Insights into Regression Testing †

Hareton K. N. Leung

Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1

Lee White

Department of Computer Engineering and Science
Case Western Reserve University
Cleveland, Ohio 44106

Abstract

Regression testing is a significant, but largely unexplored topic. In this paper, the problem of regression testing is described in general terms and its characteristics are contrasted with those of testing. Regression testing can be grouped into *progressive regression testing* and *corrective regression testing*, depending on whether the specification is changed or not, respectively. The test cases can be grouped into five classes: *reusable, retestable, obsolete, new-structural* and *new-specification* test cases. A problem facing those conducting maintenance testing is to identify the proper test classes. The notion of *regression testable* is introduced as a way to measure the ease of retesting. Both the program design and the test plan design may affect the regression testability of a program. The *testing set, testing number, regression number, stable* and *workable* metrics are developed to measure the program and the test plan. We propose the *Retest Strategy* for performing corrective regression testing. The guiding principle of Retest is to view the regression testing problem as composed of two subproblems: the *test selection problem* and the *test plan update problem*, and to structure the retesting process into two phases: the *test classification phase* and the *test plan update phase*.

1. Introduction

Regression testing is a testing process which is applied after a program is modified. It involves testing the modified program with some test cases in order to re-establish our confidence that the program will perform according to the (possibly modified) specification. In the development phase, regression testing may begin after the detection and correction of errors in a *tested program*. A tested program is a program which has been tested with a *high quality* test plan. Regression testing is a major component in the maintenance phase where the software system may be corrected, adapted to new environment, or enhanced to improve its performance. Modifying a program involves creating new logic to correct an error or to implement a change and incorporating that logic into an existing program. The new logic may involve minor modifications such as adding, deleting, rewriting a few lines of code, or major modifications such as adding, deleting or replacing one or more modules or subsystems. Regression testing aims to check the correctness of the new logic, to ensure the continuous working of the unmodified portions of a program, and to validate that the modified program as a whole functions correctly.

Most people have assumed that regression testing is simply repeating all the tests in the test plan and retesting all the features of the program [17, 2]. Recently, regression testing researchers have begun to emphasize retesting only the modified and new features of the program [19, 5].

1.1. Basic Concepts

To facilitate the presentation, we define some of the terminology that will be used in the sequel. The initial description is restricted to single routines to simplify the presentation, as the general theory can be applied to complete programs. A routine can be represented by a directed graph, called a *control flow graph*, G=(N,E,n_s,n_f), where N is a set of nodes and E is a set of edges in NxN. Each node in the graph represents an executable *instruction*, while the edge, denoted by (n_i, n_j), indicates that a possible transfer of control exists from node n_i to node n_j. We assume there exists a single entry point, the start node, n_s, and a single exit point, the final node, n_f, in the control flow graph. If necessary a null node can be added to the graph for the start node and likewise for the final node to convert the graph into single entry, single-exit. A *subpath* from n_{j_1} to n_{j_k} of length k is a list of nodes ($n_{j_1},\ldots,n_{j_i},..,n_{j_k}$) such that for all i, $1 \le i \le k-1$, (n_{j_i}, $n_{j_{i+1}}$) is in E. A *path* is a subpath that begins at the start node, n_s, and ends at the final node, n_f. Some paths may be nonexecutable due to contradictory conditions on the transfer of control along the paths. A path is *feasible* if there exists input data which causes the path to be traversed during program execution. Taking a definition given in [9], we will call a feasible path that has actually been executed for some input a *trajectory*.

In a control flow graph each node corresponds to either a simple statement or a conditional expression. A node can represent an assignment statement, an input or output statement, the conditional expression of an if-then-else, or the conditional expression of a while statement. A node representing a conditional expression will be called a *condi-*

† This work was partially supported by a grant from the Natural Science and Engineering Research Council of Canada under grant number OGP2517.

Table 1. Differences between Corrective and Progressive Regression Testing	
Corrective regression testing	Progressive regression testing
• Specification is not changed • Involves minor modification to code (e.g., adding and deleting statements) • Usually done during development and corrective maintenance • Many test cases can be reused • Invoked at irregular intervals	• Specification is changed • Involves major modification (e.g., adding and deleting modules) • Usually done during adaptive and perfective maintenance • Fewer test cases can be reused • Invoked at regular intervals

tional node. The conditional expression will be treated as an instruction, called a *conditional instruction.* All other instructions will be referred to as *non-conditional instructions.*

1.2. Interaction of Instructions and the All-Essential Assumption

It is possible for a statement to be executed and still not be essential to the computation being performed. On a particular path, there may be some instruction which does not affect any output variables (in terms of control flow and data flow), while the same instruction may affect output variables on another path. An *essential instruction* of a path P_i is an instruction on P_i which may affect an output variable. To determine the non-essential instructions of a path P_i, P_i must first be generated and then control flow and data flow analysis be applied to the path. In order to reduce the amount of dynamic analysis, we will make the following simplifying assumption:

All-essential assumption:

Every instruction on a path affects the overall path computation.

The all-essential assumption implies that if an instruction is executed by s test cases, then its subcomputation is used by all s test cases. If I is modified, then all test cases which traversed I should be rerun because some change may occur to the path computation. Observe that the all-essential assumption may not hold for all programs.

2. Regression Testing

In the development phase, regression testing may begin after the detection and correction of errors in a program. At the last stages of program development when the program has been reasonably tested, testing is aimed at revealing the hidden persistent software errors [4]. At this stage, a well-developed test plan should be available. It makes sense to reuse the existing test cases, rather than redesigning all new test cases, in retesting the program after it is corrected for any errors.

Many modifications may occur during the maintenance phase where the software system is corrected, updated and fine tuned. Software maintenance is defined as the performance of those activities required to keep a software system operational and responsive after it is accepted and placed into production [13]. There are three types of modifications, each arising from different types of maintenance. According to [12], *corrective maintenance,* commonly called "fixes", involves correcting software failures, performance failures, and implementation failures in order to keep the system working properly. Adapting the system in response to changing data requirements or processing environments constitutes *adaptive maintenance.* Finally, *perfective maintenance* covers any enhancements to improve the system processing efficiency or maintainability.

During adaptive or perfective maintenance, new modules are usually introduced. The specification of the system is modified to reflect the required improvement or adaptation. However, in corrective maintenance, the specification is not likely to be changed and no new modules are likely to be introduced. Most modifications involve adding, deleting and modifying instructions. Many program modifications occurring during the development phase are similar to that of corrective maintenance, since we don't expect the specification will normally be modified due to a discovery of an error in the program.

2.1. Types of Regression Testing

Two types of regression testing can be identified based on the possible modification of the specification. *Progressive regression testing* involves a modified specification. Whenever new enhancements, or new data requirements are incorporated in a system, the specification will be modified to reflect these additions. In most cases, new modules will be added to the software system with the consequence that the regression testing process involves testing a modified program against a modified specification.

In *corrective regression testing,* the specification does not change. Only some instructions of the program and possibly some design decisions are modified. This has important implications because most test cases in the previous test plan are likely to be valid in the sense that they correctly specify the input-output relation. However, because of possible modifications to the control and data flow structures of the software, some existing test cases are no longer testing the previously targeted program constructs. The corrective regression testing is often done after some corrective action is performed on the software.

Table 1 lists the major differences between corrective and progressive regression testing. Typically, progressive

regression testing is done after adaptive or perfective maintenance, while corrective regression testing is done during testing in the development cycle and after corrective maintenance. Since adaptive or perfective maintenance is typically done at a fixed interval, for example, every six months, progressive regression testing is usually invoked at regular intervals. By contrast, program failures can occur any time and most of them need to be corrected immediately. Thus, corrective regression testing may be invoked after every correction.

2.2. Differences between Testing and Regression Testing

Most people assume that regression testing is a simple extension of testing. However, it is not always the case. There are several major differences between these two processes.

Availability of test plan Testing begins with a specification, an implementation of the specification and a test plan with test cases added during the specification, design and coding phases. All these test cases are *new* in the sense that they have not been used to exercise the program previously. Regression testing starts with a possibly modified specification, a modified program and an old test plan which requires updating. All test cases in the test plan were previously run and were useful in testing the program.

Scope of test The testing process aims to check the correctness of a program, including its individual components (e.g., functions and procedures) and the interworking of these components. Regression testing is concerned with checking the correctness of parts of a program. The portion of a program which is not affected by modifications need not be retested. An interesting problem is to determine the affected portion of a program in an efficient manner. A solution to this problem may be the use of *retestable units [11]*.

Time allocation Testing time is normally budgeted before the development of a product. This time can be as high as half the total product completion time. However, regression testing time, especially time for corrective regression testing, is not normally included in the total product cost and schedule. Consequently, when regression testing is done, it is nearly always performed in a crisis situation. The tester is normally urged to complete retesting as soon as possible and most often is given limited time to retest.

Development information In testing, knowledge about the software development process is readily available. In fact, the testing group and the development group may be the same. Even if an organization has a separate testing group, the testers can usually query the developers about any uncertainty in the software. But in regression testing, the testers most likely will not be the developers of the product. Since regression testing may be done at a different time and place, the original developers may no longer be available. This situation suggests that any relevant development information should be retained if regression testing is to be successful.

Completion time The completion time for regression testing should normally be less than that for testing since only parts of a program are being tested.

Frequency Testing is an activity which occurs frequently during code production. Once the software product is put into operation, testing is completed and any further testing will be considered as regression testing. Typically, regression testing is applied many times throughout the life of a product, once after every modification is made to an operating product.

2.3. Similarities between Testing and Regression Testing

Several aspects of regression testing are similar to that of testing. In particular, the purposes and testing techniques used are almost the same.

Purposes The purposes of testing and regression testing are quite similar. They both aim to:

1) increase one's confidence in the correctness of a program, and
2) locate errors in a program.

Some additional goals of regression testing are to:

3) preserve the quality of the software; the modified software should be at least as reliable as its previous version; this may be achieved in many ways; one possible method is to insist that the same structural coverage is achieved by both versions of the software;
4) ensure the continued operation of the software; this is an important goal because some users may become dependent on the software product, and software developers have a responsibility to continue to provide the same service to users.

Testing techniques Since test cases in a test plan depend on the chosen testing technique, the testing technique used by both testing and regression testing should be the same if the regression testing process involves the reuse of test cases. If regression testing were to involve a different testing technique, then it would be difficult to reuse the existing test plan. Another reason for using the same testing technique is that it is easier to evaluate the quality of two software products if they are tested by the same technique. At the current state of the art, it is difficult to compare the relative test effectiveness of two different testing techniques.

2.4. Test Classes for Regression Testing

There are two common techniques in generating test cases. The first method is to generate test cases based on specification, i.e., black box testing. One specification-based testing method is functional testing as described recently by Howden [8]. Contrary to Howden's approach, we assume all functional tests are created based on specifications, not on design or program information.

Another common testing method is to apply structural testing which involves test cases based on the control or data flow structures of a program. Structural testing requires the tester to design tests to execute certain components of a program, or some combinations of them. If no error has been detected during testing according to this strategy, then the tester's confidence about the program reliability is increased. We will call test cases created from black box testing tech-

niques *specification-based test cases* and those from structural testing *structural-based test cases*.

Both testing methods have deficiencies. Empirical research indicates that using functional testing or structural testing alone cannot detect all errors in a program [6]. Many researchers in the field have advocated that both testing techniques be used to complement each other. We will assume that the test plan will include both *types* of test cases.

Notice that all specification-based test cases execute some components of a program and thus their trajectory results can be included for structural measures. Thus, specification-based test cases may be used to satisfy the structural coverage criteria. However, the converse is not true. Not all structural-based test cases can be used as specification-based test cases because some structural-based test cases are designed to test a certain program component and this program component may have no functional meaning by itself; it is solely used with other components to synthesize a function. A common practice in testing is to first generate specification-based test cases, and then augment a test plan with additional structural-based test cases so that the structural testing strategy (e.g., statement, branch, or dataflow measures such as data contexts [10], or required-pairs [15]) is satisfied.

In the sequel, we first give the test case classification for progressive regression testing and then a simpler one for corrective regression testing. Typically, only a part of the specification is modified in progressive regression testing. The test cases which test the unmodified parts of the specification will remain valid, despite the specification and program modifications. Observe that if a specification is modified, then the program constructs implementing the specification must also be modified. After a modification is made to the software, we can classify the test cases in the previous test plan into the following mutually exclusive classes:

1) Reusable test cases (R_t) - This class includes both types of test cases. These test cases are testing the unmodified parts of the specification, and their corresponding unmodified program constructs. Reusable test cases need not be rerun because they will give the same results as previous tests.
2) Retestable test cases (T_t) - This class includes both types of test cases which should be repeated because the program constructs being tested are modified, although the specification for the program constructs are not modified. Observe that although these test cases specify the correct input/output relations, they may not be testing the same program constructs as before the modification.
3) Obsolete test cases (O_t) - This class includes both types of test cases that can no longer be used. There are three ways that a test case may become *obsolete:*
   a) If a test case specifies an incorrect input/output relation due to a modification to the problem specification, then it can no longer be used.
   b) If the targeted program component has been modified, some test cases may correctly specify the input/output relation, but may not be testing the same construct. For example, in domain testing [18], test cases are derived to test the borders of each domain. If some predicate in a program is modified, then some domain borders will be shifted. Although some test cases for testing the old borders still specify the correct input-output relation, they no longer test the new borders effectively.
   c) Finally, a structural test case may not contribute to the structural coverage of the program. Since all structural tests are designed to increase the structural coverage of the program, any structural test which does not increase the coverage measure can be deleted during the testing phase.

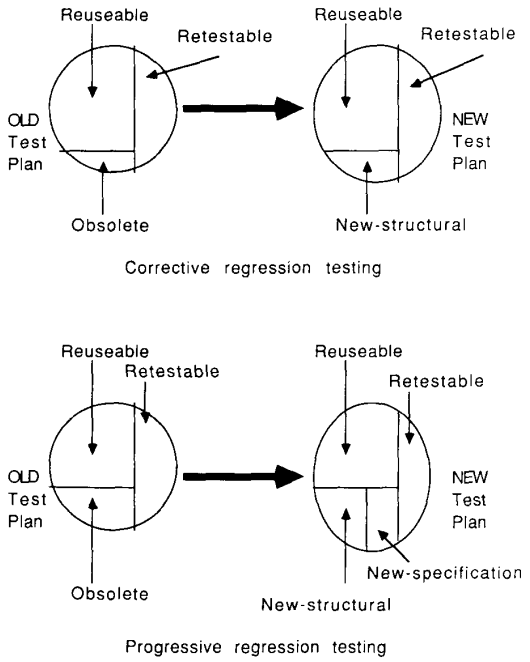

Figure 1. Evolution of Test Plan

Extensive analysis may be needed to distinguish retestable test cases from obsolete test cases. We will use the term *unclassified test cases* to refer to test cases which may either be retestable or obsolete. After the modification, two new classes of test cases may be introduced in a test plan:

1) New-structural test cases (S_t) - This class includes structural-based test cases that test the modified program constructs. They are usually designed to increase the structural coverage of the program.
2) New-specification test cases (N_t) - This class includes only specification-based test cases. These test cases test

Table 2. Classification of test cases according to the specification and the program constructs being tested.			
Test Class	Specification	Target Construct	Test Type
reusable	unchanged	unchanged	structural, specification
retestable	unchanged	changed	structural, specification
obsolete	unchanged	changed	structural
	changed	unchanged/changed	specification
new-structural	unchanged/changed	new	structural
new-specification	changed	new	specification

the new code generated from the modified part of a specification.

Corrective regression testing can be viewed as a special case of progressive regression testing where the specification is not modified. It is easy to show that corrective regression testing may involve reusable, retestable, obsolete and new-structural test cases. There are no new-specification test cases in corrective regression testing because the problem specification is not modified. The changes that may occur in a test plan under both types of regression testing are illustrated in Figure 1. Observe that the area in each region represents the number of test cases. The total number of test cases in a test plan may change with each application of regression testing. It should be noted that the input domain for progressive regression testing may change due to modifications occurring in the specification. For example, the specification may change from accepting input of positive integers to accepting input of all integers. Table 2 summarizes the change relationships between test class, specification, program construct, and test type. The target construct denotes the program constructs exercised by the test cases.

2.4.1. An Example

```
1     read(x,y)
2     if (y < 0) then
3        power = -y
      else
4        power = y
      endif
5     z = 1
6     while (power <> 0) do
7        z = z * x
8        power = power - 1
      end
9     if (y < 0) then
10       z = 1 / z
      endif
11    answer = z
12    write(answer)
```

Figure 2. An example program.

The program [16] shown in Figure 2 can be used to illustrate the above test case classification. This program calculates x to the power y, with both x and y integers. From the specification, we can find some natural ways of subdividing the x and y domains. Suppose the specification-based test cases include all combinations of the following values:

x input: x<-1, x=-1, x=0, x=1, x>1

y input: y=0, y=1, y<0 (assume the test case used is y=-1)

This program has infinitely many feasible paths and infeasible paths due to the presence of a loop. For example, any path that includes (1,2,3,...,9,11,12) is infeasible. Thus, test data cannot be generated for each individual path domain. Suppose the structural testing strategy requires all feasible paths be executed and for the loop, one path which skips the loop, one which iterates the loop once, and one which iterates the loop two or more times be tested. After running the program with specification-based test cases, we add the following structural-based test case to complete the structural testing requirement: y>1 (assume actual test case used is y=2) and x=4. Observe that the set of structural-based test cases may be different if a different testing strategy is used.

Suppose a faster program is needed. By taking advantage of the binary representation of y, a method which performs the computation in time proportional to log y can be used. Two extra instructions (6.2 and 8.2) are added to the original program, as shown below. After the change, since there is no modification in the problem specification, corrective regression testing should be performed.

```
6     while (power <> 0) do
6.2      if odd(power) then
7           z = z * x
         endif
8        power = power - 2
8.2      x = x * 2
      end
```

Now we can classify the original test plan as follows. Retestable: y=0, y=1, y=-1 and any x values; reusable: y=2 and any x values. In order to satisfy the structural testing strategy, we need to have test cases which include paths that traverse both branches of instruction 6.2. Thus, a new-structural test case y=-2 and x=5 must be added to the test plan. There is no obsolete test case because the original structural test case is the only one which traverses one branch of instruction 6.2.

Recently, Benedusi, et al [1] described a retesting strategy, based on path testing strategies, which may be used to identify the above test classes.

2.5. Regression Testing Philosophy

We believe that the regression testing process should follow the same procedure as the testing process and the same standard should be applied to both processes. In particular, if the testing process requires storing certain dynamic information, then the regression testing process should also store that information. If the testing process is required to satisfy a certain coverage criterion, then the regression testing process should strive to achieve the same coverage measure. If the test cases are applied in a particular order during the testing process, we should also try to apply the test cases in a similar order during regression testing. It is assumed that all specification-based tests are applied before any structural-based tests. We will call the order of applying the test cases the *test-case-order*. In most cases, the ordering among the specification-based test cases are not important since they are designed to test the implementation with respect to the specification. However, the ordering of the structural-based tests may represent a way of achieving the structural coverage using a small number of test cases. Thus, some important testing information is stored implicitly in the order of applying the test cases and this information should be captured in the test plan.

The following definitions will be used in our discussion. A *redundant* test case is a structural test case in the test plan which covers the same program components as those covered by a disjoint group of tests. An *exclusive* test plan is a test plan with no *redundant* test cases.

If the testing process achieves several testing objectives, the regression testing process should also try to achieve similar objectives. In particular, if an objective of the testing process is to produce an *exclusive* test plan, then the regression testing process should also do the same. An exclusive test plan T is *minimum* if there does not exist another exclusive test plan T' with |T'| < |T|. Finding a minimum test plan can be shown to be a NP-complete problem. Fischer's method may be adapted to compute a minimum test plan [3]. We have developed an O(mn) algorithm for computing an exclusive test plan, where n is the number of test cases and m the number of program components which can represent branches, instructions, or define-use chains. Our algorithm is based on the execution count vectors and the set of test cases which are kept has the property that each traverses at least one unique component. Observe that the order of applying the test cases may render some test cases in the test plan redundant. Test case A in the test plan may become redundant after test case B is executed and added to the test plan. If test case B were to be executed before A, then when A is executed and makes no contribution to the coverage measure, it will not be added to the test plan.

2.6. The Notion of Regression Testable

A program is *regression testable* if most single statement modifications to the program entail rerunning a small proportion of test cases in the current test plan. Observe that the property of regression testability is actually a function of both the program and the test plan. Both the design of the program and the design of the test plan affect the amount of retesting for any program modification. A well-designed program usually requires less effort in testing and regression testing. However, if the test design is poorly done, then a well-designed program is no guarantee for easy testing and retesting. Before giving a measure for regression testable, we first define several regression testing metrics for the program and the test plan.

2.6.1. Testing Set, Testing Number, and Testable Program

In this section, we define the notion of a *testable program*. Before defining a testable program, we first introduce the concepts of testing set and testing number, which will be used in the definition of testable. Given an instruction I in a program P, there are two sets of paths which concern I: a set of paths where I occurs and a set where I does not occur. Due to the presence of loops, there may be an infinite number of paths which may traverse I. A compromise is to treat all interior paths as the same path [7].

Let P represent a set of paths $\{P_1, P_2, ..., P_n\}$, where n is the total number of paths. Let the *traversed set* T(I) denote the set of paths which may traverse I. Since it is not decidable whether or not a given path is feasible, T(I) may actually include some infeasible paths. Now, if I is a conditional instruction, then every path in T(I) is "affected" by I in the sense that the Boolean outcome of I indirectly affects the path computation. But if I is a non-conditional instruction (e.g., an assignment instruction), then some paths in T(I) may not use the definition at I. Let the *non-testing set* N(I) denote the set of paths which does not use the definition in I. Then the *testing set* R(I) = T(I) - N(I) is a set of paths which is affected by I, and executing these paths actually test the computation at I. One can determine the testing set of each instruction by a static analysis of the data flow and control flow of the program. Because the effect of I is limited to its testing set of paths, in the case that I is modified, we can restrict our analysis to R(I) in testing and measuring the effect of the modification. Let R(P) denote the *testing number* of the program P, $|R(I_i)|$ represent the cardinality of the set $R(I_i)$, and n denote the total number of instructions in P, then

$$R(P)=\sum_{i=1}^{n}|R(I_i)|/n$$

R(P) gives the average number of paths that are affected by a single instruction in P, assuming all instruction modifications are equally likely. The testing number can also serve as a measure of the number of paths affected by a single modification to the program.

The notion of testing set can be applied to other program components such as variables and procedures. For each program component C, there is a set of paths which directly or indirectly test C. All these paths belong to the testing set of C. If the paths in the testing set of C are executed and their results are correct, then our confidence about C's correctness will be increased.

Given the definition of testing number, we can now define *testable*. A program is *testable* if its testing number is small. The testable property measures the program design, independent of the test plan. A program can be untestable

due to poor design. A program P is said to be *more testable* than program Q if R(P) < R(Q). For a single modification, a more testable program requires an analysis of fewer paths on average than a less testable program.

2.6.2. Stable and Workable Test Plan

In the previous section, we introduced the notion of a testable program which is independent of the test plan. We next define two metrics that are directly computable from the test plan. A test plan is *stable* if it requires few deletions of old tests and additions of new tests for testing the new program version. A stable test plan is one which does not require many changes (replacement, addition, or deletion) to its test cases for testing the new version of the program.

A test plan is *workable* if it requires a small number of test cases to be executed for testing the new program version. There is a subtle difference between the metrics stable and workable. The stable metric measures the amount of *changes* in the test plan due to program modifications, while the workable metric measures the amount of *execution* or *testing effort* required in testing the program modifications. A test plan can be unworkable but stable.

The workable and stable metrics are actually indirectly dependent on the program. A poorly designed program usually involves difficult modifications that may entail extensive retesting. Observe that although a program may be testable, the design of the test plan may be badly constructed such that the test plan is unworkable and unstable. For a testable program, the workable and stable metrics are primarily measures of the test plan design alone. We will say a program is regression testable if it is testable and its test plan is workable.

We next introduce the notion of *regression number*, which may be used to determine the stability and workability of a test plan. For each instruction I in the program, there are two sets of test cases that may affect I. The first set of test cases causes execution to traverse I and the other set does not traverse I. We will call the first set of test cases the *affected test set* of I. Observe that according to the all-essential assumption, all these test cases should be rerun after I is changed. Let the *affected test number, a(T),* for a test plan T be the expected number of affected test cases for a single change. Let the *regression number,* $r_a(T)$, be defined as $r_a(T)=a(T)/n_T$, where n_T is the total number of test cases in the test plan. The affected test number and the regression number may be used to determine the stability and the workability of a test plan, respectively. The affected test number gives an estimate of the number of test cases affected by a change, while the regression number gives the proportion of test cases that are affected by any modification to a single instruction. Since a regression number is computed based on the information collected during testing, its value is dependent on the test cases in a test plan. A regression number can be any positive number above 0 and no more than 1. A program with a regression number close to 1 means that any statement modification will affect almost all test cases in a test plan. A program with a small regression number close to 0 means that only a small proportion of test cases are affected for any statement modification. We will call a test plan unworkable if its regression number is close to 1. More research is going on to find satisfactory criteria to better qualify the stable and workable metrics.

2.7. Regression Testing Assumptions

We summarize below several assumptions that will be made in our analysis.

Test plan requirement

1) The Test Plan includes a *testing guideline* which is a complete specification of the testing process giving the test design strategy, the coverage measure achieved, and the procedure for handling the obsolete test cases.
2) The Test Plan includes enough information for the identification of test classes. It should also store the test-case-order.

Changes information

We assume that the tester learns from the software maintainer the modifications made to a software product. In particular, he is given a list of modified functions, or modified statements, or both. Without this information, the tester would have to retest the whole program and no testing cost can be saved.

Characteristics of programs

The class of programs that will be considered should have the following features:

1) The program is single entry and single exit.

We assume that all input statements are located in the entry node and all output statements are located in the exit node of the program flow graph. We also assume that all initializations are done at the entry node. The above assumptions are made to simplify the analysis of the program.

2) The program is not too simple.

Each routine should have at least four paths. Any routine having fewer paths is likely to have simple logic and will not require many test cases. Thus, it will not be too costly to rerun all test cases during regression testing.

3) The program is not too complex.

Extremely complex routines are hard to test and may involve many test cases. Any routine which has a very complex structure should be reanalysed to reduce its complexity. A simple measure of complexity is the cyclomatic number [14].

4) The program is testable.

2.8. The Problem of Regression Testing

The problem of regression testing may be broken down into two subproblems: the *test selection problem* and the *test plan update problem*. The *test selection problem* is concerned with the design and selection of test cases to fully test a modified program. Some test cases may be selected from those in the existing test plan, while others have to be created based on modifications made to the program. The *test plan update problem* deals with the management of a test plan as a program is undergoing successive modifications. Certain

old test cases will become obsolete and new test cases must be added to test the modified and new features of the software. The test plan update problem may be defined as follows:

Test Plan Update Problem:

> Given a program P, and its specification S, a test plan T for testing P, a program P' which is a modified version of P, and its specification S', generate a test plan T' for P' from T, P, S, P' and S'.

From this definition, we can break down the test plan update problem into two subproblems:

(1) Test case classification problem

> Given a program P, and its specification S, a test plan T for testing P, a program P' which is a modified version of P, and its specification S', group the test cases in T into three mutually exclusive sets: reusable, retestable, and obsolete.

(2) Update problem

> Update the test plan by deleting the obsolete test cases, and adding in the new-structural and new-specification test cases.

Observe that before we can solve subproblem (2), we need a solution to the test selection problem.

3. The Retest Approach to Regression Testing

In this section, we describe an approach for corrective regression testing. The Retest approach tries to reuse the test plan in a way that will reduce the amount of testing. The previous test plan is used to assist in test selection, and is analysed in order to classify the various test cases; only a subset of the previous test plan is rerun. The Retest regression testing process consists of two major phases: the *test case classification phase* and the *test plan update phase* (see Figure 3). We next describe the major components of this regression testing process.

Static Analyser

This component is *static* in the sense that no test case is executed. All analysis is done using the change information and the information stored in the test plan. The static analyser uses this information to classify the existing test cases into two classes: *reusable* and *unclassified*. The reusable test cases are those tests which do not exercise any modified code, while the unclassified test cases are those which do. Another function of the static analyser is to compute the structural coverage measure achievable from the reusable test cases, based on the stored dynamic behavior of these test cases. Both the coverage measure and the unclassified test cases are passed to the Dynamic Analyser.

Dynamic Analyser

This component is *dynamic* because it actually executes some test cases in order to

1) test the program,
2) satisfy the structural coverage criterion, and
3) classify the unclassified tests into *obsolete* and *retestable* test classes.

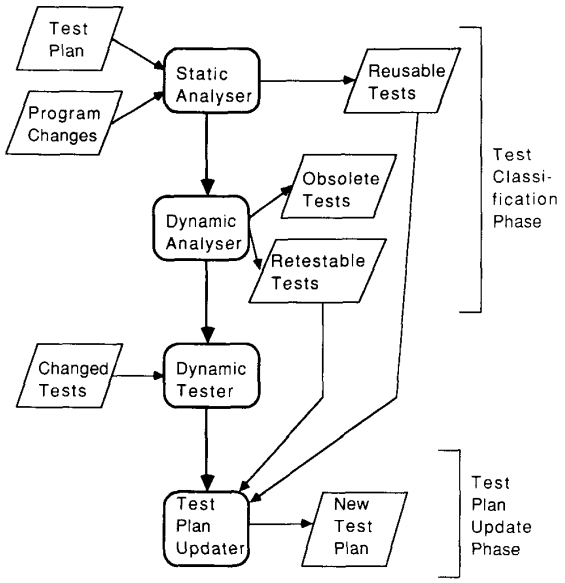

Figure 3. The Retest regression testing process for Corrective Regression Testing

The Dynamic Analyser executes each test which is unclassified in an order implied by the test-case-order. These test cases will be executed according to their ordering within the test-case-order. Note that all specification-based tests will be run first and put in the retestable class. They will not become obsolete because the specification is assumed to be unchanged. Any structural test which does not increase the structural coverage measure will be put into the obsolete test class. All others are grouped into the retestable test class. The Dynamic Analyser will stop test case execution whenever the structural coverage criterion is satisfied. Any remaining unclassified tests may be placed into the obsolete class, or handled according to the testing guideline specified in the test plan.

An output of the Dynamic Analyser is the coverage measure achieved by both the reusable and the retestable test cases. If this coverage measure is not satisfactory, then the Dynamic Tester is invoked; otherwise the testing process enters the test plan update phase.

Dynamic Tester

This component is similar to the Dynamic Analyser and is invoked only when the structural coverage criterion is not satisfied by both the reusable and retestable test cases. A major function of this component is to execute the new-structural test cases which are designed to exercise the modified code or to satisfy the structural coverage criterion. Only the test cases which will increase the structural coverage measure will be stored. Once the structural coverage criterion is satisfied, the Test Plan Updater is invoked.

A major difference between the Dynamic Analyser and the Dynamic Tester is that the former involves the changing and deleting of existing information in the test plan, while the latter involves the execution of new tests and the addition of new information to the test plan.

There are two advantages in running the unclassified tests before running the new-structural test cases:

1) some unclassified tests will test the modified code and therefore they can reduce the effort in designing new tests, and
2) some unclassified tests may be used to assist in test generation.

Test Plan Updater

This component creates a new test plan from the reusable, retestable and new-structural test cases. The objective is to generate an up-to-date test plan for the next cycle of modifications and regression testing. Most of the information stored in the new test plan have actually been collected by the Dynamic Analyser and the Dynamic Tester.

An operational overview of the Retest process is presented in Figure 4.

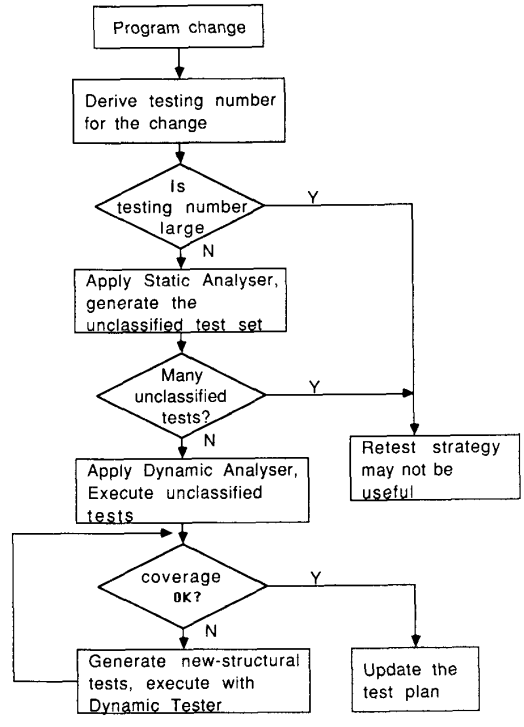

Figure 4. An operational overview of Retest

4. Conclusion

Although regression testing is an important topic, a fundamental study of the issues involved is long overdue. We have carried out an analysis of the problem of regression testing. We have identified two types of regression testing: corrective regression testing and progressive regression testing. The key difference between the two is that the specification stays the same in corrective regression testing, whereas progressive regression testing involves a modified specification. We have argued that corrective regression testing, in general, should be an easier process than progressive regression testing because more test cases can be reused.

The test cases can be grouped into five classes: reusable, retestable, obsolete, new-structural and new-specification test cases. It seems that a way to reduce the retesting effort is to reuse some test cases in the current test plan. This entails the identification of various test classes. We have identified the test classes associated with the two types of regression testing. Corrective regression testing may involve reusable, testable, obsolete and new-structural test classes, while progressive regression testing may involve all five test classes.

The notion of regression testable is introduced to measure the ease of retesting a program. This notion leads us to postulate that both the program design and the test plan design affect the ease of retesting. A poorly designed program will lead to difficult testing and retesting. But a well-designed program does not guarantee easy testing and retesting if the test cases are not well-selected. To capture all these ideas, we introduce the testing set, the testing number, and the stable and workable metrics. The regression number is introduced as a measure of the effect of a program modification on the test plan. We also introduce the Retest strategy for corrective regression testing. This strategy views the regression testing problem as composed of two subproblems: the test case selection problem and the test plan update problem, and structures the testing process into two phases: the test classification phase and the test plan update phase.

Observe that software systems are not subjected to a series of modifications only during maintenance phase. In many ways, the modifications occurring during the testing phase are similar to those of the maintenance phase. There are big modifications which involve specification changes with major restructuring of the control and data flow of the program, and there are minor modifications which do not alter the specification. In the latter case, the Retest strategy can be applied during the testing phase. The only requirement is that the test plan satisfies the regression testing assumptions.

We should add a word about the use of regression testing to programming-in-the-large. In this paper, we have concentrated on the application of regression testing to modules and unit testing. As this paper is being rewritten for these proceedings, we are investigating the application of these regression techniques to integration testing. In both of these instances of programming-in-the-large, our notions of regression testing apply, despite the fact that the computer program and the quantity of test data may be very large. If substantial test data also exists at the system level, then our regression testing approach may not be reasonable because the size of the data may make some of the processing or data structures

suggested in this paper impractical. However, if the program design decomposes the problem so that sufficient testing is done at the module and integration levels, then the amount of test data at the systems level may not be excessive, and our regression testing approach can be applied at that level as well.

Several other major problems remain to be addressed in regression testing. For example, test selection is an interesting problem which demands attention. It seems that many existing testing methods may be used. Most of our analysis and proposed solutions have focused on corrective regression testing. Progressive regression testing presents a different set of problems which have not yet been analysed. Here the specification is modified and the situation is similar to testing a new program. We suspect that many existing testing strategies can be applied. The key step is in identifying the parts of the program that should be retested.

References

1. P. Benedusi, A. Cimitile, and U. De Carlini, "Post-maintenance testing based on path change analysis," *Proc. Conf. Software Maintenance*, pp. 352-361, 1988.
2. J. S. Collofello and J. J. Buck, "Software quality assurance for maintenance," *IEEE Software*, pp. 46-51, Sept. 1987.
3. K. F. Fischer, "A test case selection method for the validation of software maintenance," *Proc. COMPSAC 77*, pp. 421-426, Nov. 1977.
4. R. L. Glass, "Persistent software errors," *IEEE Trans. Software Eng.*, vol. SE-7 (2), pp. 162-168, Mar. 1981.
5. M. J. Harrold and M. L. Soffa, "An incremental approach to unit testing during maintenance," *Proc. Conf. Software Maintenance*, pp. 362-367, 1988.
6. W. E. Howden, "Applicability of software validation techniques to scientific programs," *ACM Trans. Program. Lang. Syst.*, vol. 2 (3), pp. 357-370, July 1980.
7. W. E. Howden, "Methodology for the generation of program test data," *IEEE Trans. Comput.*, vol. C-24, no 5, pp. 554-559, May 1975.
8. W. E. Howden, "A functional approach to program testing and analysis," *IEEE Trans. Software Eng.*, vol. SE-12 (10), pp. 997-1005, Oct. 1986.
9. B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, pp. 155-163, Oct. 1988.
10. J. W. Laski and B. Korel, "A data flow oriented program testing strategy," *IEEE Trans. Software Eng.*, vol. SE-9(3), pp. 347-354, 1983.
11. H. K. N. Leung and L. White, "A study of regression testing," *Technical Report, TR-88-15, Dept. of Comp. Sc., Univ. of Alberta, Canada,*, Sept. 1988.
12. B. P. Lientz and E. B. Swanson, in *Software Maintenance Management*, Addison-Wesley, 1980.
13. Guideline on Software Maintenance, in *Federal Information Processing Standards*, U.S. Dep. Commerce/National Bureau of Standards, Standard FIPS PUB 106, June 1984.
14. T. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. 2, 4, pp. 308-320, Dec. 1976.
15. S. Ntafos, "On required element testing," *IEEE Trans. Software Eng.*, vol. SE-10 (6), pp. 795-803, 1984.
16. S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Trans. Software Eng.*, vol. SE-11 (4), pp. 367-375, 1985.
17. H. G. Stuebing, "A modern facility for software production and maintenance," *Proc. COMPSAC 80*, pp. 407-418, 1980.
18. L. J. White and E. I. Cohen, "A domain strategy for computer program testing," *IEEE Trans. Software Eng.*, vol. SE-6(3), pp. 247-257, 1980.
19. S. S. Yau and Z. Kishimoto, "A method for revalidating modified programs in the maintenance phase," *Proc. COMPSAC 87*, pp. 272-277, 1987.