A COMPLEMENTARY APPROACH TO PROGRAM ANALYSIS AND TESTING

A. Endres and W. Glatthaar
IBM Corporation
Boeblingen/Germany

ABSTRACT

The paper is a contribution to the theory of program testing. We first discuss why program testing is superior in some respects to program proving. Then current strategies and criteria for test case selection are reviewed and their shortcomings identified. A methodology is proposed to facilitate the selection of a set of test cases which is adequate for functional verification. It is necessary, however, that certain complementary information be used which can be derived from the program text through formal analysis.

KEYWORDS

Software engineering, software reliability, program verification, program testing, program semantics.

INTRODUCTION

During the last ten years there has been an abundance of proposals on how to improve the software development process. One school of thought, as represented by *Dijkstra* (1969), *Mills* (1971) and *Wirth* (1971), puts the emphasis on so-called *constructive* methods. In essence, these are rules and guidelines on how one should proceed when developing software. By definition, these methods are neither suited nor intended for communicating the end results of the development process. This function, however, is indispensable in practical environments, and has to be provided by a complementary set of methods. We will refer to these methods as *analytical* methods.

The most effective analytical methods are both objective and repeatable. By *objective* we mean that the result should be as independent of the person applying the method as possible and, most important, the method should be applicable with minimal involvement of the program originator. A *repeatable* method has the advantage that it can be applied by

various persons, either concurrently or at different times and locations, thus reducing the risk of failure and increasing the credibility. This paper is concerned with program analysis and verification, two analytical methods that are relevant with respect to program validity and correctness. They fulfil the above postulates by being applicable to the program text itself, and also by the fact that they are formal in nature. A formal method has as a key property the facility of being mechanized.
The approach described in this paper is influenced by the work of *Miller/Paige* (1974), *Bauer* (1975), *Goodenough/Gerhart* (1975) and others. Easily accessible tutorials on the subject are given by *Huang* (1975) and *Miller* (1977). For most of the concepts of this paper, the corresponding formalism has been developed elsewhere and can be found in *Endres* (1977).

PROGRAM VALIDITY AND ANALYSIS

To define specifically the role of testing, we introduce first some concepts and definitions that allow us to restrict as much as possible the scope of information to be derived from a program through testing. The basic idea that enables us to do this is reflected in the distinction between program validity and program correctness.

A certain string g is a *valid* program of some language L if it is a member of the set G of all syntactically and semantically valid programs of that language. The following relations apply:

$$g \in G \quad \text{and} \quad G \subseteq L$$

It is the purpose of *program analysis* to show that these relations hold. While the term syntactically valid may not need any explanation, there is hardly any agreement in the literature on the term semantic validity. We would like to adhere to the most stringent definition possible. Stringent, in this connection, implies reducing as far as possible the number of programs that are accepted as valid.

Semantically, programs are function definitions. Hence, if a string g is a program, it specifies some mapping between a set of values, called the *input domain* X, and another set of values, called the

output range Z.

$$g: X \rightarrow Z$$

The sets X and Z are usually sets of tuples, i.e. Cartesian products of simple sets.

We say a program is *semantically valid* if both X and Z are specified, and if, for every tuple in X, the program terminates and the result is contained in Z. Mathematically speaking, a valid program always defines a total function.
This concept can be illustrated by the following brief examples (in PASCAL notation):

```
      var x,y: integer;
g1:   y:= x div 0
g2:   y:= x div (x - 5)
g3:   while x > 0 do x := 2 * x - 20
g4:   if x ≠ 5  then y:= x div (x - 5) else y:= x
g5:   if x ≥ 0  then y:= x  else y:= -x
```

In most semantic language definitions the above examples would be classified as follows:

```
g1:    invalid
g2,g3: partially valid
g4,g5: (totally) valid
```

In practical programming environments these classifications may be incorrect. For some machine architectures, g_1 may give a valid result; for others, g_5 may only be partially defined (i.e. if abs(Minint) ≠ Maxint).

As these examples indicate, the validity of a program is determined by the operational semantics of the programming language, a semantics which is dependent both on machine architecture and language implementation. In some cases the effort to do a thorough analysis may be quite significant, in particular if the question of loop termination is involved.
The concept of validity in the sense of total function may appear unusually restrictive to some readers. It is really the extension

of the major common idea inherent in several recent language proposals addressing either the type concept, as *Liskov* (1974) and *Wegbreit* (1974), or exception handling, as *Goodenough* (1975), or predicate analysis, as *Dijkstra* (1975).

In our opinion, it is simply irresponsible to consider program validity a run-time, rather than a compile-time, property. Compile-time, of course, is only meant in a generic sense, i.e. it stands for development time. Neither the lack of appropriate tools, nor the effort required, is a reason for not requesting a complete analysis. In other words, programs for which such basic aspects as arithmetic representation, subscript range or loop termination are not handled, should not be allowed to enter the testing phase.

How difficult the analysis of a program may be, it can either be performed, based on the program text alone, or not at all. Testing can and should not be used to contribute to the determination of program validity.

FUNCTIONAL CORRECTNESS AND VERIFICATION

After validity, the next important property of a program is its functional correctness. A program g is said to be *functionally correct* if the function F(g) realized by g corresponds to the *intended* function. The function realized by a program is the set of all pairs of input and output values.

An example:

```
        var  x,y: -2..3;
g6:     if   x < 0   then y:= x else y:= 0
g7:     if   x > 0   then y:= 2 * (x - x) else y:= 2 * x - x
g8:     y:= x; while y > 0 do y:= y - 1
```

The function calculated by the above three programs is

$$F(g_6) = F(g_7) = F(g_8) = \{<-2,-2>,<-1,-1>,<0,0>,<1,0>,<2,0>,<3,0>\}$$

Hence, these three programs are functionally equivalent. They differ, however, with respect to storage and time requirements, with respect to

the type of operations used, etc. These are algorithmic properties that are normally very important, but are irrelevant in this context.

Sometimes F(g) is referred to as the *graph* of the function defined by g. The program g, on the other hand, constitutes the *rule* to calculate the right hand elements of F for given left hand elements. If we assume for a moment that the intended function of a program, denoted by F*, can be specified in some formal way, then g is correct with respect to F* if the following relation holds:

$$F(g) \supseteq F^*$$

The relation 'equal to or superset of' includes the possibility that the realized function may be defined for a larger domain and hence contains more elements than the intended function. Functional *verification* is any process which shows that the above relation holds.

It follows from the above definitions that program correctness can only be determined relative to some other information. The availability of this information is the prerequisite for any form of verification.

It is a common problem for all formal verification methods that the intended function F* is usually only an abstract or vague concept. Apart from a few simple (and frequently quoted) examples, no concrete and formal definition of this alternate function specification is readily available.

A program is usually the result of a mental process, performed by one or more persons, who assembled, abstracted, and generalized various pieces of information that they may have obtained from multiple sources. For a programmer who is well-versed in his trade, the program text may indeed be the most precise and most concise means of function description available to him. However, it is typical of the human mind that at any given point it can concentrate on only a limited subset or on certain aspects of a multi-element object. A realistic verification method, therefore, has to cope with the situation that the intended function F* may itself be the product of a development effort, if it is formally specified, or which is more frequent, that it cannot be specified and remains a vague concept which takes on a certain form while time is elapsing or while different people's view

of it are being combined.

PROGRAM PROVING AND PROGRAM TESTING

The two principal methods of functional verification are *program proving* and *testing*. They differ mainly in the way that information on the intended function F* is introduced. In the case of proving, the information introduced is a pair of assertions, called *pre-condition* and *post-condition*, defining conditions that the input values and the output values have to fulfil. To be meaningful and adequate as a means of verification, these predicates have to meet two important criteria.
First, the assertions have to be strong enough to describe precisely the sets of input and output values and the correct relation between them. One may easily find predicates that describe a certain property of the output values, but are too weak to provide an alternate or dual specification of the intended function. The literature on program verification abounds with examples of this. As a simple illustration, we cite example g_5. This function is usually referred to as abs(x). The following three formulas are possible postconditions

c_1: $y \geq 0$
c_2: $y \geq x$
c_3: $y = \underline{\text{if}}\ x \geq 0\ \underline{\text{then}}\ x\ \underline{\text{else}}\ -x$

Obviously, formulas c_1 and c_2 are too weak; they describe properties of $F(g_5)$ which may apply to many other functions as well.

Secondly, the assertions (the *verificator*) must have a higher degree of authority as to the intent of the program than the program text itself (the *verificand*). This normally cannot be expected if they are developed at the same time as the program and by the same person. As investigations on programming errors have shown (see e.g. *Boehm* (1975), *Endres* (1975)), the most prevalent errors are so-called design errors. These errors occurred when either the problem to be solved was misunderstood, or the knowledge of applicable algorithmic solutions was inadequate. For this reason, it is mandatory that the development of the dual specification be the result of an independent thought process. In practice, the possibilities for achieving the desired independence between object and means of verification, i.e. between program text and assertions, are rather limited. Possible

ways include development in different languages, at different times or by different people. None of these alternatives is very practical. This may explain why the application of this method has been limited to the few demonstration cases published in the literature.

In the case of program testing, the information on the intended function F* is introduced by means of pairs of input and output values, so-called *test cases*. As we shall see later, it is extremely difficult to devise a set of test cases, or test set, which can serve as a complete specification of the intended function.

The great advantage of testing, however, is the degree of independence that can be achieved very easily. Testing exposes the graph of the function, which is a representation considerably different from the rule. In many cases, testing makes visible the element of time, which was only implied in the program text, if we think for example of programs that control mechanical devices. The key point, however, is that test cases can be developed and understood by persons who are not programmers. We can thus introduce into the verification process information that may not have been communicated in other forms before. A final argument is that programs are written to be executed. Only testing brings them into this important mode where they interact with their environment, and where errors or misinterpretations of the input data may be exposed.

Testing has remained, therefore, in spite of many critical comments, the most widely used and the most successful method of program verification. What is primarily lacking, however, is the theoretical understanding needed to develop effective strategies and meaningful criteria for test case selection and application.

PROGRAM TESTING STRATEGIES

The strategies in use today to select a set of test cases for verifying a given program fall essentially into two categories. The one group considers the program to be tested as a 'black box'. In this case two different test criteria are worth mentioning. Most frequently one performs a test with an *arbitrary sample* of input values, probably with special preference for some limit values. It is impossible in this case to say which tests were significant (unless errors were found) or which degree of verification was achieved. The only meaning-

ful test criterion under the black-box strategy is an *exhaustive* test. This means that every point in the input domain X is verified by a test case. Due to the typical number ranges of many input domains, an exhaustive test leads very frequently to astronomical time requirements and is hence not feasible.

All serious efforts to devise systematic test approaches which are also practical and feasible start out from the conviction that the black-box view is too limited. It is also our view that it is legitimate to combine in the verification process different types of information, provided it can be obtained in an objective and reproduceable way.

All methods that fall in this category make use of additional information obtained through a formal analysis of the program text. Since usually only the control structure of the program is taken into account, some authors refer to their approach as *structured* testing.

The weakest of the criteria in this group is the one requiring that every *statement* in the program should be executed at least once. For a conditional statement, this means that only one of the branches is taken. The tool needed for this level of testing is a tag or counter that is added to every statement. A much more stringent criterion, that includes the previous one as a subset, requires that every path in the program be executed at least once.

Although there appears to be a consensus that only a test covering each path at least once can be considered 'minimally thorough', nothing is said as to the degree of verification such a test has achieved. That a single execution of a path does not contribute significantly to the functional verification of the path will become clear from the following small example. For this we assume that along the respective path the function z := x*y is to be computed, where z is the output and x and y are input variables. If the test case used is the pair (x=2, y=2), the program will be considered correct even if the function actually evaluated was z := x+y.

A NORMAL FORM OF PROGRAMS

A key element for any testing strategy is the concept of a *path*. The precise definition of what constitutes a path varies among authors. We

want to use a definition which is based on the static text of a program and does not depend on the number of iterations for a loop. To do this, we imagine that the given program g is transformed into a form such that it does not contain any intermediate variables (see *Bauer* (1975)). For every output parameter, we obtain a piece of code which expresses the type of computation leading to it. It will have the form of a generalized assignment statement:

$$f := p_1 \rightarrow e_1,\ p_2 \rightarrow e_2,\ \ldots,\ p_n \rightarrow e_n$$

In this scheme, f, p and e are metavariables; f designates the output parameter, p is a Boolean expression and e is any expression. This form can be derived from a given program text via backward substitution and predicate transformations. Expression simplification, applied to both p and e, may exhibit certain redundant or identical computations.

The above form resembles *McCarthy's* (1962) notation for recursive function definitions. It will, therefore, be called *functional normal form* or, simply, *functional notation*.

Each condition form $P_i \rightarrow e_i$ will be considered as a *path*, p_i will be called *path predicate* of path i, and e_i is the corresponding *path expression*. All expressions will be in normal form as well.

Path expressions can be either elementary, i.e. depending on input parameters and/or constants only, or recursive. In the second case, they are also dependent on the output parameters. These output parameters are invocations of the expression e_i itself for some other input value. They are designated by using the name f of the output parameter as function designator. Note that recursive expressions occur in practically all cases where loop programs are considered.

The above representation of programs differs from McCarthy's notation, in that it requires that all p_i are mutually disjoint, hence the sequence of evaluation is irrelevant. We can assume that the functional analysis has ascertained that the union of all subdomains X_1 through X_n indeed covers the entire input domain X, that each expression e_i is totally defined for the subdomain X_i as described by p_i, and that termination is assured for every point in X. The following equations describe this situation:

$$X = \bigcup_{i=1}^{n} X_i$$

$$X_i = \{ \bar{v} \mid p_i(\bar{v}) \}$$

$$i \neq j \supset X_i \cap X_j = \{\} \qquad i,j = 1..n$$

Here $\bar{v}$ designates a particular tuple of values from the range of the input parameters v_1 through v_m, and n is the number of paths in a program. We do not require that n be minimal, that means, that any two path predicates p_i and p_j where e_i is identical to e_j be combined into a single path with the path predicate $p_i \vee p_j$.

The functional normal form described above is a very inefficient representation of programs by any convential measure, e.g. storage or time requirement. It is advantageous, however, in that it exhibits most clearly the functional dependency between input and output parameters.

ERROR AND TEST NEIGHBOURHOODS

The semantic analysis and the transformation into normal form will have eliminated an entire set of typical programming errors that have to do with the overall functional characteristics of the program, like its input domain, the number of independent paths, the number of input and output parameters, etc.

For the next step in the verification process, we take those properties as given. The type of errors that may be left have to do with the detailed functional specification of path predicates and path expressions. Similar to *Howden* (1976), the relevant properties can be classified as follows

a) correct path expression
b) correct path predicate
c) correct association between path predicate and path expression

These properties can, in fact, be verified through testing. Even assuming the normal form as described before, there are still many different valid programs that are functionally equivalent, e.g. all permutations and modifications of parameter names. They are indistinguishable to any test procedure.

We therefore introduce as our universe of discourse the space DF(G), which is the space of all functions F definable by programs in G. This space can be substructured in many subspaces, depending on some pro-

perties of elements in G. As an example, all single parameter programs form such a subspace, all two parameter programs, etc. If we have to verify a program g, we only have to verify it with respect to the subspace of functions in which it falls due to the properties derived by the associated analysis of g. We call such a subspace a *neighbourhood* of F(g). A neighbourhood of a function always includes the respective function.

The most interesting neighbourhood of F(g), and the one that would be most useful for any functional verification, will be called *error neighbourhood* EN(g). We can imagine it as being the smallest subspace of definable functions in DF(G) containing F(g) and all other functions which can result from errors made while writing g. If we would have sufficient and reliable empirical data as to the type of errors most likely to occur when using certain construction elements in g, we could define such a tolerance range for g and consequently for F(g). Testing of a program would then mean testing the equivalence of two functions F(g) and F*, both of which are elements of the subspace EN(g).

Unfortunately we are unable to define precisely the error neighbourhood of F(g). Therefore, we shall address the problem from the opposite direction by asking: What are function spaces in which we can determine the equivalence of two functions through a point-by-point comparison of a subset of the function's elements? We call such a space the *test neighbourhood* TN(g) of F(g) (see Figure 1). Note that EN(g) and TN(g) are really abbreviations for EN(F(g)) and TN(F(g)) respectively.

If such a function space exists, relative to a given g, it must be a parameterization of g, i.e. we must be able to introduce additional degrees of freedom in g and eliminate these again by point-to-point comparison with the intended function F*. Of course, we must also first ascertain that F* belongs to the same space as well.

A well-known class of functions, where individual member functions can be uniquely determined by a finite number of points, are the algebraic polynominals.

Before applying this concept to any particular class of functions, we have to define more stringently some previously used terms. We refer to a *test case* t as any tuple

$$t = \langle \bar{v}, f \rangle$$

of corresponding input and output values. A test case set, or *test set*, T(TN), is a minimal set of test cases, for a given test neighbourhood, which is adequate in a sense to be defined.

$$T(TN) = \{ t_i \mid i=1..s \ \& \ Cond_{TN}(\bar{t}) \}$$

Here s stands for the number of test cases required to make up the set T(TN) and $Cond_{TN}(\bar{t})$ is a predicate to be fulfilled by the test cases t_1 through t_s belonging to the set. The predicate $Cond_{TN}(\bar{t})$ usually does not select a specific set T as the only adequate set of test cases, but rather an entire *class* TS of *test sets* of which T is an element; hence

$$T(TN) \in TS(TN)$$

If the path expression under consideration is limited by a path predicate p_i, then the applicable test cases have to be elements of the appropriate subdomain

$$TS_i(TN) = \{ T_j \mid j = 1..s \ \& \ Cond_{TN}(\bar{t}) \ \& \ p_i(\bar{v}) \}$$

Mathematically, each TS_i is a set of sets. There are many possible sets T that satisfy TS. Any individual member T may be selected at random.

TEST SELECTION FOR ELEMENTARY EXPRESSIONS

The type of path expressions considered first are elementary (i.e. non-recursive) arithmetic expressions without division. Their normal form are polynominals.

If for a given path expression e_i, we know that it depends on a single input variable only and the degree of this variable (in the normal form of the expression) is less than or equal to k, then the equivalence of $F_i(g)$ and F_i^* can be shown by k+1 test cases. This follows from the central law of algebra, saying that a polynominal of degree k is uniquely determined if k+1 disjoint points are given.
If v is the independent variable, the polynominals of degree k can be expressed by the following expression scheme:

$$P_1^k: a_k * v^k + a_{k-1} * v^{k-1} + \ldots + a_1 * v + a_0 \qquad \text{or}$$

$$\sum_{i=0}^{k} a_i * v^i$$

An individual member of this class is determined if the coefficients a_0 through a_k are determined. By using k+1 value pairs <v,f>, we obtain k+1 linear equations which can be solved for the k+1 'unknowns' a_0 through a_k. Hence the number s of test cases is

$$s = k+1$$

The same principle applies to polynominals with m variables ($m \geq 1$). The corresponding scheme for this class of functions is

$$P_m^k : \sum_{0 \leq i_1 \leq k} a_{i_1 \ldots i_m} * v_1^{i_1} * \ldots * v_m^{i_m} \qquad l = 1..m$$

In this scheme, different independent variables are designated by v_1 through v_m. While m is the number of variables, k is the highest degree in one of the variables. The summation above occurs over the set of m-tuples $(i_1,\ldots,i_m)$, such that $0 \leq i_1 \leq k$ and l = 1..m. The number s of coefficients in this case is

$$s = (k+1)^m$$

In order that the respective system of linear equations becomes solvable, the points (test cases) chosen have to be linearly independent of each other. This results in the condition that for any chosen set of test cases, the determinant $D(\bar{v})$ formed from the row vectors has to be different from zero, or

$$Cond_{TN}(\bar{t}): \quad D(\bar{v}) \neq 0$$

For examples including division, the normal form is (sometimes) a quotient of two polynominals. Here additional conditions arise, e.g. that the value of the polynominal in the denominator has to be different from zero. Since the properties of the division operator result in many other restrictions, we shall not treat this class any further.

What we have done above is the following. For a given path expression e_i, we have chosen as test neighbourhood the class p_m^k of expressions. In addition, we have to pick upper bounds for both m and k. The number

m of variables occuring on a given path is certainly limited by the total number of input variables to the program. The highest degree k of any variable is probably often a judgemental decision that should be made jointly by implementer and tester. In large classes of applications, degrees higher than 2 are unlikely to occur. Any erroneous expressions may in this case be caught via the semantical analysis.

If an error is made as to the functional class chosen as test neighbourhood, the result can be misleading. The approach is, therefore, only valid relative to a correctly chosen test neighbourhood.

If the program consists of multiple paths, the correct association between path condition p_i and path expression e_i can be verified if all test cases for e_i are chosen such that the output values are different from all output values of the test cases for other paths. This will give rise to additional conditions for each test set.

TEST SELECTION FOR RECURSIVE EXPRESSIONS

Since loops within a program will most likely result in recursive path expressions, it is important to treat this class as well. We make the restriction again that all expressions and subexpressions are polynominal (i.e. excluding division).
The functional scheme for this class of expressions can be given as follows:

$$R_{m,r}^{k} : \sum_{0 \le i_1 \le k} a_{i_1 \ldots i_{m+r}} * v_1^{i_1} * \ldots * v_m^{i_m} * u_1^{i_{m+1}} * \ldots * u_r^{i_{m+r}}$$

$$\text{with } u_j = f(h_j(\bar{v})), \quad j = 1..r, \; l = 1..m+r$$

In this scheme the parameters to be picked are
- m: the number of different independent variables
- k: the highest degree in one of these variables
- r: the number of different recursive invocations of the function $f(\bar{v})$.

We consider two recursive invocations of a function as different if their respective argument expressions $h_j(\bar{v})$ are different. The degree k applies to variables occuring either in the auxiliary functions $h_j(\bar{v})$ or in the main function.

When calculating the number of coefficients to be determined (and hence

the number of test cases required), we treat each different invocation of f like an additional independent variable u_j (j=1..r). The corresponding formula is therefore

$$s = (k+1)^{m+r}$$

By doing this we pretend that the function to be verified is from a higher function space, i.e. it has more degrees of freedom, than is actually the case.

In addition to the function $f(\bar{v})$ itself, we have to be able to determine the coefficients of the auxiliary functions $h_j(\bar{v})$ as well. This increases the number of test cases further. On the other hand, any independent test case t_i may trigger the execution of several dependent test cases t_j, t_{j+1}, etc. The proper relation between values of h_j and f is established by keeping track which test case t_i may trigger the execution of several dependent test cases t_j, t_{j+1}, etc. The proper relation between values of h_j and f is established by keeping track which test case t_i relies on which other test case t_j. For a successful verification,not only the values of all independent and dependent test cases have to match with the expected values, but t_j must be a correct predecessor of t_i in the history of the function.

This point can be explained in terms of recursive function theory as follows. A recursive function f computes its n^{th} value based on the history up to the $n\text{-}1^{th}$ value. What has to be verified is how the n^{th} value is being calculated, based both on the actual input parameters $\bar{v}$ and on that subset of values from the history of f that are being used for this step.

AN EXAMPLE

As an illustration of the previously developed concepts we select a well-known loop program, namely the determination of the greatest common divisor (gcd). We first give the program in loop notation.

```
function z(x,y: integer): integer;
begin while x≠y do
      if x>y then x := x-y else y : = y-x;
      z := x end
```

Without giving the details, we perform a transformation into our functional notation where this program takes on the form:

$$f := p_1 \rightarrow e_1,\ p_2 \rightarrow e_2,\ p_3 \rightarrow e_3$$

with p_1: x>y & x>0 & y>0 and e_1: z(x-y,y)
p_2: x<y & x>0 & y>0 e_2: z(x,y-x)
p_3: x=y & x>0 & y>0 e_3: x

We are dealing with three path expressions e_1 through e_3, one of which is elementary and two are recursive. The dependent variable is called z, the independent variables are x and y. The domain X of this function is the set

$$X = X_1 \cup X_2 \cup X_3 = \{ \langle x,y \rangle \mid \langle x,y \rangle \in Int \ \& \ x>0 \ \& \ y>0 \}$$

This information on the actual domain is being derived from the semantic analysis of the program and is independently verified.

We now look at the individual paths. We start with the third one, since its path expression e_3 is non-recursive. This gives us at the same time an illustration for any loop-free program. The decision that must be made in order to proceed is to pick the proper test neighbourhood. We choose the class p_2^1 which means that no occurence of a variable of degree higher than 1 is anticipated. Here is where analysis and testing complement each other. The property assumed when testing has to be consistent with what the analysis has yielded. In terms of the variable names x and y, the above class is represented by the scheme:

$$P_2^1(x,y): a_3*x*y + a_2*x + a_1*y + a_0$$

The test set class TS, which is adequate to verify any function relative to this function scheme, can be characterized by the following set equation

$$TS(P_2^1(x,y)) = \{ \langle x_i,y_i,z_i \rangle \mid i = 1..4 \ \& \ D(x,y) \neq 0 \}$$

Here suffixes are used to designate individual values from the range of the variables x, y and z. The notation D(x,y) is an abbreviation for the following determinant

$$D(x,y) = \begin{vmatrix} x_1*y_1 & x_1 & y_1 & 1 \\ x_2*y_2 & x_2 & y_2 & 1 \\ x_3*y_3 & x_3 & y_3 & 1 \\ x_4*y_4 & x_4 & y_4 & 1 \end{vmatrix}$$

It should be noted that this determinant is not particular to the program in question, but rather to the function class P_2^1. TS is a set of 4-element sets.

If we pick three of the four test cases at random, then the determinant D(x,y) results in a condition that the fourth test case has to fulfil. As an example, we choose the following values for the three first test cases:

$$\{ <1,1,1>,<2,2,2>,<3,3,3> \}$$

Then the condition resulting from the determinant is

$$\begin{vmatrix} 1 & 1 & 1 & 1 \\ 4 & 2 & 2 & 1 \\ 9 & 3 & 3 & 1 \\ x_4*y_4 & x_4 & y_4 & 1 \end{vmatrix} \neq 0$$

Solving the determinant leads to the inequation

$$y_4 \neq x_4$$

Hence

$$\{<1,1,1>,<2,2,2>,<3,3,3>,<1,2,1>\} \in TS(P_2^1)$$

In our particular case, the condition for the fourth test case is in conflict with the path condition

$$p_3: x = y$$

This reflects the fact that the same function would be computed if in the program x were replaced by y. In other words, the class to be verified is really P_1^1, and not P_2^1; hence two test cases are already sufficient.

We now consider the two recursive path expressions e_1 and e_2. Making

the same decisions as before, the test neighbourhood to be picked is the class $R^1_{2,1}$. In terms of the variables x and y, the corresponding scheme is

$$R^1_{2,1}(x,y) : a_7*u*x*y + a_6*u*x + a_5*u*y + a_4*x*y + a_3*u + a_2*x + a_1*y + a_0$$

In this case u stands for

$$u = z(h_1(x,y), h_2(x,y)),$$

where h_1 and h_2 are considered to be elementary functions from the class P_2^1. In order to also verify these auxiliary functions, we have to determine the coefficients in the following two schemes

$$h_1(x,y): b_3*x*y + b_2*x + b_1*y + b_0$$
$$h_2(x,y): c_3*x*y + c_2*x + c_1*y + c_0$$

A test set for this function class has to be a member of the following set of sets

$$TS(R^1_{2,1}(x,y)) = \{ <x_i,y_i,z_i> \mid i = 1..8 \ \& \ D(u,x,y) \neq 0 \ \& \ D_{h_1}(x,y) \neq 0 \ \& \ D_{h_2}(x,y) \neq 0 \}$$

Here three different determinants occur, corresponding to the three different functional schemes to be tested. They are

$$D(u,x,y) = \begin{vmatrix} u_1*x_1*y_1 & u_1*x_1 & u_1*y_1 & x_1*y_1 & u_1 & x_1 & y_1 & 1 \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ u_8*x_8*y_8 & u_8*x_8 & u_8*y_8 & x_8*y_8 & u_8 & x_8 & y_8 & 1 \end{vmatrix}$$

$$D_{h_1}(x,y) = D_{h_2}(x,y) = \begin{vmatrix} x_1*y_1 & x_1 & y_1 & 1 \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ x_4*y_4 & x_4 & y_4 & 1 \end{vmatrix}$$

In this case it is not practical to first pick 7 test cases at random and then derive a condition for the eighth. Rather, a software tool

could be conceived to keep a record of which dependent test cases are triggered by a randomly chosen independent test case.

Figure 2 gives an example of such a test record. It comprises three test sets T_1 through T_3 corresponding to the three paths 1 through 3. Note that both T_1 and T_2 show nine test cases each, the ones in parenthesis, however, are insignificant and hence not part of the set of relevant tests. All independent and arbitrary chosen test cases are numbered, with circled numbers ① through ⑧. These eight test cases trigger the execution of 14 other different test cases. Also, the predecessor relation between test cases is given by an appropriate arrow. Note that test cases for one path may trigger a test case for another path. The imagined test tool should, for each new test case, check its significance for the appropriate path and gradually fill out and evaluate the respective determinants.

While the determinant $D(u,x,y)$ is the real defining criterion for the corresponding test set, the determinants $D_{h_1}(x,y)$ and $D_{h_2}(x,y)$ may be fulfilled by any 4 out of the 8 test cases for a path. For path 1, the test cases ①, ② plus its dependent test case, and ③ already result in the appropriate condition.

That the set T_1 in Figure 2 is an adequate set to verify the function computed along path 1 relative to the class $R_{2,1}^{1}$ follows from the fact that the two determinants $D(u,x,y)$ and $D_h(x,y)$, as given below, are both different from zero.

$$D(u,x,y) = \begin{vmatrix} 20 & 5 & 4 & 20 & 1 & 5 & 4 & 1 \\ 3 & 3 & 1 & 3 & 1 & 3 & 1 & 1 \\ 2 & 2 & 1 & 2 & 1 & 2 & 1 & 1 \\ 96 & 16 & 12 & 48 & 2 & 8 & 6 & 1 \\ 108 & 36 & 9 & 36 & 3 & 12 & 3 & 1 \\ 54 & 18 & 9 & 18 & 3 & 6 & 3 & 1 \\ 384 & 48 & 32 & 96 & 4 & 12 & 8 & 1 \\ 128 & 32 & 16 & 32 & 4 & 8 & 4 & 1 \end{vmatrix} = 155\ 520$$

$$D_h(x,y) = \begin{vmatrix} 20 & 5 & 4 & 1 \\ 3 & 3 & 1 & 1 \\ 2 & 2 & 1 & 1 \\ 48 & 8 & 6 & 1 \end{vmatrix} = 45$$

A test of the correct association between path predicates and path expressions is not performed in this example.

SUMMARY

In this paper we have presented a first step towards a practical and systematic approach to program verification by combining program analysis and testing. For certain classes of programs, we were able to clarify the question of what information can be obtained through testing and what constitutes an adequate test case set. It will be very interesting to extend this concept to other classes of programs, in particular to programs operating on data structures, lists or non-numerical data.

The testing strategy resulting from the proposed approach offers the great advantage that the information gained and the degree of verification performed increases in proportion to the effort invested. This makes it superior to most other testing strategies currently in use. The major disadvantage of the approach is the size of effort needed to establish and evaluate the determinants for each test set. This cost may be compensated by the savings achieved by the elimination of redundant test runs.

As other formal methods continue to fall short of their expectations, it becomes increasingly important that we better understand the role that testing can and should play. We hope that this paper may spur on more work in this important area.

LITERATURE

Bauer, F.L. : Variables considered harmful,Tech. Univ. Munich, Report 7513 (1975)

Boehm, B.W. et al: Some experience with automated aids to the design of large-scale reliable software, Proceedings Intern. Conf. on Reliable Software, Los Angeles (1975).

Dijkstra, E.W.: Notes on structured programming, Eindhoven Technical University, Report EWD 249 (1969).

Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs, Comm. ACM 18,8 (1975)

Endres, A.: An analysis of errors and their causes in system programs, IEEE Transactions on Software Engineering, 1,2 (1975).

Endres, A.: Analyse und Verifikation von Programmen, R. Oldenbourg, Munich (1977)

Goodenough, J.B. and Gerhart, S.L.: Toward a theory of test data selection, IEEE Transactions on Software Engineering, 1,2 (1975).

Goodenough, J.B.: Exeption handling: Issues and a proposed notation, Comm. ACM 18,12 (1975)

Howden, W.E.: Reliability of the path analysis testing strategy, IEEE Trans. in Software Engg. 2,3 (1976)

Huang, J.C.: An approach to program testing, ACM Comp. Surveys 7,3 (1975).

Liskov, B.H.: A note on CLU, MIT Project MAC, Memo 112 (1974).

McCarthy, J.: Towards a mathematical science of computation, Proceedings IFIP Congress, Munich (1962).

Miller, E.F. and Paige, M.R.: Automatic generation of software test cases, Proceedings Eurocomp Conference, Uxbridge (1974).

Miller, E.F.: Program testing: art meets theory, IEEE Computer 9,5 (1977)

Mills, H.D.: Top down programming in large systems, In Debugging Techniques in Large Systems, R.Rustin (ed), New York University (1971).

Wegbreit, B.: The treatment of data types in EL1, Comm ACM 17,5 (1974).

Wirth, N.: Program development by stepwise refinement, Comm ACM 14,4 (1971).

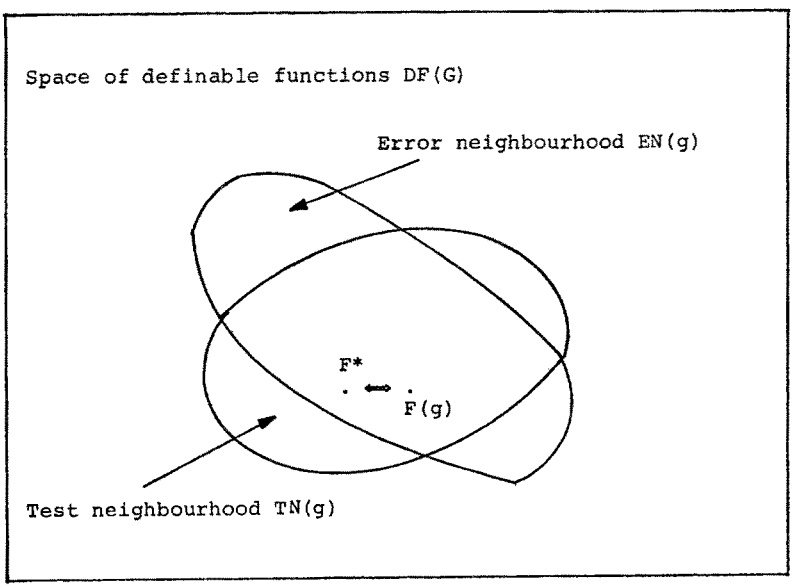

Fig. 1: Error and test neighbourhoods of a function F(g)

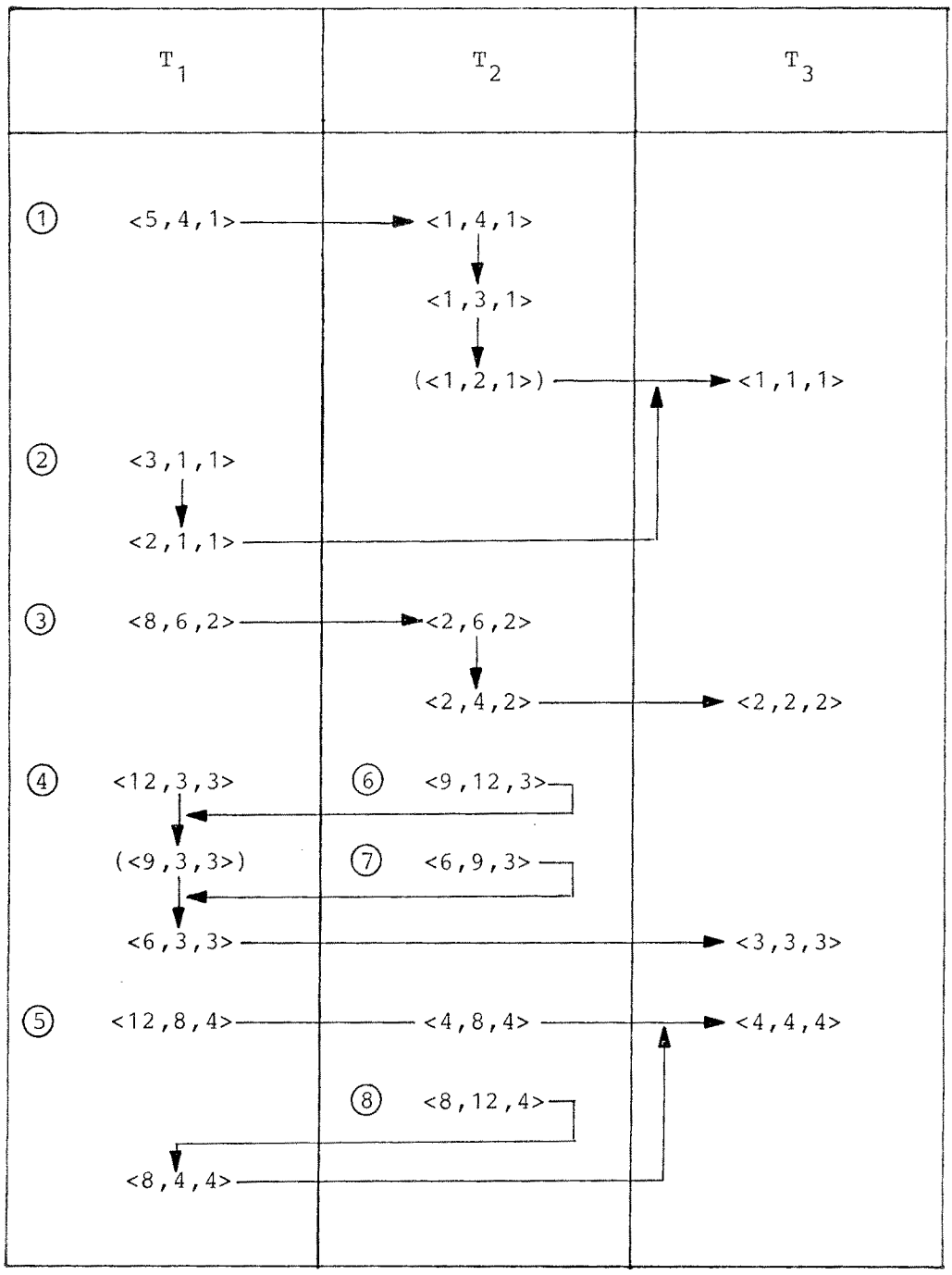

Fig. 2: Test record for the sample program