) is relevant to the topic of the programming example. If so, RECOGNIZE-CONCEPT sets RECOG to T and passes control to the concept that takes control overriding the parser. Once a concept has been recognized, the specialist for that concept continues in control until the entire sentence has been processed. The position that a relevant concept occupies in the sentence is not an impediment for that concept to take control, except if the concept is in a prepositional phrase that starts a sentence.

The following data structures are used during parsing. A global variable, STRUCT, holds the result of the parsing. STRUCT can be considered as a STM for the low level linguistic processing. A BLACKBOARD [24] holds the high level recomendations, messages etc that the high level conceptual specialists pass to the low level linguistic experts and among them. Because the information in the blackboard does not go beyond the sentential level, it may be considered as STM for the high level sources of knowledge. A global variable WORD holds the word being examined, and WORDSENSE holds the semantic features of that word.

Example 1

An instructor records the name and five test scores on a data card for each student. The registrar also supplies data cards containing a student name, identification number and number of courses passed.

The parser is invoked by activating SENTENCE. Because "an" has the marker DESCR, SENTENCE passes control to DECLARATIVE which handles sentences starting with a nominal phrase. (There are other functions that respectively handle sentences starting with a prepositional phrase, an adverbial clause, a command, an -ing form, and sentences introduced by "to be" (there be, will be etc.) with the meaning of existence.) DECLARATIVE invokes DESCRIPTION. This parses "an instructor" obtaining the concept <u>instructor</u>. Before returning control, DESCRIPTION activates the functions RECOGNIZE-TOPIC and RECOGNIZE-CONCEPT. The former function checks in the dictionary if there is a frame associated with the concept parsed by DESCRIPTION. The frame EXAM-SCORES is associated with <u>instructor</u>, then the variable TOPIC is instantiated to that frame. The recognition of the frame that may be a very hard problem [25] is very simple in the programing problems we have studied and normally the first guess happens to be correct. Next, RECOGNIZE-CONCEPT is invoked. Because <u>instructor</u> does not belong to the relevant concepts of the EXAM-SCORES frame, it returns control. Finally DESCRIPTION returns control to DECLARATIVE, along with a list containing the semantic features of <u>instructor</u>. DECLARATIVE, after checking that the feature TIME does not belong to those features, inserts SUBJECT before "instructor" in STRUCT.

Before storing the content of WORD, "records", into STRUCT, DECLARATIVE invokes RECOGNIZE-CONCEPT to recognize the verbal concept. All verbs with the feature record, as we said above, activate the input specialist.

When INPUT-SP is activated, STRUCT looks like (SUBJ (INSTRUCTOR)). As we said in the introduction, the INPUT specialist is a collection of production rules. One of those rules says :

IF the marker RECORD belongs to WORDSENSE then Activate the function ACTION-VERB and pass the following recommendations to it : 1)activate the INPUT-SUPERVISOR each time you find an object 2) if a RECIPIENT case is found then if it has the feature HUMAN, parse and ignore it. Otherwise awaken the INPUT-SUPERVISOR 3) if a WHERE case (the object where something is recorded) is found, awaken the INPUT-SUPERVISOR.

The INPUT-SUPERVISOR is a function that is controlling the input for each particular problem. ACTION-VERB parses the first object and passes it to the INPUT-SUPERVISOR. This ckecks if the semantic feature IGENERIC (this is a semantic feature associated with words that refer to generic information like "data", "information" etc) does not belong to the object that has been parsed by ACTION-VERB. If that is not the case, the INPUT-SUPERVISOR, after checking that name is normally associated with the input for EXAM-SCORES, inserts it into the CONSIST-OF slot of input. The INPUT-SUPERVISOR returns control to ACTION-VERB that parses the next object and the process explained above is repeated.

When ACTION-VERB finds the preposition "on", the routine ON-SP is activated. This, after checking that the main verb of the sentence has been parsed and that it takes a WHERE case, checks the BLACKBOARD to find out if there is a recommendation for it. Because that is the case, ON-SP tells DESCRIPTION to parse the nominal phrase "on data cards". This returns with the concept card. ON-SP activates the INPUT-SUPERVISOR with card. This routine, after checking that cards is a type of input that the solver handles, inserts "card" in the INPUT-TYPE slot of input and returns control. What if the sentence had said "... on a notebook" ? Because notebook is not a form of input, the INPUT-SUPERVISOR would have not inserted "book" into the INPUT-TYPE slot. Another alternative is to let the INPUT-SUPERVISOR insert it in the INPUT-TYPE slot and let the problem solver make sense out of it. There is an interesting tradeoff between understanding and problem solving in these contexts. The robuster the understander is, the weaker the solver may be, and vice versa. The prepositional phrase "for each student" is parsed similarly. ACTION-VERB returns control to INPUT-SP that inserts "instructor" in the SOURCE slot of input. Finally, it sets the variable QUIT to T to indicate to DECLARATIVE that the sentence has been parsed and returns control to it. DECLARATIVE after checking that the variable QUIT has the value T, returns control to SENTENCE. This resets the variables RECOG, QUIT and STRUCT to NIL and begins to examine the next sentence.

The calling sequence for the second sentence is identical to that for the first sentence except that the recognition of concepts is different. The passive frame for EXAM-SCORES does not contain anything about "registrar" nor about SUPL (see sec. 3.4). DECLARATIVE has called ACTION-VERB to parse the verbal phrase. This has invoked DESCRIPTION to parse the object "data cards". STRUCT looks like : (SUBJ (REGISTRAR) ADV (ALSO) AV (SUPPLIES) OBJ). ACTION-VERB is waiting for DESCRIPTION to parse "data cards" to fill the slot of OBJ. DESCRIPTION comes with <u>card</u> from "data cards", and invokes RECOGNIZE-CONCEPT. The specialist INPUT-SP is connected with <u>card</u> and it is again activated. This time the production rule that fires says :

If what follows in the sentence is <universal quatifier> + <D-VERB> or simply D-VERB then activate the function DESCRIPTIVE-VERB and pass it the recommendation of activating the *INPUT-SUPERVISOR each time a complement is found.*

The pattern <universal quantifier> + <D-VERB> appears in the antecedent of the production rule because we want the system also to understand : "data cards each containing...". The rest of the sentence is parsed in a similar way to the first sentence. The INPUT-SUPERVISOR returns control to INPUT-SP that stacks "registrar" in the source slot of <u>input</u>. Finally the concept <u>input</u> for this problem looks :

```
INPUT  CONSIST-OF  (NAME (SCORES CARDI (5)))  SOURCE
(INSTRUCTOR)
                   (NAME P-COURSES) SOURCE (REGISTRAR)
       INPUT-TYPE        (CARDS)
```

If none of the concepts of a sentence are recognized - that is the sentence has been parsed and the variable RECOG is NIL - the system prints the sentence followed by a question

mark to indicate that it could not make sense of it. That will happen if we take a sentence from a problem about checking-accounts and insert it in the middle of a problem about exam scores. The INPUT-SP and the INPUT-SUPERVISOR are the same specialists. The former overrides and guides the parser when a concept is initially recognized, the latter plays the same role after the concept has been recognized. The following example illustrates how the INPUT-SUPERVISOR may futhermore override and guide the parser.

The registrar also provides cards. Each card contains data including an identification number ...

When processing the subject of the second sentence, INPUT-SP is activated. This tells the function DESCRIPTIVE-VERB to parse starting at "contains ..." and to awaken the INPUT-SUPERVISOR when a object is parsed. The first object is "data" that has the marker IGENERIC that tells the INPUT-SUPERVISOR that "data" can not be the value for the input. The INPUT-SUPERVISOR will examine the next concept looking for a D-VERB. Because that is the case, it will asks the routine DESCRIPTIVE-VERB to parse starting at "including an identification number..."

Example 2

We will comment briefly on the first six sentences of the example in fig. 1. There is a specialist that has grouped the knowledge about checking-accounts. This specialist, whose name is ACCOUNT-SP, will be invoked when the parser finds a concept

that belongs to the slot of relevant concepts in the passive frame. The first sentence is: "A bank would like to produce... checking accounts". The OUTPUT-SP is activated by "like". When OUTPUT-SP is activated by a verb with the feature of REQUEST, there are only two production rules that follow. One that considers that the next concept is an action verb, and another that looks for the pattern <REPORT + CONSIST D-VERB> (where "REPORT" is a semantic feature for "report", "list" etc.). In this case, the first rule is fired. Then ACTION-VERB is activated with the recommendation of invoking the OUTPUT-SUPERVISOR each time that an object is parsed. ACTION-VERB awakens the OUTPUT-SUPERVISOR with (RECORDS ABOUT (TRANSACTION)). Because "record" has the feature IGENERIC the OUTPUT-SUPERVISOR tries to redirect the parser by looking for a CONSIST D-VERB. Because the next concept is not a D-VERB, OUTPUT-SUPERVISOR sets RECOG to NIL and returns control to ACTION-VERB. This parses the adverbial phrase introduced by "during" and the prepositional phrase introduced by "with". ACTION-VERB parses the entire sentence without recognizing any relevant concept, except the identification of the frame that was done while processing "a bank".

The second sentence "For each account the bank wants ... balance." is parsed in the following way. Although "account" belongs to slot of relevant concepts for this problem, it is skipped because it is in a prepositional phrase that starts a sentence. The OUTPUT-SP is activated by a REQUEST type verb, "want". STRUCT looks like : (RECIPIENT (ACCOUNT UQ (EACH))

SUBJECT (BANK)). The production rule whose antecedent is <RECORD + CONSIST D-VERB> is fired. The DESCRIPTIVE-VERB function is asked to parse starting in " showing", and activate the OUTPUT-SUPERVISOR each time an object is parsed. The OUTPUT-SUPERVISOR inserts all objects in the CONSIST-OF slot of output, and returns control to the OUTPUT-SP that inserts the RECIPIENT, "account", in the CONSIST-OF slot of output and returns control.

The next sentence is "The accounts and transactions ... as follows:" DECLARATIVE asks DESCRIPTION to parse the subject. Because account belongs to the relevant concepts of the passive frame, the ACCOUNT-SP specialist is invoked. There is nothing in STRUCT. When a topic specialist is invoked and the next word is a boolean conjunction, the specialist asks DESCRIPTION to get the next concept for it. If the concept does not belong to the list of relevant concepts, the specialist sets RECOG to NIL and returns control. Otherwise it continues examining the sentence. Because transaction belongs to the slot of relevant concepts of the passive frame, ACCOUNT-SP continues in control. ACCOUNT-SP finds "for" and asks DESCRIPTION to parse the nominal phrase. ACCOUNT-SP ignores anything that has the marker HUMAN or TIME. Finally ACCOUNT-SP finds the verb, a APPEAR D-VERB, and invokes the DESCRIPTIVE-VERB routine with the recommendation of invoking the ACCOUNT-SUPERVISOR each time a complement is found. The ACCOUNT-SUPERVISOR is awakened with card. This inserts "card" in the INPUT-TYPE slot of account and transaction and returns control to the DESCRIPTIVE-VERB

routine. AS-SP (the routine for "as") is invoked next. This, after finding "follows" followed by ":", indicate to DESCRIPTIVE-VERB that the sentence has been parsed. ACCOUNT-SP returns control to DECLARATIVE and this, after checking that QUIT has the value T, returns control to SENTENCE.

The next sentence is: "First will be a sequence of cards ... accounts." The INPUT-SP specialist is invoked. STRUCT looks like : (ADV (FIRST) EXIST). "Sequence of cards" gives the concept card activating the INPUT-SP specialist. The next concept is a REPRESENT D-VERB. INPUT-SP activates the DESCRIPTIVE-VERB routine and asks it to activate the INPUT-SUPERVISOR each time an object is found. The INPUT-SUPERVISOR checks if the object belongs to the relevant concepts for checking accounts. If not, the ACCOUNT-SUPERVISOR will complain. That will be the case if the sentence is : "First will be a sequence of cards describing the students". Assume that the above sentence says : "First will be a sequence of cards consisting of an account number and the old balance." In that case, the INPUT-SP will activate also the INPUT-SUPERVISOR but because the verbal concept is a CONSIST D-VERB, the INPUT-SUPERVISOR will stack the complements in the slot for INPUT. Thus, what the supervisor specialists do depend on the verbal concept and what is coming after.

The next sentence is: "Each account is described by ..., in dollars and cents." Again, the ACCOUNT-SP is activated. The next concept is a CONSIST D-VERB. ACCOUNT-SP assumes that it

is the input for accounts and activates the DESCRIPTIVE-VERB function, and passes to it the recommendation of activating the INPUT-SUPERVISOR each time an object is parsed. The INPUT-SUPERVISOR is awakened with (NUMBERS CARDINAL (2)). Because number is not an individual concept (like,say, 0 is) the INPUT-SUPERVISOR reexamines the sentence and finds ":", it then again asks to DESCRIPTIVE-VERB to parse starting at "the account number...". The INPUT-SUPERVISOR stacks the complements in the input slot of the concept that is being described : account.

The next sentence is: "The last account is followed by ... to indicate the end of the list." The ACCOUNT-SP is invoked again. The following production rule is fired: If the ordinal "last" is modifying "account" and the next concept is a SPATIAL D-VERB then activate the END-OF-DATA specialist. This assumes control and asks DESCRIPTIVE-VERB to parse starting at "followed by" with the usual recommedation of awakening the END-OF-DATA supervisor when a complement is found, and the recommendation of ignoring a PURPOSE clause if the concept is end-of-list or end-of-account. The END-OF-DATA is awakened with "dummy-account". Because "dummmy-account" is not an individual concept, the END-OF-DATA supervisor reexamines the sentence expecting that the next concept is a CONSIST D-VERB. It finds it, and redirects the parser by asking the DESCRIPTIVE-VERB to parse starting in "consisting of two zero values". The END-OF-DATA is awakened with "(ZERO CARD (2))". Because this time the object is an individual concept, the

END-OF-DATA supervisor inserts it into the END-OF-DATA slot of the concept being described: account.

5. A Checking-Account Programmer

The solver receives as input a list containing the name of the concepts that the parser has built. In principle, any account problem is decomposed in the following steps: 1) Read the accounts 2) Read and process transactions 3) Print reults . For each of these conceptual steps, there is a problem solving specialist that "knows" what to do. There are routine actions, the same for a large class of problems, that these specialists must perform. For instance, the READ-ACCOUNTS specialist get the value of the slot for the input of the accounts. Then it checks the consistency of the input. For example, it checks if there is at least some kind of identification (account number,name etc). Next it examines each of the arguments of the input to find out what type of variable (real, character, etc.) must be declared. Finally it gets the value of the end-of-data slot of account to check consistency or to do something about it, if the user forgot to indicate the end of data.

Obviously our purpose has been to build a problem solver that will have the capability of solving a large class of checking-account programs, not just the one in fig.1. In particular, the problem solver we have implemented can produce a program for any combination of the following outputs : 1) All accounts whose final balance is greater than the old

balance, 2) All accounts whose old balance is less than $24.00 3) All overdrawn accounts, 4) All accounts whose number of deposits is greater than the number of wihtdrawals etc..

6.1 The Planner

Consider the problem of fig.1 and assume that the output is statement 1) above. In that statement, it is not said that the final balance must be computed and that the old balance, when reading the accounts, must be saved. The function of the PLANNER consists of examining each one of the output statements and telling each problem solving specialist what to do. Each specialist has a list, called the task list. When the PLANNER examines the output statements, it fills these lists with the computations each specialist must perform. For example, if PLANNER examines statement 1) it will insert "final-bal" in the task list of the specialist for processing transactions, and "old-bal" in the list for the specialist for reading the accounts. The insertion will take place if "final-bal" or "old-bal" are not already there.

Beside assigning individual concepts to each problem solving specialist, the PLANNER must determine how the output statements as a whole must be computed. The problem solver assumes that the output statements must be printed in the order the user has formulated them. Thus, statement 1) of above has to be printed after the transactions have been computed, and is, therefore, assigned to the PRINT-RESULTS specialist. But, the statement 2) of above, if it is the first output statement,

can be printed while reading the accounts itself. Nevertheless, if other output statements are preceding it, the most appropriate thing to do is (while reading the accounts) to save in one array those accounts whose old balance are less than $24.00 and in another array the old balance, then print the content of these arrays at the approppiate time. Thus the PLANNER creates two array names,say A and B, and inserts the statement (PRINT (A B)) in the task list of PRINT-RESULTS and the following statement in the task list of the READ-ACCOUNTS specialist: [((ACCOUNTS UQ (ALL)) WHOSE (SUBJ (OLD-BAL) LESS (24)) STORE (ACCOUNT OLD-BAL) INTO (A B)] The list preceding "STORE" is the parser output for statement 2). The remainder in the list has been introduced by the PLANNER. When READ-ACCOUNTS gets to this statement, it checks that ACCOUNT belongs to the input for the problem and asks WHOSE-TR to translate the whose clause into a programming language statement. We have implemented a WHOSE-TR function that handles clauses with several subjects or predicates, not just the simple example of above. Then READ-ACCOUNTS asks STORE-SP, a low level function, that "knows" how to store the content of arrays into other arrays, to translate the remaining of the list.

We hope that we have conveyed an idea of how our checking-accounts programmer works. We are writing the code necessary for the program to handle checking-accounts programs with essentially different input conditions. For instance, all the transactions for an account are grouped following the

account number.

6. Comparison with Related Approaches and Future Reseach

The work by Rieger and Small [26] has influenced our research. Our prepositional experts are modelled after their word experts. Rieger's idea that individual words have contextual knowledge about its various uses can hardly be contested. But we do not go along with the idea of building an expert for each word. Words are surface manifestations of something deeper. In our parser, concepts and not words are the guiding principles.

In our approach to natural language, we have had present the view expressed in [23], [27], [28], and [15] that natural language comprehesion is an integrated process in which high level sources of knowledge guide low level processes. We have already indicated the main difference between [15] and the present work. In our opinion, one of the most sticky problems with Schank's and his collaborators systems has been its difficulty in dealing with new texts. We think that our concept of high level sources of knowledge overriding low level linguistic knowledge allows our system to handle new problem areas with not too much difficulty. Assume that we want LLULL to understand programming problems about roman numerals, say. We are going to find uses of verbs, prepositions etc. that our low level parser will not handle. We integrate those uses in our parser (its modular nature makes that integration relatively simple). On top of that, we will build several high

level specialists that will have knowledge about roman numerals. In our future research, we are going to extend the breadth of the system by augmenting both its low level linguistic knowledge and the themes it is able to understand. At the same time, we are going to increase its depth on checking account programs, in such a way that it will be able to understand and solve any "programming story" about checking accounts.

7. A computer run

(A BANK WOULD LIKE TO PRODUCE RECORDS OF THE TRANSACTION DURING AN ACCOUNTING PERIOD IN CONNECTION WITH THEIR CHECKING ACCOUNTS. FOR EACH ACCOUNT THE BANK WANTS A LIST SHOWING THE BALANCE AT THE BEGINNING OF THE PERIOD, THE NUMBER OF DEPOSITS AND WITHDRAWALS, AND THE FINAL BALANCE. THE ACCOUNTS AND TRANSACTIONS FOR AN ACCOUNTING PERIOD WILL BE GIVEN ON PUNCHED CARDS AS FOLLOWS: FIRST WILL BE A SEQUENCE OF CARDS DESCRIBING THE ACCOUNTS. EACH ACCOUNT IS DESCRIBED BY TWO NUMBERS: THE ACCOUNT NUMBER (GREATER THAN 0), AND THE ACCOUNT BALANCE AT THE BEGINNING OF THE PERIOD, IN DOLLARS AND CENTS. THE LAST ACCOUNT IS FOLLOWED BY A DUMMY ACCOUNT CONSISTING OF TWO ZERO VALUES TO INDICATE THE END OF THE LIST. THERE WILL BE AT MOST 200 ACCOUNTS. FOLLOWING THE ACCOUNTS ARE THE TRANSACTIONS. EACH TRANSACTION IS GIVEN BY THREE NUMBERS: THE ACCOUNT NUMBER, A 1 OR -1 (INDICATING A DEPOSIT OR WITHDRAWAL, RESPECTIVELY), AND THE TRANSACTION AMOUNT, IN DOLLARS AND CENTS. THE LAST REAL TRANSACTION IS FOLLOWED BY A DUMMY TRANSACTION CONSISTING OF THREE ZERO VALUES.)

fig.1
(From An Introduction to Programming (Conway and Gries, 1975)

```
OUTPUT  CONSIST-OF (ACCOUNT OLD-BAL DEPOSITS WITHDRAWALS FINAL-BAL)
ACCOUNT INPUT (ACCOUNT-NUMBER SPEC GREATER (0) OLD-BAL SPEC (DOLLAR-CENT))
        INPUT-TYPE (CARDS)
        END-OF-DATA (( ZERO CARD (2)))
        NUMBER-OF-ACCOUNTS (200)
TRANSACTION INPUT (ACCOUNT-NUMBER (1 OR -1) REPRESENT (DEPOSIT OR WITHDRAWAL)
TRANS-AMOUNT)
            INPUT-TYPE (CARDS)
            END-OF-DATA ((ZERO CARD (3)))
```

fig. 2 Parser Output for problem of fig. 1

REFERENCES

(1) Biermann, A., "Approaches to Automatic Programming", in Advances in Computers, M. Rubinoff and M. C. Yovits, (eds) Academic Press, 1976

(2) Elschlager, R. and Phillips, J. Automatic Programming, Memo HPP-79-24, Report STAN-CS-79-758, Computer Science Department, Stanford, November 1979

(3) Hammer, M. and Ruth, G.,"Automating the Software Development Process.", in Research Directions in Software Technology Wegner (ed.), MIT Press, 1979

(4) Phillips, J., and Green, C., "Towards Self-Described Programming environments", Computer Science Dep., Systems Control Inc., Palo Alto, CA., 1980

(5) Rich, C., and Shrobe, H., "Design of a Programmer's Apprentice", in Artitificial Intelligence: An MIT Perspective, MIT press, 1979

(6) Balzer, R. and Goldman, N., "Principles of Good Software Specification and Their Implications for Specification Languages," IEEE Proceedings Specifications of Reliable Software, Boston, 1979

(7) Balzer, R., "Transformational Implementation: An Example", USC Information Sciences Institute, Marina del Rey, Ca. 90291, 1979

(8) Biermann, A. W. and Ballard, B., "Toward Natural Language Computation", AJCL, Vol. 6, Number 2, 1980

(9) Novak, G. S., "Computer Understanding of Physics Problems Stated in Natural Language", AJCL Microfiche 53, 1976

(10) Bobrow, D. G. "Natural Language Input for a Computer Problem-Solving System", in Semantic Information Processing, Minsky (ed.), MIT, 1968

(11) Charniak, E. "CARPS, A Program which Solves Calculus Word Problems", Report MAC-TR5, MIT, 68

(12) Heidorn, G. E. "Natural Language Inputs to a Simulation Programming System", NPS-55HD72101A, Naval Postgraduate School, Monterry, Ca., 1972

(13) Hayes, J. R., Simon, H. A., "Understanding Writen Problem

Instructions", in Knowledge and Cognition, L. W. Greeno (ed.), Lawrence Erlbaum Associates, 1974

(14) Ginsparg, J., "Natural Language Processing in An Automatic Programming domain", Stanford, Computer Science Dept., AIM-316, 1978

(15) Schank, R., Lebowitz, M., and Birnbaum, L. "Parsing Directly Into Knowledge Structures", in IJCAI-79

(16) Wirth, N. "Program development by stepwise refinement.", CACM 14, 1971

(17) Gomez F., and Chandrasekaran B. "Knowledge Organization and Distribution for Medical Diagnosis.", IEEE Trans. Syst., Man, Cybern., vol. SMC-11, no. 1

(18) Newell A., and Simon. H. A., Human Problem Solving, Englewood Cliffs, N.J., Prentice-Hall, 1972

(19) Strawson P. F. "On Referring", MIND, 1950

(20) Gershman A. V. "Conceptual Analysis of Noun Group in English", IJCAI-77

(21) Levi J. N. The Syntax and Semantics of Complex Nominals, Academic Press, 1978

(22) Carnap, R. Meaning and Necessity, University of Chicago Press, 1950

(23) Schank R. C. (ed.) Conceptual Information processing, North Holland, 1975

(24) Erman L. D. and Lesser V. R. "A Mult-Level Organization for Problem-Solving Using Many Diverse Cooperating Sources of Knowledge", IJCAI-75

(25) Charniak E. "With a spoon in hand this must be the eating frame", in Proc. Conf. Theoretical Issues in Natural Language, 1978

(26) Rieger, C. and Small S., "Towards a theory of distributed word expert natural language parsing", in IEEE Trans. Syst., Man, Cybern., vol.SMC-11, no. 1

(27) Wilks, Y. "An Artificial Intelligence Approach to Machine Translation", in Computer Models of Thought and Language, R. C. Schank and K. Colby (eds), W. H. Freeman and Co., San Francisco

(28) Risbeck, C. and Schank, R. "Comprehesion by Computer: Expectation-based Analysis of Sentences in Context", Res. Rept. 78, Yale University, 1976

2.2.2. Natural and Social System Metaphors in Distributed Problem Solving: Introduction to the Issue

IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS, VOL. SMC-11, NO. 1, JANUARY 1981

Natural and Social System Metaphors for Distributed Problem Solving: Introduction to the Issue

B. CHANDRASEKARAN, SENIOR MEMBER, IEEE

Abstract—Naturally occurring information systems provide a number of useful metaphors for distributed problem solving. An introduction to some aspects of these metaphors is given which simultaneously serves as a guest editorial for a special issue devoted to the topic. The ubiquity of a distributed mode of computation in information processing in natural phenomena in general and in human societies in particular is observed and related to the evolutionary and complexity-reducing advantages of this mode. The forms of communication media available to coordinate the problem solving activities of the individual processors are examined. Some general remarks are made on how problem solving is distributed and coordinated in some human organizations, and the potential usefulness of the "society of specialists" notion in explicating cognitive activity is pointed out. Along the way, the contents of the papers in the special issue are considered in relation to various points raised in the discussion.

I. Ubiquity of Distributed Information Processing

ISSUES about *distributed* computing, as about any other aspect of computing, can be formulated at various levels of abstraction. Each level has a different conceptual content, and raises a correspondingly different set of issues. In distributed computing most of the recent emphasis has been at a level that is closely related to physical connection of different processors, secure transmission of data among them, and the corresponding operating system problems of scheduling different processors. These issues have dominated discussion so much that the term distributed processing has come to mean almost exclusively that set of issues. The papers in this special issue deal with distributed processing at a different, "higher," level of abstraction. The questions of interest at this level concern the strategies by which the decomposition and coordination of computation in a distributed system are matched to the structural demands of the task domain. Distributed problem solving (DPS) is an appropriate term for the phenomena at this level of abstraction.

As the theme of the issue implies, a motivating belief is that information processing phenomena that occur in the natural world are a source of a number of useful metaphors for distributed processing in general and distributed problem solving in particular. It is clear that distribution of processing or computation is an intrinsic characteristic of most natural phenomena which can be captured within a computational or symbol processing framework. Social organizations from honeybee colonies to a modern corporation, from bureaucracies to medical communities, from committees to representative democracies are living examples of distributed information processing embodying a variety of strategies of decomposition and coordination. Computation in biological brains, especially in their sensory processors such as vision systems, displays a high degree of distribution. There is substantial evidence that higher cortical functions are also computed (and controlled) in the brain in an essentially distributed mode: regions have been identified in the cortex whose activities are highly correlated with specific higher cortical functions such as language processing. Geschwind [1] indicates that some regions in the brain are extremely specialized: there is an identifiable processor which specializes in human face recognition!

Control of movements in biological systems is also accomplished by distributed computation [2]. Evidence is available that control of normal walking movements resides in the spinal cord [3]. Volitional movement can be viewed as being generated by low-level programs coordinated and regulated by higher level controllers. The task of generating all the impulses for all muscle fibers for each movement is surely beyond the resources of any centralized biological movement processor.

In all these examples—from social organizations to brains and motor systems—the overall computational task is distributed among a collection of separate processors. These separate processors coordinate their computations by means of exchanging appropriate symbolic information.

II. Why (and Why Not) Distribution

A. Advantages of Distribution

Why should distributed computation be such a ubiquitous mode in naturally evolved information processing systems? The following advantages of distribution may be relevant here.

1) *Decomposition* of processing is an absolutely basic strategy for controlling the *complexity* of computation. Central computation is just too costly in both memory and time. Distributing the computation among different

Manuscript received September 20, 1980. This work was supported in part by the Air Force Office of Scientific Research under Contract F 49620-79c-0152.

The author is with the Artificial Intelligence Group, Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210.

0018-9472/81/0100-0001$00.75 ©1981 IEEE

processors generates possibilities for parallel activities by different processors which may be able to work on essentially nonoverlapping segments of the data most of the time. Also the scope of each processor is limited, i.e., the size of the input domain is much smaller. Complexity of computation is often an exponential function of input space size.

2) Appropriately distributed computing increases the prospects for graceful degradation of response when there is degradation of input data or failure of portions of the system.

3) Distribution is a natural attribute of *evolutionary* systems. As the system grows and increases in complexity, a distributed mode provides for replacing a processor with several processors and making *mostly* local changes in linkage among processors; or, as the external environment changes, distributed information processing makes *adaptation to change* easier, since again, as long as the rate of external change is not large, changes to the system can be mostly local, if the original decomposition reflected the structure of the task environment correctly. As programmers well know, these are really advantages of *modularity*, but a distributed architecture provides a natural means of implementing this modularity.

4) In complex information processing systems involving very large numbers of *sensors* and *effectors*, a central processor will require *very large bandwidths* for responding to sensors or activating effectors. Imagine an army whose commanding general alone is authorized to make *all* the field decisions!

5) Often a task decomposition will lead to the generation of a large number of identical subtasks, each, however, operating over different subsets of input data or regions of the environment. In this case, once a particular processor is optimized to be effective for such a subtask, in a distributed approach this processor can be *replicated* as often as needed. This provides for considerably increased efficiency due to parallelism, and is especially useful in systems that deal with large volumes of sensory information. However, it is useful in a variety of other situations also. An example is the optimization of the training of salespersons in a commercial organization and consequent production of large numbers of them.

B. Possible Costs of Distribution

Of course there could sometimes be a cost associated with distribution as opposed to a central computation. The computation of each processor is often a filtering operation, in that it communicates only the *result* of its computation. (Thus in a vision system a higher level processor may not have direct access to the image intensity data, but only to the outputs of edge detectors operating on these data.) Another processor may arrive at a different result with the same data if it did not depend on the computation by the other processor. Several strategies are used in naturally occurring distributed systems to deal with this problem. Marr [4] talks about the *principle of least commitment* as one way by which the processors at a lower level in a vision system may be constrained from introducing too much of a filtering. This principle suggests decomposing the problem in such a manner that at any level commitments are made in a conservative fashion, i.e., to the least abstract entity that is necessary. If this principle is applied at each level of abstraction carefully, the processors at the higher levels of abstraction will have available to them generally reliable information from the lower level processors. The penalty this extracts is a certain profligacy with respect to the number of processors, since typically this principle would lead to an increase in the number of levels of abstraction. Even with this principle, processors at a lower level of abstraction may still be forced to make some commitments which are not quite correct in specific instances, though on the average it may be a reasonable thing to do. This sort of thing explains certain kinds of visual illusions and visual effects, e.g., the "sun" effect in [4]. That is, visual illusions show dramatically the commitments made by lower level processors that happen not to be warranted for special classes of situations.

Another strategy is a sort of *local relaxation* by which the results of contiguous processors dealing with data in a neighborhood are compared for consistency with each other, and a processor's result would be ignored by a higher level processor if it is substantially deviant from those of its neighbors. This, of course, is a double-edged strategy, since in the occasional instances in which the deviant processor is correct, its result nevertheless does not get passed up to the higher level processors.

C. From Committees to Hierarchies

The fear of this filtering and consequent bias is at the heart of full participatory democracies like the city-state democracy of ancient Athens, where all the citizens voted on almost all the issues. However, as the size and number of issues grow large, this form of information processing begins to place great burdens on the available bandwidth, and the democratic processes become more organized, and abstractions in the form of *representation of constituencies* begin to come about. These in turn produce the problems of filtering mentioned above. They also begin to manifest another consequence of evolutionary distributed systems: a tendency to swamp out changes which are local in space or time. This is the other side of the coin of robustness that natural distributed systems often display. The changes in the environment or input have to be sufficiently large to overcome the filtering and the abstractions made by the various processors at levels of abstractions close to the sensory data.

The contribution of Wesson *et al.* for this special issue considers an experimental comparison of two distributed architectures for a message puzzle task, where a network of human sensors, each of whom sees only a small portion of a two-dimensional environment, attempts to interpret it. One arrangement was an "anarchic committee" architec-

ture—somewhat like the city-state democracy—where all the nodes were free to communicate with each other. The second arrangement was a hierarchical one. For the particular collection of experiments that were conducted, the committee architecture worked better than the hierarchies. Several aspects of the experiment are worth noting. The size of the environment as well as the sensor network was small. It would be interesting to see if the result would hold for larger size environments and large networks of nodes. If the observations in the preceding paragraph are correct, then one would expect a gradual shift in favor of a more hierarchical arrangement with increasing size. Secondly, hierarchies may be appropriate only when the environment has a sufficient amount of structure. For example, when different regions of the environment correspond to different identifiable configurations, groups of sensors will need to exchange information only within each group to identify the local configuration. The bias effect that we discussed earlier would be most pronounced when the architecture is not matched to the structure of the environment.

III. Interprocessor Communication

It seems reasonable to suppose that different architectures of distribution would emerge depending upon the costs of communication among processors or, equivalently, upon available bandwidth. When a multiplicity of media with differing bandwidths and accessibilities is available, the architecture of a naturally occurring evolutionary distributed system would be organized so as to use the available bandwidth most effectively. A modern corporation with telephones, radio, and other media available has a different architecture than one in times or regions with more primitive communication structures. Increased bandwidth availability would seem to decrease processor autonomy, i.e., place greater constraints on the amount of filtering allowed at the local processor level. This, however, would typically be counteracted by the increased burdens on the top-level processors that an overload of information will pose. So the degree of autonomy is a balance between these contending tendencies in distributed natural systems.

The communication media available can be categorized into two broad classes: one which is used by senders and receivers who know the identity of each other, and another which has more of a *broadcast* character, and is more associative in nature, i.e., the receiver uses whatever information in the medium that it deems appropriate to its needs. In large distributed systems, it is not practical, even if the bandwidth were available, for the first class of communication media to be used without constraint. For one thing, as the size of the system grows, the directory size for each processor would grow rapidly, burdening the information processing capabilities of the processor. For another, as processors are deleted or added in response to local changes, the directories will have to be updated all over the system. This collection of constraints typically leads to this class of media being used within a narrow and local scope. Most often the communication is hierarchical, thus eliminating directory size and updating problems. Information meant for or needed from other processors is directed to the broadcast media, such as *blackboards* or *journals*. However, once again, this cannot also be done with abandon, since these media will then be clogged with the outputs of the large number of processors, making it useless for receivers, unless they are willing to invest a large portion of their limited processing resources to sort out the clutter. Extremely careful and powerful abstractions, closely matching the structure of the class of tasks for which the distributed system is designed, will need to be generated as appropriate inputs to the broadcast media. Understanding how this is done is a central theoretical enterprise. We shall later comment upon the use of blackboards by three papers in this issue: those by Gomez and Chandrasekaran, Rieger and Small, and Cullingford.

IV. DPS in Human Communities

As we mentioned earlier decomposition is the basic weapon against complexity. Thus when a socially important task is too complex for individual humans, organizations with a number of humans evolve whose architecture matches the structure of the task, and whose total computational capacity is adequate for it. Task decomposition in human organizations often provides a great deal of parallelism, which is conceptually and operationally important for increased efficiency. Several hundred years ago, each competent physician possessed almost all of the medical knowledge then available. Today the complexity of medical knowledge has resulted in the creation of a complex organization of specialists, where no one knows more than a small part, but the community overall advances medical knowledge and provides care. The modern corporation is often a large distributed system with a large number of specialized subdivisions which, when successful, mesh together in a miracle of purposefulness, but when the overall structure strays too far from the changing environment, it resembles a maladaptive dinosaur (see comments on adaptation and distribution in Section II). The scientific community is another human organization whose architecture has evolved in a distributed fashion, the shaping forces in this case being the dictates of the scientific method, and the communication requirements for the creation and verification of new scientific knowledge. There are countless other examples of human organizations.

Gomez and Chandrasekaran's work in this issue concentrates on the epistemological structure of medical diagnosis, which is independent of whether the task is accomplished by a single human, a community of specialists, or a collection of microprocessors. They relate the identity and structure of specialists to the *conceptual content* of the domain. The distributed problem solving that they propose has a great deal of parallelism in it.

In their explication of the scientific community metaphor, Kornfeld and Hewitt emphasize its inherent parallelism. They develop some concurrent language primitives to emulate some of the problem solving behavior of scien-

tific communities. Fox proposes that, among the criteria for distribution in human organizations are *complexity*, *uncertainty*, and *resource constraints*. He considers the organization of a distributed system such as Hearsay-II [5], and studies the extent to which it already incorporates the insights of organization theory.

Markets are an interesting kind of distributed system. They are not primarily information processing systems, but they use a distributed, mostly local information exchange to achieve a certain kind of global optimality in resource utilization. They are the sources of the metaphors of *prices* (a kind of abstraction) and *contract and bids* (a kind of mechanism) that enable a global optimality to be reached over a period of time. The paper by Smith and Davis in this issue deals with some aspects of these metaphors and their use in distributed systems.

V. DPS in Cognitive Activity

It is easy to conceive of distributed computing in the case of what are evidently communities of individual processors, such as ants building hills, armies, corporations, or the scientific community. What is less obvious is the utility of this conception in understanding the information processing of an individual human being. The metaphor of a society of little minds—the homunculi—has come up repeatedly in psychology and philosophy of mind. Dennett [6] gives a brief but useful account of the history of this metaphor. These models have floundered on the apparent infinite regress involved in explaining a mind by postulating a collection of minds. It has been only recently that, due to work in artificial intelligence, we can begin to see how "mind-like" is not an all-or-none affair, that more complex mind-like behavior can be obtained by the coordination of less complex mind-like entities. The less complex entities are specialists, i.e., they have a narrower scope. This kind of decomposition can be applied recursively at quite a few levels: Minsky [7] has recently formulated a "society of minds" model dealing with epistemological issues all the way down at the "neurodevelopmental" level, and Marr's work is very suggestive of how vision can be conceptualized as a society of specialists: groups of specialists all the way from very low-level ones (edge specialists) through those at slowly increasing levels of abstraction to high-level conceptual specialists.

When top-level control in a society of specialists is weak or nonexistent, the subordinate specialists may speak up in different, possibly conflicting, voices. "Multiple control" substitutes unitary control. Jaynes [8] recalls the *Iliad* and its heroes' ascription of many of their actions to the demands of gods whose voices they hear. He relates this to the evidence that many schizophrenics—examples of non-unitary consciousnesses—report hearing "inner voices" during acute attacks. Hilgard [9] discusses hypnosis as a breach of this unitary control. Freud's theories, of course, were based on the view of the mind as an interacting society of many agents. In this context it is tempting to speculate that one of the roles of consciousness is in providing a blackboard (we earlier discussed it as one form of communication medium), for the agents at the top levels of cognition.

For our purposes, the foregoing speculations can be given a concrete cast by asking: What scientific value does the distributed processing metaphor—or the society of minds notion—have in explicating high-level cognitive phenomena? Gomez and Chandrasekaran in their paper in this issue point out that there is really not much that is new in the notion of a society of specialists as a model of complex computations: almost all large programs are modular and the modules are specialists. Thus they suggest that for this metaphor to be technically useful criteria are needed for *decomposing* tasks into specialists. Their paper provides such criteria for one well-defined class of cognitive activity, viz., diagnosis. The specialists are conceptual specialists who are hierarchically organized. In addition to the hierarchical communication, they also use a *blackboard*, i.e., a broadcast form of communication. The authors provide an explicit account of the structure of the blackboard for this particular task.

Similarly, Rieger and Small propose a particular criterion to organize the specialists in parsing natural language utterances. They propose that the specialists be "word experts," and suggest how their activities should be coordinated. Again it is instructive that the experts use a broadcast communication medium, viz. the "control workspace." Cullingford, in his paper in this issue, considers the problem of integrating and controlling the experts in a system to "understand" a class of newspaper stories. Again the blackboard notion finds expression in his work. For the particular class of tasks considered by him, he proposes that the blackboard data contain not only results computed by specialists, but in addition some control information, i.e., some indication of *how* the blackboard information is changed, e.g., *who* changed an item. In addition to the papers in this issue, I should like to draw the readers' attention to the work of Sacerdoti [10] who has considered how a distributed architecture might be designed for natural language understanding.

VI. Concluding Remarks

Hearsay-II [5] was one of the first large artificial intelligence (AI) systems to use an essentially distributed architecture for a complex problem and was the first system to use a blackboard as an interprocessor data structure. The paper by Lesser and Corkill in this issue attempts to concretize some of the lessons from Hearsay-II and other knowledge-base systems for the design of distributed processing systems. They concentrate on how uncertainty in input data and inaccurate processing by individual specialists can be compensated for by the collective so that the whole is more robust than the individual processors. They call their approach functionally accurate (referring to robustness of performance), cooperative (indicating some form of relaxation procedure—see Section II-C— by which

each contributes to the reduction of uncertainity of other processors) distributed systems.

Distributed problem solving, or more generally distributed artificial intelligence, is important conceptually, strategically, and practically. Its conceptual importance lies in that a distributed paradigm eludicates the structure of the processes of intelligence, such as vision, speech processing, or language understanding, whether or not an AI system is in fact implemented in a distributed manner. Its strategic importance arises from the fact that it is a good research strategy to look for decompositions of a complex problem. Its practical significance arises both from applications that are essentially distributed in nature, such as distributed sensor networks [11], as well as from the technological breakthroughs in microprocessors that make a distributed implementation of AI systems an elegantly practical possibility. I hope that the papers in this special issue direct attention to this important scientific and engineering enterprise.

Acknowledgment

I thank Andy Sage for a very helpful and cooperative role in his capacity as Editor of this Transactions, the authors of all submitted papers for their cooperation, and the reviewers R. Bhaskar, R. Cullingford, R. Davis, M. Fox, P. Kugel, J. Reggia, C. Rieger, R. Smith, N. S. Sridharan, K. Stevens, L. Svobodova, S. Tanimoto, and S. W. Zucker—for their invaluable help. R. Bhaskar, Bruce Flinchbaugh, and Fernando Gomez made useful comments on earlier drafts of this introduction. I am grateful to David Zeltzer for information on biological movement control.

References

[1] N. Geschwind, "Neurological knowledge and complex behaviors," *Cognitive Sci.*, vol. 4, no. 2, pp. 185–193, 1980.

[2] N. Bernstein, *The Coordination and Regulation of Movement*. New York: Pergamon, 1967.

[3] K. Pearson, "The control of walking," *Scientific American*, vol. 235 (6), pp. 72–86, Dec. 1976.

[4] D. Marr, "Early processing of visual information," *Phil. Trans. Roy. Soc.*, vol. 275, ser. B 942, pp. 489–519, 1976.

[5] L. D. Erman, and V. R. Lesser, "The Hearsay-II system : A tutorial," in *Trends in Speech Recognition*, W. A. Lea, Ed., Englewood Cliffs, N. J., Prentice-Hall, 1979.

[6] D. C. Dennett, "Artificial intelligence as philosophy and as psychology," in *Brainstorms*. Bradford Books, pp. 109–126, 1978.

[7] M. Minksy, "K-Lines: A theory of memory," *Cognitive Sci.* vol. 4, no. 2, pp. 117–133, 1980.

[8] J. Jaynes, *The Origin of Consciousness in the Breakdown of the Bicameral Mind*. Boston: Houghton Mifflin, 1976.

[9] E. R. Hilgard, *Divided Consciousness: Multiple Controls in Human Thought and Action*. New York: Wiley, 1977.

[10] E. Sacerdoti, "What language understanding research suggests about distributed artificial intelligence," in *Proc. Distributed Sensor Nets Workshop*, Carnegie-Mellon University, pp. 8–11, 1978.

[11] *Proc. Distributed Sensor Nets Workshop*, Carnegie-Mellon University, 1978.

2.3. Automated Translation of Computer Programs

As mentioned earlier, the major thrust here was on some basic theoretical results concerning the computability and complexity of translator generation. The vehicle for this research was the Ph. D dissertation of Doyt Perry. In the following section we present the summary section of this dissertation.

COMPUTABILITY AND COMPLEXITY ISSUES OF TRANSLATOR GENERATION

The Ohio State University, 1982

Doyt Lee Perry, Ph.D.

Overview of Results

In this work we have extended the theory of translation by carefully identifying and investigating the problem of automatic generation of translators. The immediate context for this work was provided by Buttelmann [But74], Pyster [Pys75], and Krishnaswamy [Kri76], who identified formalisms for languages, translation, and translator generation. Continuing in the same theoretical spirit, we have generalized and extended several of their ideas in the course of our work. The primary conclusion of this thesis is that searching for general methods of automatically generating translators is likely to fail. In pursuing this result we believe several contributions were made to translation theory.

We were able to provide a variation of the language definition schemes of Buttelmann, Pyster, and Krishnaswamy that makes more precise the means by which the semantics of a language definition are specified. We demonstrated that one type of language definition system, namely an acceptable LDS, is representative of all language definition systems in the sense that results developed for an acceptable LDS will carry over to other language definition systems. This permitted us to focus our computability studies on one system and be confident our results were generally applicable.

We were able to formalize translation and suggested two basic problems for study - the translator generation problem and the translator generation decision problem. We noted that solutions to these are related in that generating a translator is one way of confirming a translator exists. This permitted us to look primarily at the problem of deciding the existence of translations with the assurance our results were relevant to generating translators.

In the area of computability, we were able to establish, using the vehicle of translator generation decision sets, a framework for assembling the known results about the complexity of translator generation. We added several results that more precisely characterized the computability of translator generation. Significant among these was the placing of several translator

generation problems in the arithmetic hierarchy. This permits those problems to be compared to other known unsolvable problems.

Given the impossibility of general solutions, we examined the effects of restricting the classes of language definitions for which we desire to decide translation. When we found subclasses of language definitions having certain desirable properties, we discovered that we were unable to decide translator generation for these subclasses. We then looked at several specific restrictions based on ideas from formal languages and computability theory. When most combinations of these restrictions were enforced, they were found to yield classes of language definitions which still had undecidable translator generation problems. It was demonstrated that noncomputable problems arose for both syntactic and semantic reasons. However, a characterization was made of some language definition classes for which we could solve the translator generation and TG decision problems. Included was one class that used extremely simple operations in defining the semantics of a language.

For the classes of language definitions found to have solvable translator generation problems, we analyzed the complexity of their solutions. We found an inherent "hardness" about those problems that implied that any solution would need to use an inordinate amount of resources (such as time) when applied to infinitely many instances of the problems. Although we normally associate high complexity with semantic processing, it was found that even the language definition systems using very simple semantics were seen to have provably intractable solutions. A postscript on these complexity results noted that "bad complexity" instances may arise from the particular choice of definitions for languages.

Finally, we gave a formal sketch of an oracle-based translation scheme that uses an outside source of knowledge to assist in performing translation. We focused on a translator generation scheme suggested by Buttelmann [But74] that used a particular oracle and a particular method of translation. In that case we discovered an oracle-based method of "partially solving" the translator generation problem. However, we found that there is no bound on the work done by such a procedure, nor on how many consultations it would request of the oracle.

Assessing the Results

We believe there are two primary contributions made by this work. First, a body of computability and complexity results for translator generation have been developed and organized. Secondly, the work implies that any attempt to find formal, procedural solutions to automatic translator generation will likely run afoul of computability or complexity difficulties. The results presented here are undeniably negative. It was not the intent nor the expectation of this work to acquire such a collection of pessimistic comments on translator generation. At each step we were surprised and fascinated by the levels of noncomputability and intractability of the translator generation problems. If anything, these results may be theoretical evidence for what we know from experience in programming and natural language translation - discovering and implementing translations can be difficult.

The Future

The results in this thesis suggest a search for algorithmic translator generation is destined to be difficult. In the face of this, we suggest some courses for future research in translator generation.

Heuristic Approaches

One alternative is to abandon the search for formal algorithmic solutions that guarantee semantic-preserving translations or total translations. This might involve the discovery of translation heuristics that permit procedures to cut through the exhaustive searching that often leads to the high complexity of a problem. The price paid for using such heuristics might be that all sentences of the source language might not have a translation. Perhaps not all translations will be semantic preserving. Such approximate translation might be acceptable in many cases, especially if the translator generation process using heuristics posessed reasonable complexity.

Translation semantics

We have been faithful to an abstract view of language definition and semantics (we have not looked at semantics for programming languages or for natural languages). Perhaps one could develop a "translation semantics" specially developed with translation in mind. If a class of languages were defined in terms

of these semantics, perhaps the generation of translators for this class could be done efficiently. Such an idea is consistent with some work by Krishnaswamy [Kri76], where he used "identical semantics" as a means of doing translator generation.

Translation between Similar Languages

Often the translation we want to do is not between radically different languages but rather between "dialects" of the same language. This is especially true of the programming languages area, where we often convert programs written in one version of a programming language to equivalent programs in a different *version* of the same language. Similarity between languages, their syntax and semantics might reduce the work *needed* to perform translator generation.

Choosing Appropriate Language Definitions

As a final suggestion, some of our results suggest the sensitivity of translator generation to the particular choice of language definitions selected to describe source and target languages. This raises the possibility that translator generation might be possible or tractable if only we could select the "right" language definitions. One aspect of this is the balance between syntax and semantics in a definition. Although this thesis makes some suggestions, it is important to study further the effect the selection of language *definitions* has on the computability and complexity of translator generation.

Finally, note that this *thesis* has focused on formal translations and on general procedures for generation of translators. Pitched at *this abstract* level, it has no direct application to *practical* problems of translation methodologies. For example, no attempt has been made to study the particular translator generation problems for programming languages. In the spectrum that ranges from the theoretical to the practical, our experience over many years has led us to conclude that finding translators is a difficult practical matter. We believe that this thesis has echoed, on the theoretical end, that translator generation is difficult.

LIST OF REFERENCES

But74 BUTTELMANN, H.W. Semantic-Directed Translation, American Journal of Computational Linguistics 2, (1974), Microfiche 7.

Kri76 KRISHNASWAMY, RAMACHANDRAN Methodology and Generation of Language Translators, Ph.d Dissertation, The Ohio State University, 1976.

Pys75 PYSTER, ARTHUR Formal Translation of Phrase Structure Languages, Ph.d. Dissertation, The Ohio State University, 1975.

3. PUBLICATIONS AND OTHER ACTIVITY

3.1. List of Publications

The following is a list of publications that resulted from research supported by this grant. Asterisk (*) indicates that the paper or report in question appears as a section in this final report.

i. Lee J. White, "Basic mathematical definitions and results in testing," in Computer Program Testing, B. Chandrasekaran and S. Radicchi, Ed., North Holland, 1981, pp. 13 - 24.

ii. Lee J. White, Edward I. Cohen, and Steven J. Zeil, "A domain strategy for computer program testing," in Computer Program Testing, B. Chandrasekaran and S. Radicchi, Ed., North Holland, 1981, pp. 103 - 113. (*)

iii. Steve J. Zeil and Lee J. White, "Sufficient Test Sets for Path Analysis Testing Strategies," Proc. 5th Intern. Conf. Software Engineering, San Diego, Calif., March 9 -12, 1981, pp. 184 - 191.

iv. Lee J. White, "Some Research Approaches Motivated by the Domain Testing Strategy," Digest of the Workshop on Effectiveness of Testing and Proving Techniques, Avalon, Calif., May 1982.

v. Allen Haley and Stuart Zweben, "Module Integration Testing," in Computer Program Testing, B. Chandrasekaran and S. Radicchi, Ed., North Holland, 1981, pp. 289 - 299.

vi. Allen Haley and Stuart Zweben, "Development and application of a white box approach to integration testing," submitted to Journal of Systems and Software. Also appeared in Digest of the Workshop on Effectiveness of Testing and Proving Techniques, Avalon, Calif., May 1982.

vii. Fernando Gomez, "Towards a theory of comprehension of declarative contexts," Proc. of 20th Ann. Meeting of the Association for Computational Linguistics, 16- 18 June, 1982, Ontario, Can., pp. 36 -43.

viii. B. Chandrasekaran, "Guest Editorial: Special Collection on Program Testing," IEEE Trans. Software Engineering, May 1980, pp. 233 - 235. (*)

ix. B. Chandrasekaran, "Natural and Social System Metaphors for Distributed Problem Solving: Introduction to the Issue," IEEE Trans. System, Man & Cyb., January 1981, pp. 1 - 5.

x. B. Chandrasekaran and S. Radicchi, editors, Computer Program Testing, North Holland Publications, 1981.

3.2. Dissertations supported by the Grant

i. Fernando Gomez, "On General and Expert Knowledge Based Methods in Problem-Solving," The Ohio State University, 1981.

ii. Steven Zeil, "Selecting Sufficient Sets of Test Paths for Program Testing," The Ohio State University, 1981.

iii. Doyt Perry, "Computability and Complexity Issues of Translator Generation," The Ohio State University, 1981.

iv. Allen Haley, dissertation in progress on integration testing.