Theoretical and Empirical Studies of Program Testing

WILLIAM E. HOWDEN

Abstract—Two approaches to the study of program testing are described. One approach is theoretical and the other empirical. In the theoretical approach situations are characterized in which it is possible to use testing to formally prove the correctness of programs or the correctness of properties of programs. In the empirical approach statistics are collected which record the frequency with which different testing strategies reveal the errors in a collection of programs. A summary of the results of two research projects which investigated these approaches are presented. The differences between the two approaches are discussed and their relative advantages and disadvantages are compared.

Index Terms—Algebraic, graph theory, program equivalence, proofs of correctness, reliability, statistics, testing, traces.

I. Introduction

Reliability of Program Testing

A COMMON problem in program testing is the lack of a well-defined, reliable method for selecting tests. This paper discusses a theoretical and an empirical approach to the problem.

In order to use testing to validate a program it is necessary to assume the existence of a *test oracle* which can be used to check the correctness of test output. The most common kind of test oracle is one which can be used to check the correctness of output values for a given set of input values. Other kinds of test oracles can be used to check the correctness of value traces of selected program variables. Formally defined oracles may consist of tables of values, algorithms for hand computation, or formulas in the predicate calculus. Informally defined oracles are often simply the assumed ability of the programmer to recognize correct output.

Theoretical and Empirical Approaches

The theoretical approach involves the characterization of situations in which testing can be used to prove the correctness of programs or the correctness of selected computational substructures. In the approach described below, a program P is proved correct by proving that it is equivalent to a hypothetical correct program P*. Computational substructures of P are proved correct by proving that they are equivalent to corresponding substructures in P*. A test oracle for a program P can be thought of as a source of information about P*.

The empirical approach involves the compilation of statistics on the relative frequency with which selected testing strategies are reliable in discovering the errors in a program. The use of the approach requires the availability of a collection of incorrect programs whose bugs are known in advance. The approach is more convincing if it is applied to programs with naturally occurring errors rather than seeded errors.

II. Theoretical Approach

Mathematical Basis

If a computer program has a small finite domain, and a test oracle is available which can be used to check the correctness of output values, then the use of testing to prove correctness is straightforward. When the domain is large it is necessary to rely on mathematical theory. The following examples illustrate the use of two kinds of mathematical theories.

Graph-Theoretic Methods

Symbolic traces are generated by traversing paths through the source code of a program. The simplest kind of symbolic trace consists of the source statements which occur along a path through the program. It is often desirable to omit certain kinds of statements from a symbolic trace or to print information derived from a statement rather than the statement itself. Fig. 2 contains a symbolic trace of a path through the program fragment in Fig. 1. go to's, begin's, and end's are omitted from the trace. Each time a branch from a conditional statement is followed, the branch predicate is included in the trace. The trace element "until il found" is used to trace the loop exit branch from the repeat until loop.

The program fragment in Fig. 1 is derived from the polynomial interpolation routine INTERP in [1]. Input to INTERP includes two vectors x and y which contain a set of npts pairs of values (xi,yi). The pairs are part of a function f for which f(xi) = yi. The routine uses nterms of the pairs to define a polynomial which is used to compute the value of yout at the point xin. The program fragment finds the set of nterms pairs which "straddles" xin as evenly as possible. The nterms pairs are selected by computing values for i1 and i2. The set of pairs that is used is defined to be

$$\{(x(i1),y(i1)), (x(i1+1),y(i1+1)), \ldots, (x(i2),y(i2))\}.$$

nterms is equal to i2 - i1 + 1.

Symbolic traces like that in Fig. 2 are often useful in analyz-

Manuscript received October 25, 1977; revised February 16, 1978. The research on the theoretical approach was funded by the National Science Foundation. The empirical research was funded by the National Bureau of Standards under the supervision of Dr. D. Fife.

The author is with the Department of Mathematics, University of Victoria, Victoria, B.C., Canada.

```
        :
        :
1 ← 1 ;
ilfound ← false; youtfound ← false;
repeat until ilfound;
    if  xin = x(i)  then begin yout ← y(i);
                               ilfound ← true;
                               youtfound ← true;
                          end;
        else if  xin < x(i)  then begin il ← i-nterms/2;
                                        if  il ≤ 0  then il ← 1;
                                        ilfound ← true;
                                  end;
            else if  i = npts  then begin  il ← npts -nterms+1;
                                           ilfound ← true;
                                      end;
                  else  i ← i+1;
end;
if not  youtfound  then
        begin  i2 ← il + nterms - 1;
            if npts < i2  then
                begin  i2 ← npts;
                       il ← i2 - nterms+1;
                       if il ≤ 0 then begin il ← 1;
                                            nterms ← i2 - il+1;
                                      end;
                end;
        end;
else return;
        :
        :
```

Fig. 1. Fragment of INTERP program.

```
i ← 1;
ilfound ← false; youtfound ← false;
repeat until ilfound;
    xin ≠ x(i);
    xin < x(i);
        il ← 1 - nterms/2;
        il > 0;
        ilfound ← true;
until ilfound;
not youtfound;
    i2 ← il + nterms - 1;
    i2 ≤ npts;
```

Fig. 2. Elementary trace of INTERP.

ing the different sequences of operations that can be carried out by a program. The following theorem proves that it is possible to deduce the correctness of the complete set of symbolic traces of a program from the correctness of a well-defined finite subset. A program is correct if all of its symbolic traces are correct.

Definition: Let P be a program and let T be the set of all symbolic traces of P. Let E be the set of all traces in T which are generated by paths through P that traverse loops zero or one times. E is the set of *elementary traces* of P.

A path traverses a loop zero times if it follows the loop exit branch from the loop without traversing the code inside the loop. Repeat-until loops are always traversed at least once because the loop exit branch occurs at the end of the loop.

Theorem 1: Suppose that P and P* are two programs with sets of symbolic traces T and T* and sets of elementary traces E and E*. If $E \subset E^*$ then $T = T^*$.

The proof of Theorem 1 is contained in [2]. The proof is graph-theoretic in the sense that it involves proving that the paths through two labeled graphlike structures are identical. The proof of the theorem depends on the type of information recorded in the traces and on certain properties of the programming language in which the traced programs are written. It is assumed that the information recorded in the traces is like that in Fig. 2 and that the traced programs are written in a structured Algol-like language in which there are restrictions on the use of go to's.

```
subroutine  pdiv(idimx,x,idimy,y,idimw,w);
dimension   x(idimx), y(idimy), w(idimw);
idimx ← idimx - idimy + 1;
idimx ← idimy - 1;
repeat for  i ← idimw  by  -1 to 1
  il ← i + idimx;
  w(i) ← x(il)/y(idimy);
  repeat for  k ← 1  by  1  to idimy
      j ← k - 1 + i;
      x(j) ← x(j) - w(i) + y(k);
  end;
end;
```

Fig. 3. PDIV program.

```
idimx = 3, idimy = 2
i = 2
    w(2) ← x(3)/y(2)
    k = 1
        x(2) ← x(2) - w(2)*y(1)
    k = 2
        x(3) ← x(3) - w(2)*y(2)
i = 1
    w(1) ← x(2)/y(2)
    k = 1
        x(1) ← x(1) - w(1)*y(1)
    k = 2
        x(2) ← x(2) - w(1)*y(2)
```

Fig. 4. Value trace of PDIV.

Algebraic Methods

A *value trace* is a trace of the values which are assigned to a selected set of variables during an execution of a program. Value traces usually contain both values and other information which can be used to identify the values. Fig. 4 contains a value trace of the program in Fig. 3. The trace contains a record of the values used to index the arrays in the program. In addition to the array indices, the trace also contains a record of the values of the variables used to compute the array indices. The array indexing values are identified in the trace by reproducing the statements in which they occur.

The PDIV program in Fig. 3 is from the IBM scientific subroutine package. The program can be used to carry out polynomial division. The array variables x and y, of length idimx and idimy, are used to store the coefficients of the polynomials. The coefficients of the quotient are returned in the variable w of dimension idimw. The coefficients of the remainder are returned in array x. It is assumed that idimx $\geqslant$ idimy > 0. The division process is carried out by subtracting successive multiples of the divisor from the dividend.

Value traces like that in Fig. 4 can be used to check the validity of simple algebraic computations. The following theorem proves that it is possible to deduce the correctness of simple computations from the correctness of a finite set of value traces.

Theorem 2: Suppose that f and g are linear functions of the variables $x_1, x_2, \cdots, x_{s-1}$. Let $x_{i,j}$, $1 \leqslant i \leqslant s, 1 \leqslant j \leqslant s-1$ be s sets of values for the variables x_j, $1 \leqslant j \leqslant s-1$, and let T be the matrix in Fig. 5. Suppose that f = g over each test T_i, $1 \leqslant i \leqslant s$, where

$$T = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,s-1} & 1 \\ x_{2,1} & x_{2,2} & \cdots & x_{2,s-1} & 1 \\ \vdots & & & & \\ x_{2,1} & x_{s,2} & \cdots & x_{s,s-1} & 1 \end{pmatrix}$$

Fig. 5. Test values matrix.

$T_i = (x_{i,1}, x_{i,2}, \cdots, x_{i,s-1})$.

Then, if T is nonsingular, f and g are identical polynomials.

Theorem 2 is a special case of a more general theorem which is described in a more detailed discussion of the theoretical approach [2]. The more general theorem is not limited to linear functions. Additional general theorems, similar to Theorem 2, are proved in [3].

In order to use Theorem 2 to prove the correctness of a linear computation f in a program P it is necessary to assume that P has certain properties in common with the correct program P*. In particular, it is necessary to assume that there is a "corresponding" linear function f* in P* which is defined over the same domain as f.

Two assumptions are sufficient to allow the application of Theorem 2 to the PDIV example. The first is that the array indexing computations in PDIV*, like those in PDIV, can be written as linear functions of the input variables idimx and idimy and of the loop indices of the loops containing the array-indexing computations. The second assumption is that PDIV and PDIV* are identical except for possibly the array indexing computations. This second assumption makes it possible to match each array index in PDIV with a corresponding array index in PDIV*.

The trace in Fig. 4 contains examples of input-output values for each of the array-indexing computations in PDIV. A value of 3, for example, is generated for the index in the array reference x(j) in the inner loop of PDIV when the values of idimx, idimy, i, and k are 3,2,2, and 2. In order to apply Theorem 2 to the indexing computations in the inner loop of PDIV, it is necessary to generate five examples of input-output pairs. The required pairs can be generated by testing PDIV for (idimx, idimy) = (1,1), (3,2), and (2,1). The test values matrix T which can be constructed from those tests is nonsingular. The same set of tests also generates sufficient input-output pairs for the verification of the array indices in the outer loop.

III. Empirical Approach

Program Testing and Program Analysis Techniques

A program testing or analysis technique is *reliable* for an error in a program if its use is guaranteed to result in the discovery of the error. A study was carried out in which the reliability of the following techniques was analyzed.

Testing Techniques

Path Testing [4]: This technique requires that each executable path through a program be tested at least once. The technique is not practical since a program may have an infinite number of paths. The reliability of the technique is an upper bound on the reliability of techniques that require the testing of a special subset of the set of all paths.

Branch Testing [5]-[7]: This technique requires that each branch be tested at least once.

Structured Testing [8], [9]: In order to use this technique it is necessary to assume that the program consists of a hierarchial structure of small functional modules. Each path through a functional module which executes loops less than k times is tested at least once. Usually k = 2.

Special Values Testing [10]: Experience indicates that it is important to test programs on certain kinds of special values. The "distinct-values" rule, for example, requires that different elementary items in an input data structure be assigned different values. The "zero-values" rule requires that tests be carried out in which zero values are assigned to or computed for variables that appear in arithmetic expressions. The "input-classes" rule is useful for programs for which the input values divide into several different classes of data. A program will typically carry out different kinds of processing for each class. The rule requires that the program be tested for each class or combination of classes of input data.

Symbolic Testing [11]-[17]: The techniques just described are usually associated with the testing of programs over "actual data." It is also possible to apply the techniques using symbolic data. "Symbolic structured testing" is like structured testing, except that paths are executed over symbolic data rather than actual data.

Program Analysis Techniques

A number of program analysis techniques which do not involve the execution of a program can be used to examine a program for errors.

Interface Consistency [18]: This term is used to refer to rules which require an analysis of the interface between routines or between routines and an external data set. An example is a rule for checking the consistency of the format of structured input records with the format of the variables used to read in the records.

Anomaly Analysis [19]-[21]: Anomaly analysis involves the examination of a program for suspicious looking constructs. The use of an uninitialized variable is one example of an anomaly. Another is the use of a fixed-point (PL-1) variable in a situation where it can assume arbitrarily large values.

Specification Requirements: Some errors can be avoided if certain restrictions are imposed on program specifications. It may be required, for example, that the specifications contain a description of the type and range of each variable.

Effectiveness of Testing Strategies

The reliability of testing and program analysis methods can be measured by applying them to programs containing known errors. The methods described above were applied to a collection of six programs containing 28 errors. Two of the programs were written in Algol, and one each in Cobol, PL1, Fortran, and PL360. Descriptions of the programs and the types of errors that were analyzed can be found in [10].

The table in Fig. 6 describes the results of the analysis. Two of the references [10], [22] contain a more detailed account

Path	18	Interface	2
Branch	6	Anomaly	4
Structured	12	Specifications	7
Special Values	17	Combined Non-Symbolic	25
Symbolic Structured	17		

Fig. 6. Reliability statistics.

```
word:=empty string;
    :
repeat
    if input = emptystring then input:=read+' ';
    letter:=first(input); input:=rest(input);
    word:=word+letter;
until  letter = ' ';
```

Fig. 7. Fragment of Algol string processing program.

of the results as well as results of the analysis of additional methods. The results in Fig. 6 indicate that path testing is reliable for 18 of the 28 errors. Branch testing was reliable for only 6 of the 28 errors. This low figure indicates that many of the errors that are reliably discovered by path testing depend on the traversal of combinations of branches rather than single branches. Structured testing is reliable for 12 of the 28 errors. Structured testing increases the reliability of branch testing by forcing the testing of important combinations of branches. Structured testing reveals all of the errors in three of the six programs. Branch testing is completely reliable for only one program. The combined use of the structured, special values, interface, anomaly, and specifications techniques is reliable for 25 of the 28 errors.

Structured testing was unreliable for a significant number of the errors in one of the larger programs, a PL-1 program called TEST. The most reliable method for TEST was the special values method. The reliability experiments with TEST indicate the importance of program testing methods which, unlike structured testing, are not defined in terms of the control structure of a program.

On the sample programs, symbolic structured testing was found to be 10-20 percent more reliable than structured testing over actual data. The five errors that are reliably discovered by symbolic structured testing but not structured testing over actual data all involve the use of an incorrect variable. In all five errors it is possible for the incorrect variable to take on the values of the correct variable during testing on actual data, thus hiding the presence of the error. If the normal practice of using variable names for symbolic values is followed, then the use of the incorrect variables is clearly revealed by symbolic testing.

Symbolic testing revealed one error which was not revealed by any of the other techniques. The combined use of all of the listed techniques results in the guaranteed discovery of 26 of the 28 errors.

The following example illustrates a typical situation in which one technique is reliable and another unreliable.

The program fragment in Fig. 7 is taken from the telegram program described in [23]. The program incorrectly fails to delete leading blanks whenever a new buffer of text is read with the "input := read+' '" statement. This can result in the generation of a word consisting of a single blank. In order to force the error in the fragment, it is necessary to follow the true branch from the conditional statement and then exit from the loop on some first iteration of the loop. Structured testing, but not branch testing, will force the testing of this combination of circumstances.

IV. Conclusions

Theoretical Approach–Advantages and Disadvantages

The advantage of the theoretical approach is obvious. If a particular testing strategy has a sound theoretical basis, then the user of the strategy knows what has been proved and what has not been proved.

There are several disadvantages to the theoretical approach that limit its applicability. The proofs of the theorems that are used in the theoretical approach depend on the availability of a test oracle which can recognize the correctness of specific kinds of test output. There are many programs for which it is reasonable to assume the existence of a test oracle, but it may not be the same kind of test oracle as that which is required in the program testing theorems. The theorems may also require the use of certain assumptions about the correct version P* of a program P that are difficult to justify.

In order to apply Theorem 1 to a program, it is necessary to assume the existence of an oracle which can recognize the correctness of elementary symbolic traces. This is a stronger assumption than it may at first appear. Stated differently, it is necessary to assume that there is an external mechanism which can determine if the elementary symbolic traces of a program P are also symbolic traces of a correct program P*.

In order to apply theorems like Theorem 2 to an algebraic computation in a program P, it is necessary to assume that there is a "matching" computation in the correct program P*. It is also necessary to assume that the matching computation has the same input variables and that there is a known upper bound on its degree.

The theoretical approach to the study of program testing has resulted in additional insight into the program testing process. In certain isolated cases, such as the testing of simple algebraic computations or the symbolic tracing of complex control logic, theoretical results can be used to guide the selection of tests. The study has not resulted in any general, useful method for testing programs.

Empirical Approach–Advantages and Disadvantages

The advantage of the empirical approach to the study of testing is that it can be applied to the study of any testing methodology. The theoretical approach can only be applied to those methods for which it is possible to prove theorems.

The major disadvantage of the empirical approach is its lack of a sound mathematical basis. The fact that a particular strategy is reliable for discovering 35 percent of the known errors in a specific collection of programs is no guarantee that it will be that reliable on some other, previously unanalyzed program. A particular combination of different testing and program analysis methods may be the most effective for discovering the errors in one program but not in another.

Future Work

It is the author's opinion that the greatest practical benefits are to be gained from continued empirical rather than theoretical studies. Ideally, patterns will emerge from the empirical study which can be used as the basis of future theoretical work.

V. Related Work

The use of testing in proofs of correctness has been proposed and studied by several people [2], [3], [24]-[26]. Reference [4] describes a theoretical approach to the reliability of a particular testing strategy. Goodenough and Gerhart published a paper on what might be called a "theory of testing" [26]. The use of algebra in proving the equivalence of simple computations is described, but not developed, by Geller [25]. The use of symbolic evaluation in proving correctness of programs and program paths is discussed in [11], [31], [27]-[29]. The relationship between the study of formal grammars and proofs of equivalence of symbolically evaluated programs is explored in [30]. The material on symbolic traces in Section II is related to research on symbolic testing and automatic test data generation systems [10]-[17].

The theoretical results in Section II are closely related to work that has been described by the author in several technical reports [3], [24]. Reference [2] contains a more detailed description of the examples in Section II and contains the proofs of the theorems. Reference [24] contains a program equivalence theorem which involves the combined use of symbolic and value traces [6].

Few empirical studies of program testing have been reported in the literature. Reference [12] describes the results of a smaller, more limited study that used the same research methodology as the study reported in this paper. An alternative approach is described by Hetzel in his dissertation [31]. Hetzel carried out a number of statistical experiments in which different groups of programmers used three different testing and program analysis techniques to debug three programs containing known errors. Hetzel's approach is suitable for the analysis of ill-defined techniques such as "code-reading" and "specifications testing." The approach described in this paper is suitable for well-defined techniques such as structured testing and interface analysis. The reliability of these techniques can be analyzed without the need for statistical experiments.

The empirical research described in Section III is described in more detail in [22]. Complete details of the empirical studies can be found in [10].

Acknowledgment

P. Eichhorst, of the University of California at San Diego, participated in the theoretical research for this paper.

References

[1] P. R. Bennington, *Data Reduction and Error Analysis for the Physical Sciences.* New York: McGraw-Hill, 1969.
[2] W. E. Howden and P. Eichhorst, "Proving properties of programs from program traces," Univ. California, San Diego, Computer Science Tech. Rep. 18, 1977.
[3] W. E. Howden, "Elementary algebraic program testing techniques," Univ. California, San Diego, Computer Science Tech. Rep. 12, 1976.
[4] —, "Reliability of the path analysis testing strategy," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 208–214, 1976.
[5] K. W. Krause, R. W. Smith, and M. A. Goodwin, "Optimal software test planning through automated network analysis," in *Proc. 1973 IEEE Symp. Computer Software Reliability* (IEEE Computer Society, Long Beach, CA), pp. 18–22.
[6] L. G. Stucki, "Automatic generation of self-metric software," in *Proc. 1973 IEEE Symp. Computer Software Reliability*, IEEE Computer Society, Long Beach, CA, pp. 94–100.
[7] R. E. Fairley, "An experimental program testing facility," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 350–357, 1975.
[8] H. D. Mills, "Top down programming in large systems," in *Debugging Techniques in Large Systems*, R. Rustin Ed. Englewood Cliffs, NJ: Prentice-Hall, 1971, pp. 41–55.
[9] C. V. Ramamoorthy, K. H. Kim, and W. T. Chen, "Optimal placement of software monitors aiding systematic testing," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 46–58, 1975.
[10] W. E. Howden, "Symbolic testing–Design techniques, costs and effectiveness," U.S. National Bureau of Standards GCR77-89, National Technical Information Service PB268517, Springfield, VA, 1977.
[11] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT–A formal system for testing and debugging programs by symbolic execution," in *Proc. 1975 Int. Conf. Software Reliability*, IEEE Computer Society, Long Beach, CA, pp. 234–245.
[12] W. E. Howden, "Symbolic testing and the DISSECT symbolic evaluation system," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 266–278, 1977.
[13] J. C. King, "Symbolic execution and program testing," *Commun. Ass. Comput. Mach.*, vol. 19, pp. 385–394, 1976.
[14] L. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 215–222, 1976.
[15] E. F. Miller and R. A. Melton, "Automated generation of test case data sets," in *Proc. 1975 Int. Conf. Reliable Software*, IEEE Computer Society, Long Beach, CA.
[16] J. C. Huang, "An approach to program testing," *Comput. Surveys*, vol. 7, pp. 113–128, 1975.
[17] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen, "On the automated generation of program test data," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 293–300, 1976.
[18] B. C. Hodges and J. P. Ryan, "A system for automatic software evaluation," in *Proc. 2nd Int. Conf. Software Engineering*, 1976.
[19] C. V. Ramamoorthy and S. F. Ho, "Testing large software with automated software evaluation systems," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 46–58, 1975.
[20] L. J. Osterweil and L. D. Fosdick, "DAVE–A validation, error detection and documentation system for Fortran programs," *Software–Practice Experience*, vol. 6, pp. 473–486, 1976.
[21] L. D. Fosdick and L. J. Osterweil, "The detection of anomalous interprocedural data flow," in *Proc. 2nd Int. Conf. Software Engineering,* 1976.
[22] W. E. Howden, "An evaluation of the effectiveness of symbolic

testing," Univ. California, San Diego, Computer Science Tech. Rep. 16, 1977.
[23] P. Henderson and R. Snowden, "An experiment in structured programming," *BIT*, vol. 12, pp. 38-53, 1972.
[24] W. E. Howden, "Algebraic program testing," Univ. California, San Diego, Computer Science Tech. Rep. 14, 1976.
[25] M. Geller, "Test data as an aid in proving program correctness," in *Proc. 2nd Symp. Principles of Programming Languages* (1976), pp. 209-218.
[26] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 156-173, 1975.
[27] W. E. Howden, "Methodology for the generation of program test data," *IEEE Trans. Comput.*, vol. C-24, pp. 554-559, 1975.
[28] R. M. Burstall, "Proving correctness as hand simulation with a little induction," in *Proc. Int. Federation of Information Processing Societies 74.* Amsterdam, The Netherlands: North Holland, 1974, pp. 308-312.
[29] L. P. Deutsch, "An interactive program verifier," Ph.D. dissertation, Univ. California, Berkeley, 1973.
[30] W. E. Howden, "Lindenmayer grammars and symbolic testing," *Inform. Process. Lett.*, to be published.
[31] W. Hetzel, "An experimental analysis of program verification methods," Ph.D. dissertation, Univ. North Carolina, 1976.

William E. Howden, for a photograph and biography, see p. 73 of the January 1978 issue of this TRANSACTIONS.

Dynamic Restructuring in an Experimental Operating System

HANNES GOULLON, RAINER ISLE, AND KLAUS-PETER LÖHR

Abstract—**A well-structured system can easily be understood and modified. Moreover, it may lend itself even to dynamic modification: under special conditions, the possibility of changing system parts while the system is running can be provided at little additional cost. Our approach to the design of dynamically modifiable systems is based on the principle of data abstraction applied to types and modules. It allows for dynamic replacement or restructuring of a module's implementation if this does not affect its specification (or if it leads to some kind of compatible specification). The fundamental principles of such "replugging" are exhibited, and the implementation of a replugging facility for an experimental operating system on a PDP-11/40E is described.**

Index Terms—**Data abstraction, domain architecture, dynamic address spaces, dynamic modification, modules, replugging.**

I. INTRODUCTION

ONE of the fundamental facts in software engineering is that large systems, once put into operation, are usually subject to frequent modification. A system is modified for several possible reasons, e.g., error correction, efficiency improvement, changes in or amplification of functional capabilities. Thus, to produce high-quality software, such modifications must be anticipated. The notion of *modifiability* comprises structural properties of software that facilitate fast and secure modification.

While requirements for software modifiability have been studied to some extent (cf. Parnas [11], Bauer [1]), the problem of how to modify a long-running system "smoothly," i.e., without stopping it and replacing it by a new version, has attracted only little attention. Shaw has mentioned the problem from the viewpoint of data abstraction [15]. Fabry was motivated to attack the problem by considering data base systems; he uses the term "changing modules on the fly" [2].

Throughout this paper, program modification during runtime will be called *dynamic modification* (as opposed to "static modification") in order to stress the analogies to similar uses of "dynamic" versus "static," e.g., static/dynamic memory management, linking. We were led to study, in a real system, dynamic modification and the design of a dynamic modification facility by a project to construct an experimental operating system. The system is to be used in operating systems research and education at a university. It is called a DAS (dynamically alterable system), and is being implemented on a PDP-11/40E. The system is designed in such a way that dynamic modifications are possible in all

Manuscript received November 15, 1977.
The authors are with the Technische Universität Berlin, Berlin, Germany.