Software maintenance requires changing programs as a result of errors, or alterations in the user requirements. However, during such modifications, new errors may be introduced causing unintended, adverse side effects in the software. To combat such occurrences, regression testing is utilized in order to revalidate the modified programs. Ideally, the existing baseline set of test cases should be executed every time in order to ensure conformance with the specifications. As a result, this particular testing phase can prove to be expensive in terms of both human and computational resources. The paper examines the current problems associated with regression testing, in particular aspects concerning selective revalidation. It describes a prototype retesting environment being developed at the University of Durham, which will allow maintenance personnel to cope with revalidation procedures.

I. Introduction

Software maintenance is normally subdivided into four activities, namely adaptive, corrective, perfective and preventive maintenance[1]. Adaptive maintenance is the modification of a system, because of some change to the external environment in which it operates. For instance, there may be an alteration to the host machine’s operating system which requires a modification in the maintained system. Corrective maintenance is the alteration of the system to correct an error that has been detected. Perfective maintenance is the modification of a system to enhance its functionality. Finally, preventive maintenance is the alteration of a system to ease future maintenance. For example, a routine which has become excessively complicated may be redesigned to make it easier to understand in the future.

Throughout the software lifecycle, testing strategies can be classified as being either structural or functional in nature. The former strategy employs knowledge of the construction of the program. As an example of a structural strategy, consider the approach of ensuring that the test data set executes all the outcomes of all the branches within a system under test. This requires access to the source code to validate the strategy. A functional test strategy uses the requirements of the system as the basis for deriving the test data. It has been shown that a combination of structural and functional strategies is the most desirable[2]. After any adaptive, corrective, perfective or preventive maintenance activities, it is likely that the program structure will have been altered.

The alteration of the program structure will imply that new structural tests will be required and some of the previous tests may no longer be relevant and so should be eliminated from the test suite. New functional tests will be necessary for perfective and corrective modifications. Although the requirements will not have changed after a corrective modification, the test case that revealed the error will now need to form part of the test suite. Adaptive and preventive modifications imply no changes to the requirements and so new functional tests are unnecessary.

In general, the issue of revalidation during the software maintenance phase appears to have been neglected for many years [2]. In order to accurately retest both the existing functionalities and any system enhancements, and to reestablish confidence in the software’s ability to perform according to the specification, it is essential to employ some form of regression testing[3].

Two types of regression testing have been identified, namely progressive and corrective regression testing[4]. The former involves testing major changes to the specification, so whenever enhancements, or new data requirements are incorporated into a system, the specification will be altered to reflect the modifications. In general, such alterations are expressed in terms of new modules or routines being added, and perhaps old ones being eliminated or altered. On the other hand, corrective regression testing is performed on a specification that remains unchanged. Typically, progressive regression testing is performed after adaptive or perfective maintenance. Therefore it is usually employed at regular, fixed intervals, whilst the corrective retesting strategy is undertaken after corrective maintenance activities, that can occur at any time and may be invoked at irregular intervals, as after every correction.

Problems associated with regression testing can be divided into two categories:

(a) The test update problem is concerned with the maintenance of the test suite. After modifications to a system, some of the test cases, whether functional or structural in nature, will no longer be relevant and need to be eliminated. However, their identification is not always straightforward.
(b) The test selection problem involves determining which test cases should be rerun after a particular modification is undertaken, as the execution of an entire baseline set of test cases to validate small numbers of modifications may consume a large amount of time and computational resources. Furthermore, an intuitive or random approach to the selection of suitable test cases to exercise the main system facilities is simply insufficient. Therefore, it would be highly desirable to establish a systematic and sensible approach to rectify this situation.

Thus, this paper will present an overview of current selective retesting strategies, and will focus on the test selection problem. In particular, it will describe one of the retesting strategies discussed below in greater detail, explaining the extensions and new ideas which have led to the development of a retesting prototype environment for use in software maintenance.

2. An Overview of Current Retesting Strategies

In the past, few publications have appeared which exclusively address selective revalidation strategies. Essentially, the three strategies discussed here are based on criteria which attempt to ensure that all modified program statements have been exercised, as well as those program sections that are directly, or indirectly addressed by the modified code and which share any data dependencies with it.

In 1977 and 1981, Fischer [5,6] developed a selective retest model and test criteria that were based upon adaptations of mathematical optimisation algorithms[7]. His 0-1 integer programming model for retesting identified the degree of connectivity, reachability and test coverage of program statements, and considered the data dependencies between them.

In 1987, Yau and Kishimoto[8] presented a revalidation methodology utilising concepts such as input partitioning and cause-effect graphing of the program specification, symbolic execution trees and test information tables for selective determination of test data, and data-driven symbolic execution for final test execution and debugging. A more detailed explanation and criticism of this technique may be found in [9].

Recently, an alternative approach has been proposed by Leung and White[4]. They monitor only the changes performed since the last analysis. Structural tests which execute the changed structures, or the routine containing those structures are then subject to reexecution and analysis. This reveals whether the old tests which executed the modified section of the program are now obsolete. However, the automatic identification of obsolete functional tests using this approach would only be feasible if the specification was formal and could be analysed to observe the changes from the old specification. Suggestions were also made for the use of a function table, listing all the functions together with the numbers of the functional test cases which executed those functions. This would enable the manual identification of obsolete functional test cases. Finally, their paper outlines different data structures and operations which attempt to support the maintenance of the test suite.

These methods attempt to discover which segments have been affected by a modification to a particular segment. It is not possible to discover all the segments that have been affected, because the modification may have had a semantic affect. For instance, if a modification was made to a segment to write to a particular file which was read be a later segment, then it would not be possible to discover this connection by an analysis of the source code.

3. Fischer's Selective Revalidation Strategy

Fischer discusses a retesting methodology based on the concept of 0-1 integer programming models, in which the connectivity, reachability and test coverage of program statements, and their data dependencies are identified. Furthermore, he describes program segments with reference to FORTRAN, defining them as 'a contiguous sequence of executable statements with one entry point at the beginning and one exit point at the end'. To monitor the ripple effect[10], the overall retest strategy includes two algorithms that distinguish those segments, that are affected by the data conditions used or set in a modified segment, and which reach to and from that segment, respectively.

3.1 The Zero-One Integer Programming Model

Fischer's retesting model is based on a objective function Z, and a series of inequalities as shown below by Equation 1. The minimising function Z is given by:

\[ Z = c_1x_1 + c_2x_2 + \ldots + c_nx_n \]

which is subject to the following constraints:

\[ a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \geq b_1 \]
\[ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \geq b_2 \]
\[ \vdots \]
\[ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \geq b_m \]

Equation 1

This equation can also be represented in matrix form with a matrix of coefficients \( a_{ij} \), and two vectors \( b_i \) and \( x_j \).

a) The objective function \( Z \) represents the optimal, or minimal number of test cases required to be rerun after a modification is made to a particular program segment.
b) The coefficient \( c_i \) represents a cost element of the objective function \( Z \). In this model, the cost elements are set to \( z'z' \), as it is assumed that all test cases have an equal cost to set up and run. In future, one hopes to vary this factor in order to further investigate the influences of test case costs on the revalidation process.
c) The variable \( a_{ij} \) is given as an element of the constraint coefficients.
d) All values indicated by the \( b_i \) elements represent the lower bound of the constraint row/column i.
e) The variable \( x_j \) represents the test case existence variable, which depending on whether its value is a '0' or '1', determines whether or not a particular test case exercises a specific program segment. For example, where \( x_j=1 \), the corresponding jth test case needs to be rerun; otherwise \( x_j=0 \) in which case, the particular test does not require repeated execution.
3.2 Considering Four Different Types of Matrices

Initially, Fischer's algorithm relies on a general flow analysis of the target program, in order to accumulate information concerning the connectivity, reachability and test coverage of segments and their variable usage. This data is represented by four matrices including the connectivity, reachability, test case dependency or cross reference matrix, and the variable set/use matrix, respectively.

The connectivity matrix reflects the program's control flow by indicating any direct interconnections between the different segments. It is represented in the form of a square matrix, containing \( m^2 \) elements, in which a '1' entry shows that the control flow of the program directly transfers between two particular segments, whilst a '0' entry shows otherwise.

The reachability matrix describes both the direct and indirect interconnections between the various program segments. It can be automatically generated from a connectivity matrix by applying Warshall's Algorithm[11]. Fischer also suggests a slight alteration to this algorithm, resulting in all diagonal elements of the reachability matrix being set to a '1', to ensure that whenever a particular segment is modified, it is actually included in the retest procedure. The reachability matrix is of the same order \( (m^2) \), as the connectivity matrix. A '1' entry indicates whether a particular segment can be reached either directly or indirectly from another, a '0' suggesting otherwise.

The test case dependency matrix reflects the test case coverage of the individual program segments, by placing a '1' entry against segments that have been exercised by a given test case, a '0' stating otherwise.

Furthermore, the variable set/use matrix is used by the retest algorithm to monitor the ripple effect[10] during maintenance modifications. It records the usage of variables in the different program segments, by placing an 'S', 'U', or 'X' in the matrix to represent a variable being either set, used or set and used, respectively.

3.3 Outlining the Fischer Retest Algorithm

After determining which particular program segments have been modified, a number of constraint expressions can be derived from Equation 1.1, in which each expression corresponds to a row of the test case dependency matrix. These expressions are incorporated into the O-1 integer programming model on the left hand side of Equation 1.1, whilst those values \( b_h \) on the right hand side are established as a result of logical ORing operations performed between the respective rows and columns of the reachability matrix associated with each modified program segment. The model can be further simplified by several reduction methods [5,7] and any redundant constraints removed, resulting in an optimal value for the objective function \( Z \). The value of this function represents the number of test cases that need to be run in order to validate the given modification. Those test case existence variables which remain, and contain a value of '1', have to be executed again.

In order to analyse the data dependencies between the various program segments and to further reduce the number of selected, reusable test cases, Fischer developed two additional algorithms which should be regarded as part of the overall testing methodology.

The first algorithm, simply known as Algorithm A, identifies those program segments, which could affect the data conditions used in the modified segment, and which reach to this particular segment; the emphasis being on the usage of variables in the modified segment. In contrast, the second algorithm, called Algorithm B, determines those program segments, which could be affected by the data conditions set in the modified segment, and which are reached from the particular segment; the emphasis in this case being on the setting, and setting and usage of variables in the modified segment. The results of both Algorithm A and B are then logically ANDed with the column and row of each modified program segment, producing a set of values, which are utilised on the right hand side of Equation 1.1, instead of the previously determined set of \( b_h \) values. The model may then be reduced again, and solved to determine the minimal value of the objective function \( Z \).

To identify whether a particular variable in a program segment is set \( (S) \), set and used \( (X) \), or used \( (U) \), a set of rules was defined by Fischer. These suggest that (i) a variable is set, if its value has been changed, and thus an 'S' may be entered in the set/use matrix, (ii) a variable is used, if its value has been accessed and used, and thus a 'U' may be entered in the set/use matrix, and (iii) a variable is set and used, if its value has been set and used in a statement and thus an 'X' may be entered into the set/use matrix.

Fischer's methodology has been defined only for the FORTRAN language. The major limitation is its incapability of retesting modifications which have affected the program control structure. Furthermore, it assumes that the initial baseline set of test cases correctly reflects the user specifications and does not check whether a minimum test plan has been presented[4]. A minimum test plan is one in which there are no redundant test cases, a redundant test case being a test case in the test plan, which covers the same program segments as those covered by a disjoint group of tests. Furthermore, Fischer's publications only discuss and demonstrate the algorithm for segments within a program routine, in which the modifications are constrained to changes within a particular segment and he makes only very tentative approaches in the application of his technique outside the scope of the example routine.

However, the advantages that accrue from the Fischer strategy appear favourable when comparing the complexity and cost factors involved in other available retesting algorithms. The retest strategy described earlier[8], required the use of cause-effect graphing and symbolic execution in order to revalidate the modifications, whereas the Fischer methodology, although currently less sophisticated, relies on the establishment of the four matrices containing the connectivity, reachability, test case coverage and set/use information, together with an efficient optimisation algorithm, to compute similar results.

4. Extensions to the Fischer Algorithm

Our work at the University of Durham is aimed at extending and enhancing the original Fischer material, by overcoming some of the limitations discussed earlier, and including a number of new ideas which could result in the development of a practical, selective retesting prototype environment running on the UNIX operating system. Particular issues which are being addressed include:
a) Solving the problems associated with progressive regression testing, that is, the revalidation of source code that has been substantially modified to the extent that the control flow may have been changed, new data structures included, or perhaps old ones deleted. Although the current implementation can only be used for corrective regression testing purposes, it is hoped that the prototype will soon be extended to include progressive regression testing.

b) Extending the algorithm to allow the selective revalidation of an entire program which may consist of a separately compiled structure with different header files, modules, routines, or packages, such as those commonly employed with modern high level languages such as C, Ada or Modula-2. Our current prototype will analyse C source programs, but our ideas could easily be extended to encompass other high level programming languages, such as Pascal.

c) The definition of segments for the C programming language, and the development of a ripple effect analyser capable of monitoring the use of all types of variables, data structures, and parameters throughout a program. At the moment, the parsing tool employed, simply checks the usage of global variables and data structures, parameters and local variables associated with these parameters, and arrays.

d) The automation of the selective revalidation process, that would enable personnel to perform different types of maintenance, and automatically receive feedback from the tool as to the consequences of testing these subsequent changes. Such a prototype would provide the user with status information concerning the currently active set of tests, such as:

- Highlighting of redundant structural or functional test cases;
- Indicating whether full segment or routine test coverage has been achieved;
- Suggesting any additional test cases that may be required to exercise the enhancements or new data structures in order to achieve full test coverage.

In the case of progressive regression testing, the final retesting environment will have to reanalyse the target system and extract any reusable test cases from the existing test library. For corrective regression testing purposes, it must able to determine the subset of test data to revalidate the smaller changes. In either case, the environment will update the relevant data structures in order to keep in step with the maintenance activities.

5. Development of the Selective Retesting Environment

Although the Fischer methodology discussed revalidation at the statement level, our current developments have taken a 'higher level' approach to this issue, by regarding a program routine or function, instead of a series of statements, as a segment. Thus, two distinct models, known as the segment and function level models, have evolved.

As our prototype environment is implementing the latter model, it is less sophisticated in terms of its test coverage, as it does not check whether a sequence of statements is exercised, but only monitors routine coverage. Furthermore, the operation of the functional level revalidation process requires less computational overheads in:

(a) Statically parsing the source code for the different types of variables, parameters and data structures used,
(b) Statically analysing the program for user-defined function definitions and calls, instead of segments,
(c) Performing the dynamic analysis for functions and not segments, throughout the target program.

Thus, the size and complexity of the resulting matrices, such as the set/use, reachability and test case dependency matrices are significantly reduced. Instead of a set/use matrix being established for segments in each routine, there exists only one matrix, containing information on the usage of all global variables and data structures, parameters and local variables associated with these parameters, and arrays. Similarly, the reachability and test case dependency matrices record the function interconnections and test coverage.

By developing such a selective retesting environment, significant reductions in the computational requirements, and improvements in both test and cost effectiveness during the revalidation stage are expected. If the prototype is firstly evaluated at a functional level and has been proven in practice, then further work will see the introduction of a more sophisticated version in which the target program is properly segmented at the statement level, with all types of variable usage being monitored, and the test coverage referring to each program segment. In either case, the prototype's success will be judged by comparison with more commonly practised regression testing techniques, namely the rerunning of an entire baseline set of test cases, and the execution of a set of randomly/intuitively selected test data.

5.1 An Overview of the Prototype Environment

Figure 1 illustrates the overall architecture of the selective revalidation environment. The prototype is capable of analysing C programs with a separately compiled structure. Essentially, it can be divided into five distinct tools, which include:

(a) A Simplified C Preprocessor Tool, which will automatically expand all constants and macros defined in the source program and include all user-defined header files.

(b) A Function Extraction Tool, which extracts only the names of user-defined C functions from the source code, and places them into an intermediate data file. This file is subsequently used by the other tools listed below.

(c) A Reachability Analysis Tool, which parses the C source code for the user-defined function names, supplied by the Extraction Tool, and their definitions, to determine any direct connections that may exist between the calls in the context of the source program. The corresponding values for the reachability matrix are then computed using Warshall's Algorithm for transitive closure. Again the results are stored in an associated data file.

(d) A Dynamic Analysis Tool, which inserts 'probes' at each entry point to the user-defined functions, and automatically recompiles the source code, ready for test case execution.
(e) A Test Coverage Tool, which captures and analyses all output (trace files) produced by the execution of the current library of test cases, establishing a corresponding functional test case coverage matrix. The tool will not only create a corresponding data file, but will also inform the user of any redundant test cases, indicate whether full test coverage has been attained, and make suggestions for extra test cases that need to be developed to achieve full coverage.

(f) A C Parsing Tool, which analyses the source code to establish the set/use matrix. Its purpose is to monitor the use of global variables and data structures, parameters and local variables associated with these parameters, and arrays, placing its results into a data file. The parser attempts to comply with the ANSI C draft proposals.

(g) A Selective Revalidation Tool, which analyses all information supplied by the above tools and establishes a linear optimisation model. By solving the set of constraints it computes the minimum number of test cases required to be rerun, as well as the possible sets of test cases which can be executed in order to validate the changes. Finally, all results are directed to both standard output and a user-specified file, for off-line viewing.

5.2 Determining Reachability Among Functions

The Reachability Tool performs three distinct operations. Its first task includes a static analysis of the C source code, resulting in a dynamically linked list being created, in which all user-defined function names and the type of C construct they were parsed in, are recorded. The tool then attempts to pass through the list introducing links which represent the connectivity between functions. After completing this task, the list is then traversed once more to calculate the corresponding values in the reachability matrix. Figure 2 illustrates the connectivity that is established between user-defined functions as they are parsed in different C constructs. Furthermore, Figure 3 also provides a small, pseudo C program to show how the dynamic list evolves, together with the C data structure that represents each node in the list. The darker arrows shown in the linked list represent the function connectivity, whilst the lighter ones are used for general list traversal purposes by the underlying algorithm.

5.3 Determining the Set/Use Matrix Information

In order to monitor the ripple effect, information is gathered, concerning the usage of different variables and data structures throughout a C source program. This required the development of a complete parser. The Parsing Tool used in the prototype was developed using the UNIX tools, LEX and YACC. The ANSI C draft proposals were translated into a suitable Backus-Naur Form notation and compiled together with a number of supporting C functions. These underlying routines were triggered during the parsing process and produced the corresponding set/use matrix.
Figure 2: Connectivity Between Function Calls In Different C Language Constructs

Figure 3: An Example of the Construction of the Dynamic List and Its Structure
The main distinction is that in declare the parameter to be a pointer and access the variable than in originals, that is, memory addresses. This leads to in Pascal, in which its private, temporary copy. Call by value usually leads to high level languages like FORTRAN given the values of its arguments in temporary variables rather than in originals, that is, memory addresses. This leads to some different properties than are seen with call by reference. High level languages like FORTRAN or with var parameters in Pascal, in which the called routine has access to the original argument, not a local copy.

The main distinction is that in C, the called function cannot directly alter a variable in the calling function; it can only alter its private, temporary copy. Call by value usually leads to more compact programs with fewer extraneous variables, because parameters can be treated as conveniently initialised local variables in the called routine. However, it is possible for a function to modify a variable in a calling routine. The caller must provide the address (&) of the variable to be set, that is, technically a pointer to the variable. The called function must declare the parameter to be a pointer and access the variable indirectly through it.

Furthermore, when an array name appears as an argument to a function, the value passed to the function is the location or address of the beginning of the array - there is no copying of any array elements. The function can access and alter elements of the array by subscripting from this location.

Thus, when the parser encounters either global variables, data structures or function calls, it determines whether a particular variable has been set, used, or set and used. Should any of these variables be accessed more than once in a particular function definition, then a highest common factor rule applies, in which the set and used term possesses the highest priority. For example, if a variable A was used and set in two distinct operations within a Function D, then the overall value for variable A in the set/use matrix would be set and used.

As the parser proceeds with its analysis of the source code, any function parameters that have been called by reference are omitted for the time being, until the associated function definitions are encountered and evaluated. Only then, does the parser decide upon the values which are entered in the set/use matrix.

Thus, Figures 4 and 5 try to portray the different types of variables and parameters which need to be monitored by the parsing tool. Each arrow indicates a C source code statement that requires special action by the parser.

5.4 Modifications to Program Routines

Figure 6 illustrates the way in which the selective revalidation environment copes with changes being made to a program routine. The method described here can be extended to handle the modification of several routines at any one time, but for reason of clarity, only one function is modified.

For example, if Function A was to have been altered in some way, the selective revalidation tool would determine which subset of test cases needs to be reexecuted. All columns corresponding to these tests in the Test Case Dependency Matrix are reset, the tests executed and the subsequent test coverage monitored. The coverage status would then be displayed. Next, both the Parsing and Reachability Tools would reanalyse Function A, updating both the program set/use matrix and reachability matrix, if necessary.

The advantages of such an environment are that it reduces the amount of analysis to a minimum, rapidly analysing any given changes, computing and executing the required subset of test cases, providing suggestions for any additional tests, if necessary, and updating the corresponding data structures. Therefore, it provides an ideal platform for initial use in the area of corrective regression testing, with a view to extending it to cover progressive regression testing.
6. Conclusion and Future Research
Software maintenance operations require some form of regression testing in order to accurately retest both the existing functionalities and enhancements, and to reestablish confidence in the software’s ability to perform according to the specification. However, large amounts of time and computational resources are expended for revalidation, and the maintainers' view of this topic often lacks a sensible and systematic approach.

The development of the RETEST environment is a first step towards approaching the subject of selective revalidation and its automation. It appears that the RETEST environment may soon be linked with an established regression testing tool, known as ASSAY[3]. Further tools such as complexity analysers and comment extractors[12] can easily be added to enhance it and provide a basis for future research into the field of software maintenance and regression testing. RETEST is currently under development.

This paper has presented an overview of current selective retesting strategies, and focussed on the test selection problem. In particular, it described one of the retesting strategies in greater detail, explaining the extensions and new ideas which have lead to the development of a practical retesting prototype environment for use in software maintenance.

7. Acknowledgements
Jean Hartmann is sponsored by British Telecom Research Laboratories, Martlesham Heath, Ipswich, IP5 7RE. U.K. and would like to thank them for their support.

8. References


