A Study of Integration Testing and Software Regression at the Integration Level

Hareton K. N. Leung*

Test Strategy Development
Bell-Northern Research Ltd.
Ottawa, Ontario, Canada

Lee White

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

ABSTRACT

Integration testing is an important phase of the testing process which has not been thoroughly studied. Although integration strategies are well-understood, the test selection problem has not been systematically studied by the testing research community. In this paper, we identify the common errors and faults in combining modules into a working unit. We also make practical recommendations on test selection for integration testing, and utilize those recommendations for regression testing at the integration level. With emphasis on reusing the previous test cases and retesting only the parts that are modified, one can reduce the testing expenses. The concept of "firewall" is proposed to assist the tester in focusing on that part of the system where new errors may have been introduced by a correction or a design change. An experiment is presented in which an application of our strategy is compared to the 'retest-all' strategy. Applying our approach for test selection we were able to discover all errors found by the retest-all strategy by executing only 35% of the total number of test cases.

Keywords: software integration, integration testing, regression testing, software maintenance, test cases reuse, global variables, program errors, firewall.

1. Introduction

Integration testing is the testing applied when all the individual modules are combined to form a working program. Testing is done at the module level, rather than at the statement level as in unit testing. Integration testing emphasizes the interactions between modules and their interfaces. Although many integration strategies have been described, few give any guidelines for actually generating test cases. Myers describes six integration strategies: bottom-up, top-down, modified top-down, sandwich, modified sandwich and big-bang testing [20]. Carey and Bendick discuss build testing [4]. Beizer introduces a "mixed bag" strategy which combines bottom-up, top-down, big-bang and build testing [2]. None of the integration techniques focuses on selecting test cases. The proposed integration strategies can be grouped into two types: the *incremental* strategies merge one module at a time to the set of previously tested modules, while the *nonincremental* strategies group all the modules together simultaneously and test them. A recent study shows that approximately 40% of software errors can be traced to component interaction problems discovered during integration [1]. Most of these detected errors are due to misinterpretation of module specifications.

Many people have realized that the software system and its application will evolve as it is adapted to changing environment, changing needs, new concepts and new technologies. Software will grow in the number of functions, components and interfaces. Old modules may be expanded for uses beyond their original design. Thus, modification to the software is inevitable. Much of the time spent on software maintenance is in modifying and retesting the software. Efficient and effective regression testing can reduce the cost of maintenance.

Regression testing is a testing process which is applied after a program is modified. It involves testing the modified program with some test cases to re-establish our confidence that the program will perform according to the (possibly modified) specification. Rerunning all existing test cases after a software modification is usually costly due to program size and testing frequency, but this is currently a common practice. Regression testing can be made more efficient with strategies that emphasize partial retesting. We have identified two types of regression testing [15]: *progressive regression testing* involves a modified specification, while the specification does not change in *corrective regression testing*. In general, corrective regression testing should be an easier process than progressive regression testing because more test cases can be reused. Although regression testing is currently receiving more attention [8, 9, 25], most methods are restricted to testing at the unit level. In this paper, we also analyze the issues involved in regression testing at the integration level.

In the next section, we describe our testing model. Section 3 gives the common errors and faults that may occur in calling another module. A major mistake occurs when the calling module has an incorrect expectation of the called module. Integration testing objectives, which aim at detecting errors not found during unit testing, are described in Section 4.1. Section 4.2 focuses on the test selection problem for integration testing. In Section 5, we present the regression testing strategies for various basic types of modifications. The strategies emphasize reusing the previous analysis and test cases. Testing expenses are reduced because fewer tests and less analysis are needed. Section 6 describes the concept and construction of a *firewall* and shows three interesting properties of program modifications. In Section 7, we describe the application of our regression testing strategy to an actual software system and have obtained encouraging results. Finally problems for future research are identified in Section 8.

2. A Testing Model

According to our model, the testing process is structured into three phases: unit testing, integration testing and system testing. Each phase provides its own effectiveness with respect to detecting certain errors. In a cost-effective testing process, each testing phase should work in concert with the other testing phases. Each phase should concentrate on finding faults which cannot be easily detected by other phases. At the same time, for faults which can be detected by several phases, resources should not be expended in duplicating the same testing effort.

Our model assumes that each module is unit tested with functional and structural tests. We assume the functional testing has sufficient test cases to test the identified functions of the modules [10]. For structural testing, we assume an appropriate coverage measure is used and tests are selected so as to provide the required coverage [5, 14, 22].

After the completion of unit testing, the modules should be integrated together. During integration testing, we should concentrate on three different types of integration errors: *missing function, extra function* and *interface errors*. Not all of these errors may be present in a given software system, but an improperly designed software system will likely contain some of these errors. Section 3 describes these common errors when one module calls another module.

After the software is modified, the first step is to identify those modules whose implementations are modified and to apply unit testing to them. The regression unit testing may be divided into two major classes: *progressive regression testing* and *corrective regression testing*. For the case of corrective regression testing, a complete procedure is outlined in reference [15].

* Most of the work reported here was completed when the author was at the University of Alberta, Edmonton, Canada.

We will assume that the program consists of a number of modules, which are connected in the form of a *call graph.* A *call graph* shows the control hierarchy of the program with rectangles representing modules. An arrow from module A to module B indicates that module A may call module B. Module A is an *ancestor* of module B if there exists a path (a sequence of calls) in the call graph from module A to B; B will be called a *descendant* of A. Figure 1 provides an example of a call graph.

In order to provide an analysis of regression testing for the integration testing process, a number of assumptions are made which will be applied throughout the paper:

(1) The *change information* is correct; the change information indicates those modules and specifications which have been modified, and provides new descriptions; this does not imply that the modifications are correct. Moreover, when *specifications* are modified, we will aggressively regression test modified functions in these specifications, together with any other functions which may have inadvertently been affected. The only case where we will have to assume that a modified specification is correct occurs when the system specification (or the specification of the primary module in the call graph) is modified; in this case, there may be no other documentation or information available indicating this specification to be in error.

(2) The previous test cases are available at both the unit and integration levels for all modules; a test case consists of input data, the actual output values and a *trajectory.* A *trajectory* is a program path that has been executed by some input [13], and can be used to compute the structural coverage of the test set.

(3) For both regression unit tests and regression integration tests, actual modules are in place which are called directly or indirectly by the module under test. In early development, the tester might use "skeleton" or "stub" routines to return the information an incomplete module should provide and to simulate the behavior of that incomplete module. Since this is a study of the regression testing process, it can be assumed that all modules are now actually available.

(4) There is no recursion in the program and therefore the call graph will be a connected directed acyclic graph.

(5) In general, substantial effort may be required to test for errors caused by *global variables* or *common data areas;* since we are analyzing the call graph for the effects of local data flows through formal parameters passing between modules, we will assume that no global variables exist. In other words, there is no *common* or *external coupling* among modules; all modules are *data coupled [24].* The authors have recently addressed this problem of global variables separately for regression testing [16], which could be incorporated into the approach given in this paper.

(6) The use of *pointers* has to be restricted so as to avoid the same problem as with global variables in assumption (5). We can allow parameters to consist of *pointers,* and these pointers can address *external tables;* however, there must be a strict separation between *input tables,* which are read-only, and *output tables,* which are write-only. If an external table could be accessed by pointers from within a module, and could be used for both input and output functions, then this could lead to the same testing problems as those encountered with global variables. As shown in [16], a regression testing methodology can be developed for an unrestricted use of pointers, but would be beyond the scope of this paper.

2.1. Input Domain

We next describe our view of module inputs, which leads to the concept of *input domain.* Each input to a module M can be viewed as a single point in the k-dimensional *input space* Y, where $Y=Y_1 \times Y_2 \ldots \times Y_k$, and Y_i represents a single input variable, and where k is the total number of input variables. When the designer of a module M develops the design, a number of constraints on the input space must be taken into account. Thus the functionality of M is designed based upon this *subset* of the input space, called the *input domain* D(M). Input points from D(M) will allow M to achieve its desired functionality, whereas points from outside D(M) in the input space should result in an appropriate error message. For simplicity, we will indicate that only input points in D(M) will cause M to be executed.

It is important to associate practical program data structures with this theoretical view of input domain. The input domain D(M) for module M can consist of:

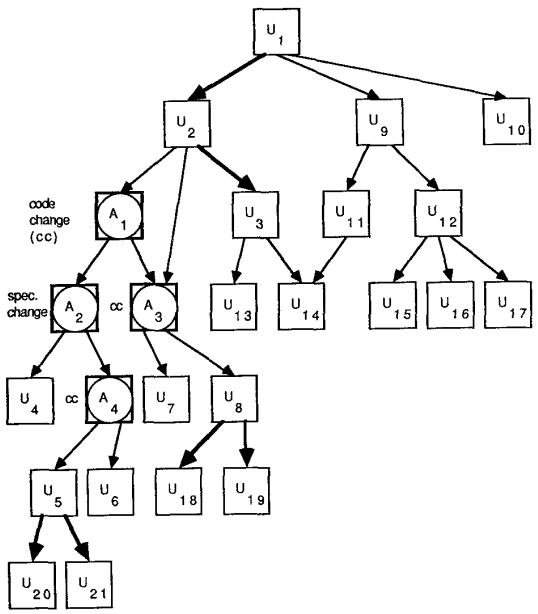

Figure 1. An Example of Use of Basis Cases and Calculation of Firewall

- values of parameters passed to M by all modules which call M;
- these parameter values may contain pointers to external input tables, as discussed in assumption (6) above;
- all data that can be accessed through these pointers passed as parameter values, such as the external input tables;
- all data that becomes accessible based on authorization codes passed as parameter values.

It should be emphasized that data *internal* to module M (such as tables of constant values) are not contained in D(M).

In general, the input spaces of a calling module and its called module will be different; in fact, the two modules may have different input variables, and the number of these input variables may be different. Let A be a calling module which calls module B. A subset of domain D(A) will cause the execution of the calling instruction to B. Before A calls B, we can compute a mapping which maps this input subdomain of A to the input space of B; the range (result) of this mapped subdomain of A will be denoted as *map(A,B),* a subset of the input space of B. This mapping may be one-to-one, many-to-one, or one-to-many, depending upon the relationship between the input variables to A and those to B. *Map*(A,B) may not contain all possible inputs to B, and some points from *map*(A,B) may be outside domain D(B).

Since both D(B) and *map*(A,B) are subsets of the input space to B, we can compare them. By intersecting these two sets, we can group the input to B into three classes:

(1) inputs which are in both D(B) and *map*(A,B);

(2) inputs which are in *map*(A,B), but not in D(B); and

(3) inputs which are in D(B), but not in *map*(A,B).

Input from both classes (2) and (3) are good test cases because they can indicate errors in the interactions of A and B. Class (2) occurs when A tries to call B with input outside the specification of B. Class (3) occurs when there are input values in the specification of B which are outside the specification of A's use of B. Unfortunately, these two classes of errors are difficult to detect since it is not always possible to effectively compare D(B) and *map*(A,B). Symbolic evaluation [6] may be used to evaluate *map*(A,B) and compare it with the symbolic path domain of B, but there are several problems with symbolic evaluation that remain to be solved [12, 19]. These problems

include: i) a high complexity involved in generating a symbolic path expression for paths that call a function or a subroutine, or paths that have complex loop structures, ii) difficulties in dealing with array and pointer variables, and iii) a high computational effort required for solving complex path expressions for test data generation.

Frequently, we intuitively refer to the process of the "traverse" of module B from A; this means that the input variables to A can be used to invoke a call of B, and when this occurs, we will obtain an input to B which is in the set *map*(A,B).

In this paper, we will show how to use this model of integration testing in a regression testing mode, where, depending upon the extent of module changes and the interactions between the modified modules and other modules, regression testing procedures are specified for both unit testing and integration testing. None of the regression analysis for integration testing in this paper will depend explicitly on the type of integration originally used.

3. Common Errors and Faults in Calling Another Module

In this section, we identify the common errors that may occur when one module calls another module. Although most of these errors should be detected during unit testing, there are some which cannot be detected before integration. An *error* is a mental mistake by a programmer or designer. A *fault* is a software defect which can cause a failure. A *failure* occurs whenever the software system fails to perform its required function according to its specification. We will first list three common errors and then classify them into two groups based upon whether the errors can be detected during unit or integration testing, respectively. Let the calling module be denoted by A and the called module by B.

(I) *Interpretation error.*

It can be argued that there are three specifications of a module: First, the *documented specification* which is (or created from) the design document, second, the *actual specification* which corresponds to the behavior of the module (reality), and third, the *interpreted specification* as perceived by a user of the module. For simplicity and the purposes of this paper, we shall assume that the documented specification is the same as the actual specification *Sa* of the module, and must be the source used to judge the correctness of the use of a module. An *interpretation error* is a misunderstanding by the module user about *Sa*, in that it differs from the interpreted specification *Si*.

Since it is difficult to reconstruct *Si*, testing should ensure that the user of the module has not made a wrong assumption. A common problem in merging modules together is that the kind of behavior expected by a user of a module may not be the same as the functionality provided by the module. An interpretation error occurs whenever the interpreted specification differs from the actual specification. The implementor of a calling module may misunderstand the functions of the called module or may be given an incomplete specification of the called module. For example, a calling module may wrongly assume that the called module will return a sorted list rather than an unsorted list. In general, although there is no testing method to guarantee the detection of this type of error, we can categorize interpretation errors into three classes:

(Ia) *Wrong function error.*

The functionalities provided by the called module B may not be those required by the specification of A. The code developer may wrongly assume that B is performing the operation that is needed by module A. In this case, the developer has the correct interpretation between both D(B) and *map*(A,B), i.e., *map*(A,B)=map_i(A,B) and [*map*(A,B)-D(B)]=[*map*(A,B)-D(B)]$_i$, where map_i(A,B) represents the user perceived *map*(A,B), and [*map*(A,B)-D(B)]$_i$ represents the user perceived difference between *map*(A,B) and D(B). Thus this error can be detected during unit testing.

(Ib) *Extra function error.*

Given the specifications of modules A and B, there are some functionalities of B not required for A, and there are some inputs from A which may invoke these functions and cause an error in A. The developer has not considered other functionalities of B which *may be used* by A. Although some functionalities of B are exactly those that are required by A, there are other functionalities of B which are not required by A. If some inputs cause these extra functionalities of B to be executed, unexpected results may be generated. This occurs when *map*(A,B)≠ map_i(A,B), and cannot be detected during unit testing.

(Ic) *Missing function error.*

There are some variable values from A used as input to B which are outside the specification of B; this can be viewed as B failing to supply all the functionalities required by A. Although some functionalities of B are exactly those that are required by A, there are some functionalities required by A that cannot be provided by B. If such functionalities are invoked by A, unexpected results will be generated. This occurs when [*map*(A,B)-D(B)]≠ [*map*(A,B)-D(B)]$_i$, and cannot be detected during unit testing.

(II) *Miscoded call error.*

A *miscoded call error* is an error which causes the developer to place the *call instruction* at the wrong point in the program. A *call instruction* is an instruction which invokes another module; the common syntax of a call instruction is the name of the called module, followed by the actual parameter list. Call(B) will be used to denote a call instruction to module B. This error may manifest itself as three possible faults:

Extra instruction fault: the call instruction is on a path which should not have the call.

Wrong placement fault: the call instruction is at the wrong location on the path which should have the call instruction.

Missing instruction fault: the call instruction is missing on the path which should have the call.

A new testing method which has received increasing attention is *fault-based testing*. The goal of fault-based testing is to demonstrate the absence of prespecified faults [18]. Examples of fault-based testing include mutation testing [7] and weak mutation testing [10]. The recently introduced RELAY model [23] uses a fault-based criterion for test data selection and guarantees the detection of errors caused by any fault of a chosen class.

Both wrong placement and missing instruction faults may be expensive to detect. For a wrong placement fault, we have to check all possible locations where the call instruction may be placed in the path. We can reduce the number of checking points using data flow information. The missing instruction fault may involve many tests because we do not, in general, know which path is missing an instruction. Therefore, we may have to check all paths. However, if we can identify the anomalous path, then the amount of testing required will be comparable to that needed for detecting a wrong placement fault.

(III) *Interface error.*

An *interface error* occurs whenever the interface standard between two modules is violated. For example, the parameters may not be in the correct order and they may not be of the right data types, formats, and input/output modes. In addition, the parameter rules (e.g., call-by-value or call-by-reference) may not be obeyed. Although some of these errors may be detected by an "advanced" compiler, we cannot always rely on a good compiler. A more serious problem occurs when the domains of the actual and formal parameters do not match. For example, a module may be designed to process a specific set of input values; if an actual parameter with value outside this set is transmitted to the module, unexpected results will occur.

Errors (Ia) and (II) (wrong function and miscoded call errors) are likely to be detected during the unit testing of A and B, provided the testing is thorough, effective and coincidental correctness does not occur. Although a wrong function error involving B may not be detected during the unit testing of A, this error should be detected during the unit testing of module B. Some errors in classes (Ib), (Ic) and (III) (extra function, missing function and interface errors) may not be detected during the unit testing of either A or B because these errors are not directly tested. Thus, their detection should be the primary objective of integration testing. Consider error (Ib): when we unit test module B, we will test all the functionalities of B, but we cannot test whether module A will ever use some of B's functionalities which are not in the specification of A. Likewise, when we unit test A, we may not detect this error. It is obvious that some error in class (III) will not be detected during unit testing of B or A since their interface cannot be thoroughly checked at that time.

4. Selecting Tests for Integration Testing

In this section, we describe our test selection strategy for detect-

ing integration errors. Like most other test selection strategies, our method does not guarantee the detection of these errors. First we will summarize the results of our error study as objectives to be achieved for this integration testing strategy.

4.1. Integration Testing Objectives

A practical objective of each testing phase is to detect errors which are unlikely to be detected by the previous testing phases. Therefore, integration testing should aim at detecting errors that may not be discovered during unit testing. To begin, we will consider integrating only two modules together. When integrating more than two modules, we can integrate them incrementally by adding one new module to the set of integrated modules. From the previous discussion in Section 3, we can identify three practical testing objectives when integrating module A with its called module B:

(1) ensure A will not use functionalities that cannot be provided by B (i.e., check for missing function error),

(2) ensure A will not use other functionalities of B if B supplies more functionalities than those required by A (i.e., check for extra function error),

(3) ensure that the interface standards between A and B are preserved (i.e., check for interface error).

4.2. Test Selection

We propose to use two types of tests: *interface tests* and *functional tests*. Some of these tests can be created using a subset of the tests applied during unit testing. By analyzing the execution path of each test, we can identify those tests which traverse the call instruction, and reuse them for integration purposes. In general, additional integration tests are needed to validate the module interactions.

A new problem is encountered when we want to "reuse" the unit tests of the called module B to test the interactions with its calling module A. This involves generating inputs to A given the inputs to B. This is a difficult problem, and such inputs to A may not even exist. However, to carry out integration testing, we must assume that the testing analyst can solve this test selection problem for A, including the situation when some specified inputs to B cannot be generated from inputs to A at all. We will say that inputs to B are *sensitized* to the calling module A if we can find corresponding inputs to A which cause the required inputs to B to be generated. A different version of the problem of sensitizing a test occurs in several other testing methods [3, 11].

4.2.1. Interface Tests

Interface tests aim to check the calling interface between the two interacting modules. There are two types of interface tests. The first type should be applied once to each syntactic call and the second type should be applied to dynamic calls. The first type includes tests which check the data type, format and the parameter passing rules of each parameter, and the order and number of parameters. This can usually be accomplished using static analysis. The second type consists of dynamic tests which check the domain of input parameters. We will call the second type *extremal tests*. Extremal tests are tests which are made up with the extreme values of the input variables.

Although the primary benefit of using extremal tests is to detect interface errors, they may also be used to detect a missing function or extra function error. When testing the integration of module A and its called module B, we should execute those extremal tests of A which also traverse module B. These tests may detect a missing function error. Since extremal values of input variables of A do not necessarily give extremal values of input to B, we should have tests which provide the extremal inputs of B. By *sensitizing* the extremal tests of B to the input of A and applying these tests to A, we may detect an extra function error.

4.2.2. Functional Tests

Both functional tests and structural tests should be applied to the module during unit testing. Howden [10] has recently formalized the functional testing method and provided guidelines for selecting functional tests. The test analyst first identifies the functions which are supposed to be implemented by his program, and then selects test data that can be used to check that the program implements the functions correctly. Both the identification of functions and the selection of test data require largely manual effort.

Any functional tests which traverse a call to another module should be identified, and repeated when we integrate the module with its called module. These tests are useful for detecting a missing function error. Also, by sensitizing the functional tests of B, we can create another set of tests which may be used for detecting an extra function error. In the case when the specification of A also describes the ways that A will use B, we can create additional functional tests which test specifically for the correct usage of B.

We will use f_A to denote functional tests of module A. For each instruction J in A, we can determine a subset of f_A which executes J. If J is a call instruction to module B, we will use $f_{A,B}$ to denote functional tests of A which also traverse the call to B. Observe that $f_{A,B}$ is a subset of f_A because they both involve inputs to module A. Given the functional tests f_B of module B, we can sensitize some of these tests to A. For each test in f_B, we may be able to find the corresponding input to A. $f_{B<A}$ is defined to be a set of test inputs to module A with the following property: each test in $f_{B<A}$, when executed by A, will traverse the call instruction of B and invoke B with a set of input parameter values that corresponds to a test from f_B. $f_{B<A}$ is not likely to be a subset of f_B because tests in $f_{B<A}$ require inputs to module A and likely require a different set of input from that for module B.

The integration test set for modules A and B should include $f_{A,B} \cup f_{B<A}$. A problem arises when either some test data from A cannot reach B or when some test data for B cannot be sensitized to A. We can definitely determine that this does not occur when

$f_{A,B} = f_A$, and

$|f_{B<A}| = |f_B|$,

where $|x|$ represents the cardinality of the set x.

We have a warning of a possible problem where additional testing may be necessary if either

(a) $f_{A,B} \subset f_A$, or

(b) $|f_{B<A}| < |f_B|$.

If (a) holds, there are some functional tests of A which do not use B. This implies that there is some function in the specification for A whose effect does not reach module B. An analysis is required to see whether any input data involving this function would reach module B; if so, this input should be added to the test set; if not, no further test data are needed. In short, we have to make sure that some input data exercising this functionality do not cause an untested condition to occur in module B, the results of which are then returned to module A, causing a potential error.

If (b) holds, there are some functional tests of B which cannot be sensitized to A. This implies that some function in the specification of B may not be used by A. An analysis is required to see whether any input data from A will reach B and exercise this function; if so, this input should be added to the test set; if not, no further test data are needed. We have to make sure that some input data from module A do not inadvertently cause an untested condition to occur in module B, the results of which are then returned to module A, causing a potential error.

5. Regression Testing at the Integration Level

After a modified module is unit tested, it should be integrated with the rest of the software. The effort involved in the integration testing will depend on the extent of the modification and the calling relations between the modified modules and other modules. In this section, we present a regression testing strategy at the integration level for a set of basis cases. Section 5.1 identifies three levels of reuse for each test case. The test selection strategy for integration of a basis set of modified modules is presented in Section 5.2. This strategy identifies the reuse levels of various existing test cases.

There are two types of modification: *non-structural modification* and *structural modification*. Both types of modifications may involve changes to the actual specification of the affected modules. In a *non-structural modification*, there is no modification of the call graph. Observe that enhancing the performance of the system seldom requires modification to the call graph. Also, no structural changes occur when we are doing 'spare parts' maintenance - replacing the entire module by another module which has the same specification and interfaces, but has a "better" implementation.

In a *structural modification*, the edges and nodes of the call graph may be added, removed, or changed. Some possible structural modifications are:

(1) Adding a new module; an example is to break up a module into two new modules.

(2) Deleting a module; an example is to merge two modules together.

(3) Adding a set of new modules; an example is to add a new software feature. A *software feature* is defined as a specific function in the software system and is usually implemented by a group of modules.

(4) Deleting a set of modules; an example is to delete a software feature.

5.1. Levels of Reuse of Tests

Before analyzing the effect of modification on the different aspects of the testing process, we introduce some definitions which will be used to describe the relations between the modified instructions and the call instruction. Informally, instruction J is in the *scope of influence* of instruction I if a) there is a definition-use relation from I to J, or b) if I is a conditional instruction, the execution of J depends on the outcomes of I [17]. We will write $S \Rightarrow R$ to denote that the set R of instructions is in the scope of influence of the set S of instructions. Instruction J is *independent* of instruction I if J is not in the scope of influence of I and I is not in the scope of influence of J. Instruction I is independent of a set of instructions $S = \{J_1, ..., J_k\}$ if I is independent of each J_i, $1 \le i \le k$. Observe that the instructions in S are not necessarily independent of each other. We will use $S \| R$ to denote that the set R of instructions is independent of the set S of instructions.

There are two ways that a test case may be changed. Recall that a test case consists of input, output and a trajectory. Any component or combination of components of a test case may be changed. We will consider that a new test case is created whenever changes are made to the input of an existing test. The allowable changes to test cases are therefore output change, trajectory change or both. It is impossible that the output will be changed without a change in the trajectory when the input is the same as before. Therefore, there are only two possible changes to a test case: trajectory change, and an output and trajectory change.

From the above observation, we can identify three levels of reuse of a test case. The *first level of reuse* is to reuse only the input of the test case. In many cases, the testing objective is to exercise some required program components. If the trajectory of an existing test case indicates that the required program components are traversed before the changed instructions, then the test input can be reused to exercise the same program components. Due to the changes in the program and/or the specification, the new trajectory and the test output may be different from the previous execution. Therefore, first level reuse test cases should be rerun.

The *second level of reuse* is to reuse both the test input and output. Tests belonging to this level are usually functional tests. When a module has only undergone code modification with its functionalities preserved, then the previous functional tests can be reused to check the correctness of the implementation. Since the trajectory may be altered or the output may be affected due to the code changes, these tests should be rerun.

The *third (the highest) level of reuse* is to reuse the input, output and the trajectory of a test case. In this case, none of the instructions in the trajectory of the test is modified. Therefore, the test output will be the same as before. There is no need to rerun these tests since the output and the trajectory will be the same as before.

Test cases in a test set S may be reused at different levels. A test set S is *reusable at level l*, $1 \le l \le 3$, if all its tests are reusable at level l or higher. A test set will be given a reusable level 0 if it has at least one test which cannot be reused. Observe that a test set at reusable level 0 may nevertheless contain some tests that are reusable.

5.2. Basis Cases for Re-integration of Two Modules

In this section, we first describe the integration testing strategies of basis cases for a non-structural modification, and then for a structural modification. Although some modifications may involve many modules, they basically consist of a combination of several basis modifications. Each basis modification involves a pair of calling-called modules, with at least one of them modified. If we have integration strategies for each basis case, we can apply these strategies to any modification.

In the sequel, *NoCh(A)* will denote that module A is not modified, *CodeCh(A)* will denote that module A has undergone code modification but its specification is not modified, and *SpecCh(A)* will denote that module A has undergone specification modification. In most cases, a specification modification also implies a code modification. However, if the specification of a called module is modified, then it is possible that the specification of its calling module will be affected although there may be no actual code modification to the calling module.

There are eight basis cases for a non-structural modification. These cases represent all the different combinations of code change and specification change to either or both calling and called modules. These cases require various degrees of analysis and effort for test generation. In each case, the integration tests should include the set of functional and extremal tests described in Section 4. Some cases can be regression tested using many previous tests while others may involve many new tests. It is understood that each modified module should first be unit tested.

We next describe the integration strategies for each of the basis cases. For those basis cases (2), (3), (4), (5) and (7), where there are multiple subcases, and a modification falls into more than one subcase, the union of the subcases would determine the test set to rerun.

(1) NoCh(A), CodeCh(B)

Since this case does not involve any specification modification, both A and B's functionalities and domains are unchanged. All the previous f_A can be reused at level 3 and f_B can be reused at level 2, and therefore the former integration tests $f_{A,B} \cup f_{B<A}$ can be reused at level 2. Observe that although f_B can be reused at level 2, this does not imply that all f_B tests should be rerun. Only a subset of f_B ($f_{B<A}$) and a subset of f_A ($f_{A,B}$) need to be rerun. We can repeat these tests to confirm that the modification does not affect the behavior of B. If these tests give the same result as before, then there is no need to propagate the integration testing above A because A and its calling modules are shown to be working properly by these integration tests.

(2) CodeCh(A), NoCh(B)

Since neither specification is modified, both f_A and f_B should remain valid. f_B can be reused at level 3 because module B is not modified, while f_A can be reused at level 2 since only the code of A is modified. Because of the modification to A, some program paths to B may be affected. The relation between the changes and the call instruction will affect the required analysis and selection of tests. There are three subcases to be considered:

(2a) Code changes $\|$ Call(B) Because the code changes and the call instruction are independent, all the previous $f_{A,B}$ should still execute the call instruction and they can be reused at level 3 because the specification of A is not modified. Also, $f_{B<A}$ can be reused at level 3 since B and the program paths to B are not modified. No execution is needed because the same results will be generated as before.

(2b) Call(B) => code changes All the previous integration tests should remain valid because there is no change in the subpaths to the call instruction and the specification of A is not modified. These integration tests can be repeated to check the interactions of A and B.

(2c) Code changes => Call(B) In this case, f_A are still valid, but they may go through different instructions. Therefore, we should compute new $f'_{A,B}$. By the same token, we may need to sensitize new tests $f'_{B<A}$.

(3) CodeCh(A), CodeCh(B)

This case is the same as case (2) except

- the reuse level of f_B is downgraded to level 2 because the code of B is modified, and
- for subcase a, the previous integration tests should be repeated to revalidate the interactions between A and B.

(4) SpecCh(A), NoCh(B)

This case is different from case (2) because it involves specification changes. Consequently, progressive regression testing of module A is needed, the reuse levels of the integration tests are reduced, and more new tests need to be selected. Because B is not modified, the previous f_B are reusable at level 3. However, new f'_A may need to be created because of the specification modification to A. Since there is a modification to A, we have to consider the relations between the code changes and the call instruction.

(4a) Code changes $\|$ Call(B) Although the code changes and the call instruction are independent, some f_A are modified. Therefore, we should compute new $f'_{A,B}$. $f_{B<A}$ can be reused at level 3 since B and the program paths to B are not modified. There is no need to repeat $f_{B<A}$ since they will give the same results as before the specification modification to A.

(4b) Call(B) => code changes $f_{B<A}$ can be reused at level 1 because its input will cause the same subpaths to the call instruction to be executed. However, the output of $f_{B<A}$ may be changed due to the specification modification to A. For the same reason, we should create new $f'_{A,B}$.

(4c) Code changes => Call(B) Because the code changes may affect the call instruction, we should compute new $f'_{A,B}$ from the new f'_A. By the same token, we need to sensitize new tests $f'_{B<A}$.

(5) SpecCh(A), CodeCh(B)

This case is the same as case (4) except

- the reuse level of f_B is downgraded to level 2, and
- for subcase a, we should repeat $f_{B<A}$ to check the modification to B.

(6) NoCh(A), SpecCh(B)

Because of modification to the specification of B, all f_A are reusable at level 2 although there is no change to A. It follows that $f_{A,B}$ are reusable at level 2. New f'_B may need to be created because of the specification modification to B. From the new f'_B, we can generate the new $f'_{B<A}$. Also for this specific case, as we shall argue subsequently, A should also be unit tested since the specification of B is changed.

(7) CodeCh(A), SpecCh(B)

Since A has only undergone code modification, the previous f_A are reusable at level 2. New f'_B are needed to test B because of the modification to B. Since there is a modification to A, we have to consider the relations between the code changes and the call instruction.

(7a) Code changes || Call(B) Because the code changes and the call instruction are independent, all the previous $f_{A,B}$ should still execute the call instruction and give the correct output. However, because of the specification modification in B, new $f'_{B<A}$ should be created.

(7b) Call(B) => code changes $f_{A,B}$ can be reused at level 2 because the subpaths to the call instruction and the specification of A are not modified. Because of the specification modification to B, new $f'_{B<A}$ should be created.

(7c) Code changes => Call(B) Since f_A may go through different paths, we should compute new $f'_{A,B}$. We may need to sensitize new tests $f'_{B<A}$ because of the new tests in f'_B.

(8) SpecCh(A), SpecCh(B)

This case is similar to integration testing during the development phase. New f'_A and f'_B need to be created. Observe that small changes in the specification may produce large or small changes in the test set; for example, such a change may render all the previous tests obsolete. On the other hand, a major change in the specification may not affect many tests. For example, the specification of A and B may undergo many changes so that they are better "matched" (i.e., B is supplying exactly the required functionalities of A). Since the two modules are better matched, less integration testing may be needed after the modification. For all three subcases, we need to design new $f'_{A,B}$ and $f'_{B<A}$.

There is one special case to be considered. It is possible that the code modifications only occur in B and the implementation of A is not modified. The specification of A is modified because of the modification to the specification of B. In this case, we can reuse those test inputs of $f_{A,B}$ for the unmodified functionalities of A because there is no change in the paths to B. Nevertheless, we need new $f'_{B<A}$ because of the modifications in A and B.

Observe that the sets f'_A and f'_B usually include some tests from f_A and f_B respectively, because not all the functionalities of the modified module are normally affected by the modification. It follows that some tests from $f_{A,B}$ and $f_{B<A}$ may be included in $f'_{A,B}$ and $f'_{B<A}$ respectively.

We next describe the basis cases which affect the call graph structure.

(9) Adding a new called module

Let the new module be denoted by B and its calling module by A. There are two subcases:

(9a) CodeCh(A) From the unit testing of B, we can obtain f_B. Although A has to be unit tested, there is no change in its functions and f_A can be reusable at level 2. Based on the location of B, we can compute $f'_{A,B} \cup f'_{B<A}$ and use this to regression test the modification.

(9b) SpecCh(A) In this case, A should be unit tested and new f'_A generated. The integration tests should include $f'_{A,B} \cup f'_{B<A}$.

(10) Adding a new calling module

Let the new module be denoted by A, which calls module B. We will assume there is no change to B and therefore no unit testing of B is required. In this case, module A should be unit tested and f'_A generated. The previous f_B can be used for generating $f'_{B<A}$.

(11) Deleting a called / calling module

Let the affected module be denoted by A and the deleted module by B. No integration testing is needed for A and B, and for the case of deleting a called module, A should be unit tested.

Table 1 summarizes the new test requirements and reuse levels of functional tests during various change conditions. We have listed the reusable levels of each test set under columns 3 to 6. A test set is given a reusable level 3 if all the components (input, output, and trajectory) of every test in the set can be reused. A reusable level 2 means that at least the first two components (input and output) of every test in the set can be reused; a reusable level 1 indicates that at least the input of every test in the set can be reused. Finally, a test set is assigned a reusable level 0 if it contains at least one test which cannot be reused. Any test set at reusable level 3 does not need to be executed since the input, output and the trajectory will be the same as before. Under the new columns, a Y is used to indicate that new tests should be created.

Given a change situation, we may use Table 1 to determine the reuse levels of various test sets, identify the testing requirement, and estimate the effort required for integration testing. For example, if case (2a) occurs, there is no need to do any integration testing. If case (2b) is encountered, we only have to repeat the previous integration tests and no new tests need to be created. In some cases, new tests should be selected by following the guideline provided for functional testing [10]. The same procedure used for the initial testing can be used to generate the test cases.

In order to implement this selection strategy, an accurate recording of the functions being tested by each test must be available. This information can be stored in table form. After the modification, the first step is to trace the specification changes to the affected functions and then examine the table to find tests related to these functions.

From the proportion of reusable and new tests, we can rank the basis cases according to the analysis and effort required to obtain the integration tests. The following relations can be established:

{(1)} -> {(2) (3)} -> {(4) (5)} -> {(8)}, and

{(1)} -> {(6)} -> {(7)} -> {(8)}.

Case (1) requires the least effort in generating the regression tests, while case (8) requires the most effort. In general, test selection for corrective regression testing is easier than that for progressive regression testing. We have put cases (2) and (3) in the same group and (4) and (5) in the same group because each pair requires roughly the same effort. Their similarities are apparent in Table 1.

6. The Notion of a Firewall

After a set of modified modules have been unit tested, the next task of the tester is to integrate these modified modules with the rest of the software system as part of the regression testing process. Two common strategies are to integration test only the modified modules, which is insufficient because some important module interactions may not be tested, and the other is to repeat the entire integration testing process beginning with the modified modules. The second strategy is usually unnecessary and a more cost-effective solution is presented in this paper. We will introduce the notion of a *firewall* to enclose the set of modules which must be integration-tested after a modification is made to the program. Under certain change conditions, we can build a firewall around the modified modules and some related modules. Only these modules within the firewall needed be re-integrated and regression tested. Because full integration is avoided, we can substantially reduce testing effort at no reduction in test effectiveness.

The construction of the firewall will involve consideration of all those modules which are not modified, but directly interact with the modified modules. These are the direct ancestors and the direct descendants of modified modules, and are shown in Figure 2 as basis cases (1) and (6), and as cases (2) and (4), respectively. In the following formal results, our objective is to prove that all modules in these basis cases must be included as modules within the firewall, but given that

		Level of Reuse				New			
Case	Changes	f_A	f_B	$f_{A,B}$	$f_{B<A}$	f'_A	f'_B	$f'_{A,B}$	$f'_{B<A}$
1	NoCh(A),CodeCh(B)	3	2	2	2				
2	CodeCh(A),NoCh(B)								
a	Changes‖Call(B)	2	3	3	3				
b	Call(B)=>Changes	2	3	2	2				
c	Changes=>Call(B)	2	3	0	0			Y	Y
3	CodeCh(A),CodeCh(B)								
a	Changes‖Call(B)	2	2	2	2				
b	Call(B)=>Changes	2	2	2	2				
c	Changes=>Call(B)	2	2	0	0			Y	Y
4	SpecCh(A),NoCh(B)								
a	Changes‖Call(B)	0	3	0	3	Y		Y	
b	Call(B)=>Changes	0	3	0	1	Y		Y	
c	Changes=>Call(B)	0	3	0	0	Y		Y	Y
5	SpecCh(A),CodeCh(B)								
a	Changes‖Call(B)	0	2	0	2	Y		Y	
b	Call(B)=>Changes	0	2	0	1	Y		Y	
c	Changes=>Call(B)	0	2	0	0	Y		Y	Y
6	NoCh(A),SpecCh(B)	2	0	2	0		Y		Y
7	CodeCh(A),SpecCh(B)								
a	Changes‖Call(B)	2	0	2	0		Y		Y
b	Call(B)=>Changes	2	0	2	0		Y		Y
c	Changes=>Call(B)	2	0	0	0		Y	Y	Y
8	SpecCh(A),SpecCh(B)	0	0	0	0	Y	Y	Y	Y
9	New(B)								
a	CodeCh(A)	2					Y	Y	Y
b	SpecCh(A)	0				Y	Y	Y	Y
10	New(A)		3			Y		Y	Y
11	Delete A or B	0	0	0	0	Y			

Table 1. Breakdown of Tests Used for Regression Integration Testing

certain conditions are met, no other unchanged modules need to be considered, and thus retested.

In this development, we need to consider both outcomes of integration testing these basis cases, one for which no error is detected, and another when an error is detected and must be corrected. In order for us to argue formally, we must assume that both unit tests and integration tests are totally reliable, in that unit tests guarantee that the computation of a module is functionally equivalent to its specification, and that integration tests guarantee that all integration errors of the types identified in Section 3 are detected. Moreover, we must also assume that all errors in the system are only due to the modification being considered, and residual errors from previous development or modifications have been removed.

First consider basis case (6) as shown in Figure 2; it is unlikely that this will occur, for ordinarily one would expect at least the code of module A to be modified. Two practical situations might have caused this case to arise:

- only a performance enhancement to B has been made, but no real functional change relative to A, so A can remain unchanged; and
- other modules which call B have required a corresponding change in B; the designer feels that the functionality provided by B to A is still the same.

The following formal result, Lemma 1, shows that if an error is detected, that module A may have to be modified after all.

Lemma 1

Consider the basis case (6), where module A is not modified, and A calls module B, whose specification is assumed to have been modified. Assume modules A and B are reliably unit tested, and A-B are reliably integration tested; if an error is detected, then either module A or B (or both) may have to be changed; if no error is detected, then any ancestor of A will not have to be tested because of the modification to B reflected through A.

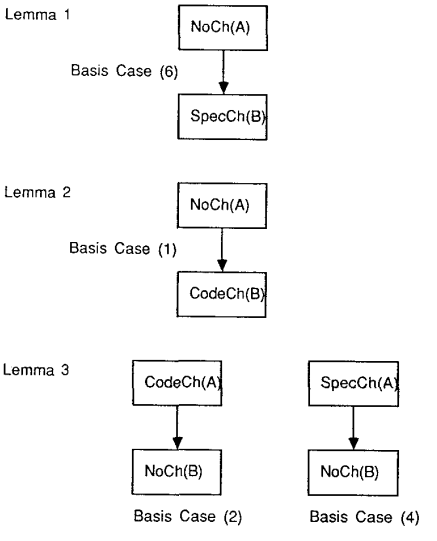

Figure 2. Boundary Cases for Firewall Construction

Proof

First assume that an error is detected: If the error is detected during unit test of A, either A or B will have to be changed (A may have to be code changed). If the error is detected during unit test of B, only B will have to be changed. There are three cases to be considered if an error is detected during integration test of A-B. If the error is an interface error, then it can be corrected in B. If the error is an extra function error, either A or B may have to be changed. If the error is a missing function error, then B can be changed.

Next assume that no error is detected; the unit test of module A ensures that it meets its specifications; the integration test with B ensures that A and B do not have interface errors. Any ancestor of A can be sure that the interface with A is correct, as no changes have occurred. The only question which might arise concerns a parameter which "passes-through" module A and interacts with the modified module B. But since this parameter is an input to module A, there must be a function in the specification of A which deals with this parameter, even though the computation with this parameter does not occur until module B (or some descendant of B). This function must be tested during unit test of A, and is also considered during the integration test of A-B, and thus should be correct. □

Basis case (6) is unusual in that module A may have to be changed as indicated in Lemma 1; however, this follows because basis case (6) should rarely occur, and the interface above a modified module B should be that described in basis case (1), where only the code of B has been changed. This also provides a guideline for designing the modification, as a code change in a module is preferable to a specification change, in order to keep the firewall from spreading upward through the call graph.

Next consider basis case (1). We want to argue that when this case is encountered, and it should be encountered in most practical cases, then we can establish a "firewall" above the modified modules.

Lemma 2

Consider the basis case (1), where module A is not modified, and A calls module B, whose specification is assumed to be unchanged, but its code is modified. Assume module A is reliably unit tested, and A-B are reliably integration tested; if an error is detected, then only module B has to be changed, and neither the specification nor the code of A have to be modified. If no error is detected, then A or any ancestor of A will not have to be tested because of the modification to B reflected through A.

Proof

First observe that a unit test for A need not be rerun since the specification for modules A and B have not changed. The unit test of B will assure that the functionality in B's specification will be met, but an integration test of A-B is required. If an error occurs in the unit test of B, clearly a further code change in B can correct this. If an error is detected in the A-B integration test, then since the specification and code of A have not changed, the only possibility is that module B must have been invoked. Moreover, since A has not changed, the same parameter values are communicated to B from A. Thus the input-output behavior of B must be different than it was previously, for A has not changed to explain a different output, and this cannot happen given the unit test of B. Since the specification of B has not changed, extra function errors and missing function errors cannot occur; if an interface error has occurred, it can clearly be corrected in module B.

Next assume that no error is detected; the argument that no ancestor of A need be retested because of the A-B interface is the same as Lemma 1, except that we can use the unit test of B to ensure that a parameter "passing-through" module A does not lead to an error in the module which originated that parameter or in any module in between. □

Lemma 2 shows that for case (1), when a modified module B is called by an unchanged module A, we can establish a "firewall" at A, and since A will not change, no ancestors of A need be examined for testing; it is possible, however, that these modules might be affected through interactions with other modified modules, but not through A.

Next consider basis cases (2) and (4), where a modified module A calls an unchanged module B, and under what conditions descendants of B also need to be examined.

Lemma 3

Consider the basis cases (2) and (4), where module A is either modified in code but not in specification, or its specification is modified, respectively, and A calls B, where B is unchanged. If either the unit tests of A or the integration tests of A-B are in error, then for either case (2) or case (4), we can correct the error by only modifying the code of A. If no error is discovered by either the unit test of A or integration test of A-B, then no descendant of B need be examined or tested because of the change of module A reflected through B.

Proof

An error in either the unit test of A or integration test of A-B is due to an improper design of the new module A. Since the specification of B is unchanged, any detected integration error is due entirely to the change in module A, and so can be corrected in module A alone.

If no error is detected in either the unit test of A or integration test of A-B, then since the specification and code of B are unchanged, no other descendant of B has to be examined or tested. If we are concerned about the effects of a parameter from module A which may be in error, passing-through module B, and affecting some descendant of B, then the unit test of A should suffice to adequately test the functionality of that parameter.
□

Lemma 3 shows the very serious effect of a design error during maintenance. In the modification of either the code or specification of module A, a common error is to misinterpret the effect of a call to module B during the execution of module A. We have shown that it is always possible to correct the error by further modification of module A. However, there may be constraints imposed upon such changes, or complexities in such modification of A. If the designer chooses to modify the specification of module B, because it is consider to be "simpler", then one consequence is that now the "firewall" must be extended, as B is changed, and any modules which B calls might also have to be modified in turn if there are errors, thus extending the "firewall" further down in the call graph. Moreover, if any other modules call B, the effect of its change upon their performance must also be established. Thus the potential cost of a design error during maintenance and the decision to modify B rather than simply modifying A is clear.

6.1. Constructing a Firewall

We can now proceed to a definition of the intuitive notion of *firewall*. A graph component is *connected* to a set S of arcs of the call graph if every arc in S is connected to exactly one node of that component. Define a graph component C as all those nodes of the call graph which correspond to modified modules, together with all modules which are their direct ancestors and all modules which are their direct descendants, and all arcs with both nodes in this defined set of nodes.

A *firewall* is composed of the subset E of arcs in the call graph with the following properties:

i) Graph component C is *connected* to the set of arcs E.

ii) The removal of the firewall arcs E separates the call graph into separate components.

Then every module interaction within C must be integration tested while those outside C need not be retested. The function of the firewall E is to clearly separate graph component C from the rest of the call graph.

Recall that we have assumed that the call graph is a connected directed acyclic graph. For simplicity, first assume that the set W of all modified modules and arcs between them comprise a connected subgraph of the call graph. Later we will indicate how to handle more complex situations when this constraint is relaxed. The *firewall* for W is a subset E of arcs in the call graph such that the following constructions are followed and properties are satisfied:

(a) Initially set E is empty.

(b) Carry out all unit and integration tests using the basis cases within W; if errors are detected between modified modules, then either incorrect specifications or incorrect code should be corrected so as to produce correct integration tests. For the firewall calculations, we are only interested in the unit and integration tests involving those unchanged modules directly connected to the modules in W.

(c) For each unchanged module A which calls a modified module B in W:

We expect that this will likely occur as basis case (1), because basis case (6) is less probable to occur.

If case (6) occurs and no error is detected, then by Lemma 1 there is no need to check the ancestor of A. Add module A, and all arcs from A to modified modules in W, to W. Add all other arcs into and out of module A to the firewall E under construction.

If case (6) occurs and an error is corrected by changing the code of A, then module A is now modified and added by definition to subgraph W together with all arcs from modified modules. All modules connected to A in any way must now be considered for integration testing, and in particular, step (c) must be repeated if A has any ancestors in the call graph.

Lemma 2 indicates that for case (1) even if an error should be detected in the operation of modules A and B, then module B can be corrected to rectify the problem and neither the specification nor the code of module A need be changed. Thus add module A, and all arcs from A to modified modules in W, to W. Add all other arcs into and out of module A to the firewall E under construction.

(d) For each unchanged module B which is called by a modified module A in W:

If no error is detected by unit or integration testing, then we can add B to W, together with all arcs from modified modules in W to module B. Add all other arcs into and out of module B to the firewall E under construction.

If an error is detected involving modules A and B, Lemma 3 assures us that the error can be corrected by changing A. But if the error is corrected by modifying the specification of module B, then module B is now modified and added by definition to subgraph W together with all arcs from modified modules. All modules connected to B in any way must now be considered for integration testing, and in particular, (d) must be repeated if B has any descendants in the call graph.

(e) After all unchanged modules recursively described in steps (c) through (d) have been considered, the firewall E is complete. If the arcs in E are deleted, the set of modules and arcs in subgraph W form a separate component in the call graph.

In the above procedure for defining the firewall for integration testing, we assumed that the set W of modified modules and the arcs between them comprised a connected subgraph of the call graph. Because of the distributed nature of computation, and also because several unrelated modifications might be implemented at the same time, this assumption might not always be valid. In this case, W might consist of several connected subgraphs W_1, ..., W_n, where each consists only of modified modules, but is maximal with respect to the property of being connected. The only difference between this case and that analyzed in the above procedure is that if the calculated firewalls for two of these subgraphs overlap, then they should be coalesced into one connected subgraph. More specifically, this will occur when a single unchanged module A is called by a modified module in a connected subgraph W_i, and A also calls a modified module in a connected subgraph W_j; then a new connected subgraph should be formed by $W_i \cup W_j$, together with module A and the two arcs involved. For any other connected subgraph W_k, where a modified module in W_k is also connected to module A, then W_k should also be added to this connected subgraph. On the other hand, if an unchanged module calls a number of distinct connected subgraphs W_1, ..., W_n, but is not called by any connected subgraph W_k, then these subgraphs should not be coalesced. Note, however, that the unchanged module should be integration tested within each of these connected subgraphs W_1, ..., W_n. The situation is similar for the case of a single unchanged module which is called by a number of distinct connected subgraphs W_1, ..., W_n.

6.2. An Example

Figure 1 shows an example of the use of the basis cases on a given modification to a call graph, and the subsequent calculation of a firewall of this modification. The modified modules are labelled A_i, $1 \le i \le 4$, and they form a connected subgraph W. Modules A_1, A_3 and A_4 all have code modification and module A_2 has specification modification. The unchanged modules are labelled U_j, $1 \le j \le 21$. All the basis cases are represented in this example except cases (6) and (8). The firewall E is calculated by first identifying those arcs into and out of unchanged module U_2 which calls a modified module in W. Initially, the subgraph W consists of $\{A_1, A_2, A_3, A_4, (A_1, A_2), (A_1, A_3), (A_2, A_4)\}$. Then U_2, arcs (U_2, A_1) and (U_2, A_3) would be added to W and arcs (U_1, U_2) and (U_2, U_3) would be added to E by step (c). Next we identify unchanged modules which are called by modules in W. If the integration tests yield no errors, then we can add more arcs to E. Specifically, U_4, U_7, U_8, U_5, U_6 and arcs (A_2, U_4), (A_3, U_7), (A_3, U_8), (A_4, U_5), (A_4, U_6) would be added to W and (U_5, U_{20}), (U_5, U_{21}), (U_8, U_{18}), (U_8, U_{19}) would be added to E by step (d). If the arcs in E are deleted from the call graph, W is a component in which all modules must be integration tested. The firewall E is indicated by bold arrows in Figure 1.

7. An Empirical Study

In this section we present a case study using our regression testing strategies for unit, integration and system testing. The unit regression testing strategy is outlined in [15]; the integration regression strategy is described in Sections 5 and 6. This study aims to investigate the following areas:

(1) the effectiveness of our regression testing framework, and

(2) the amount of saving in test effort, in terms of the number of test cases, over the traditional "retest-all" strategy.

The target program creates a student database which records the student names, identification numbers and all assignment marks. The user can enter new student records, new student marks, update an assignment mark for all the students, delete a student record, and display statistical information about the class marks. The program also checks for valid student identification numbers and assignment numbers. The original version of this interactive program has been developed and used by an instructor, an experienced designer/programmer, to demonstrate some database concepts. The actual version used for our experiment contains a few minor modifications. Figure 3 shows the call graph of the program StudentDatabase. This program consists of twenty distinct modules, thirty-one module interactions which involve the use of thirty-two modules and over 550 lines of Pascal code.

7.1. Test Cases Selection Strategy

During unit testing, functional tests were first selected based on the specification of each module. Additional structural tests for satisfying the *all-use* structural coverage criterion were then created [21]. In most cases, we only needed a few additional tests to satisfy the structural criterion. In designing the unit tests, extremal values of the input variables were used for all the functional tests whenever possible. This helped to reduce the number of required tests since each test may be testing a function and a boundary of an input domain simultaneously. Besides using this simple strategy to reduce the number of tests, no other minimization of the number of test cases was performed.

In integration testing, we applied both functional and extremal tests. A bottom-up integration strategy was used. In most cases, the integration tests were just the union of the unit tests of the two modules involved in the module interaction.

System tests were designed mainly to test all the software features of the system and were required to satisfy a testing criterion developed by the authors. The total number of unit tests, integration tests and system tests were 120, 235 and 104, respectively, with over 80% of the integration tests the same as the unit tests.

An interesting discovery is that there are tests for unit testing which will never occur during the normal operation of the system; that is, these tests cannot be *sensitized* to the main module of the software system. For example, in unit testing of module Swap, tests with 0 assignments, 3 assignments and the maximum number of assignments were used. But, in actual usage of the StudentDatabase program, Swap will not be called when no assignment mark is entered. Thus, the unit test of 0 assignments was superfluous. However, this test was required for satisfying the all-use structural selection criterion.

7.2. Program Modifications

Four modifications were requested by the user:

1. Check for unique student identification numbers.
2. Add an option for plotting a mark distribution graph.
3. Allow different maximum marks for each assignment.

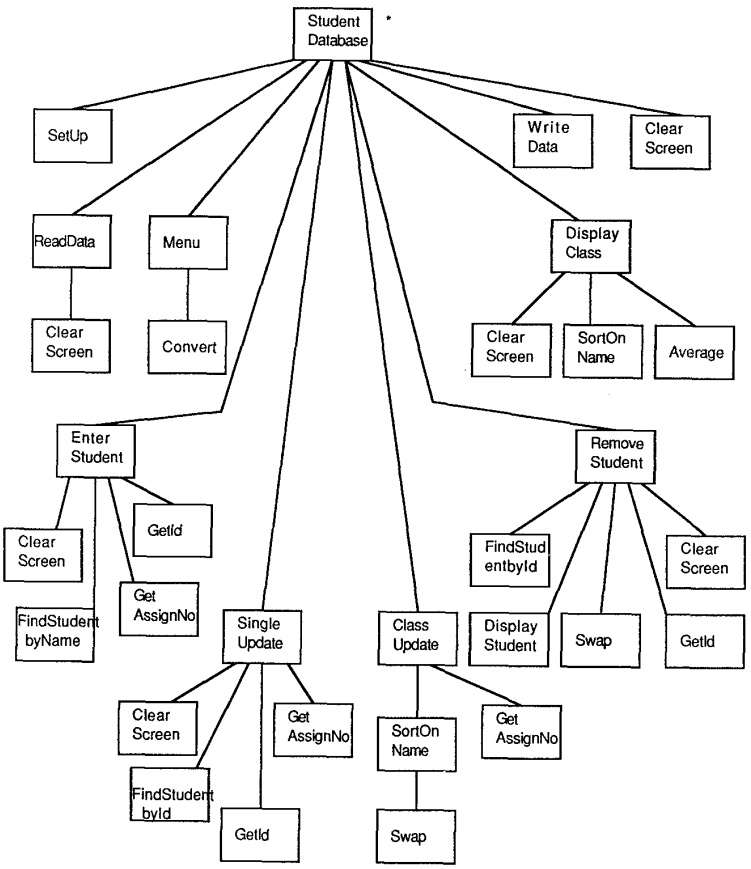

Figure 3. Call graph of StudentDatabase program

4. Assign grades to all students.

Modifications 1, 2 and 4 involved structural modification to the call graph, whereas modification 3 only involved non-structural modification.

7.3. Results

Table 2 summarizes the actual changes and the number of regression tests used for each modification. The total unit test represents the total number of new and old unit tests for testing all affected modules. This number is typically less than the total number of unit tests used to test all the modules in the initial testing because only a few modules are affected in each modification. The total integration test represents the total number of integration tests required to test all new and existing module interactions and the total system test denotes the total number of system tests. The sum of all three total tests represents the total number of regression tests used by the "retest-all" strategy. The last row of Table 2 gives the ratio of the number of regression tests according to our strategy to that of "retest-all" strategy.

To compare the effectiveness of our strategy with the "retest-all" strategy, we reran all the tests in the test plan to check whether they would detect extra errors. No new error was discovered. Thus, we conclude that extra testing by rerunning all the tests in the current test plan does not detect more errors than our strategy. The size of the test subsets for our regression strategy relative to the retest-all strategy are 24%, 17%, 63% and 30% for the four modifications, respectively. Thus the average reduced subset is 34%. These results reveal that using our strategy requires significantly fewer tests than the "retest-all" strategy. The major reason for this saving is that the concept of a firewall is effective in reducing the amount of integration testing. The savings observed in this experiment would most likely be generalized to other maintenance projects, although possibly not to the same degree. If further research establishes consistency with this result, then our strategy can be adapted to most maintenance projects.

One interesting observation is that the number of affected source lines does not seem to be the deciding factor in predicting the number of regression tests. The important factor seems to be the "extent" of a modification which can be represented by the number of affected features and affected module interactions, which subsequently has a greater effect on systems regression testing. Although modifications 2 and 4 involved more affected source lines than modification 3, modification 3 actually required the most regression tests. This occurs because modification 3 has the highest number of affected module interactions (16) and affected features (7) among the four modifications.

To test the effectiveness of our regression testing strategy, we seeded some errors in the changes and reran the set of regression tests.

	Modification			
	1	2	3	4
Number of Affected Source Lines	25	80	23	57
Number of Affected Modules	2	4	8	8
Number of Affected Module Interactions	2	3	16	10
Number of Affected Features	1	1	7	1
Number of Regression Unit Tests	15	22	50	40
Number of Regression Integration Tests	32	32	120	80
Number of Regression System Tests	46	24	130	38
Total Unit Test	27	40	67	66
Total Integration Test	246	278	275	307
Total System Test	106	130	130	158
Regression Tests/Total Tests	93/379	78/448	300/472	158/531
	24%	17%	63%	30%

Table 2. Modification Characteristics and Regression Tests

All seeded errors were placed in the affected modules to simulate possible errors made when the code was modified. Table 3 shows the results. A total of 13 logic errors were inserted in the four modifications. Two of these errors may be classified as extra function errors, one as a missing function error and the rest as wrong function errors. All errors except one were detected using our set of regression tests. 8 errors were detected during regression unit testing, 11 during regression integration testing and 12 during regression system testing. There were three errors which were not detected during regression unit testing but were detected during regression integration testing.

The error which was not detected by our testing strategy was an extreme case of an *assignment blindness error [26]*. Some blindness errors are undetectable by any testing strategy. Our error was not detected because an incorrect program constant was used, but by coincidence it had the same value as the correct program constant. In the program, if Max and MaxAssign have the same values, then an error in the loop implementing the initialization of marks in modification 3 will not be detected. Since the program constants are assigned values at the declaration stage, our error is undetectable even if all program paths are exercised. This suggests that we should also test the program by assigning different values to its program constants.

Our results confirm the expectation that some errors are likely to be missed at a certain testing phase and can only be detected at another testing phase. Many errors cannot be detected during unit testing due to the limited focus and incorrect assumption of the module's input and output. Most, but not all, of these errors were discovered during integration testing when the testing analyst considered a slightly larger "global viewpoint" of the software. Finally, some errors were not detected until system testing because they required complex interactions between modules that were not possible in the previous two phases of testing.

Although in practice most software organizations place less emphasis on integration testing and concentrate on extensive unit and system testing, our strategy has the advantage of detecting most errors before system testing starts. During both regression testing of the four modifications and the experimentation with the seeded errors, all but one of the detected errors were discovered during unit and integration testing. This suggests that by emphasizing integration testing, fewer errors are left to be discovered during system testing. Our approach is recommended because the earlier an error is detected, the less costly it is to fix. Thus, it is more desirable to detect an error during unit testing than at integration testing time, with a similar advantage of detection during integration testing rather than waiting for the testing process to be completed.

The test selection effort is actually quite manageable, despite the large number of test cases. Over 80% of the integration tests are the same as the unit tests and many systems tests (over 60%) are directly derived from the integration tests. Although it seems that the duplicate tests can be avoided, the different test sets actually aim to detect different types of errors. For example, the integration tests aim to detect the interface, missing function and extra function errors. Also, the set of modules that are being tested are different between unit and integration testing as integration testing uses more modules. Thus, although some tests are the same, they are exercising the system under slightly different conditions and are not wasting testing resources.

	Seeded Errors	Integration Errors	Detected During		
			Unit	Integration	System
Modification 1	missing computation	wrong function	Y	Y	Y
	wrong boundary condition	extra function	Y	Y	Y
Modification 2	missing computation	wrong function	Y	Y	Y
	wrong boundary condition	wrong function	Y	Y	Y
	off by 1 iteration	extra function	N	Y	Y
	failure to initialize	wrong function	Y	Y	Y
Modification 3	wrong computation	wrong function	Y	Y	Y
	extra change	wrong function	N	Y	Y
(undetected)	wrong change	wrong function	N	N	N
	missing change	wrong function	N	Y	Y
Modification 4	wrong computation	wrong function	Y	Y	Y
	missing condition	missing function	Y	Y	Y
	missing condition	wrong function	N	N	Y

Table 3. Detected Seeded Errors

8. Conclusions and Future Research

We have identified some common errors in software integration. These errors usually arise because of a misunderstanding of the specification of the called module. To study this problem, we have proposed a model of integration testing, with some general guidelines for selecting integration tests, assuming that unit tests have been performed for all affected modules. A number of basis cases have been analyzed in which the integration testing for two interacting modules is specified for a variety of conditions. The concept of a *firewall* was defined, which imposes a limit on those modules which must be considered when one or more modules are modified. An example is given to illustrate the use of these basis cases and the determination of the firewall. Finally, a study applying our regression testing strategy was carried out with encouraging results.

In this paper, we have used a relatively simple model for both functional tests and specifications; it would be important to examine some more realistic and complex models of functional tests and specifications to see how they would affect our analysis and test generation guidelines. It seems that practitioners design and select tests for interface testing, but they may not be explicitly aware of the necessity to generate tests for missing function and extra function errors described in Section 3. In the generation of these tests, we have required a complex analysis of the specifications of both the calling module and called module. It would be preferable to offer practitioners a simpler and more automated approach to obtain this information.

These ideas and concepts need to be implemented as a tool, and to be applied to some large software systems. This should provide additional evaluation of the three types of integration errors, to see how often each occurs in practice. The concepts of integration testing basis cases and of a firewall should also be evaluated with different types of programs and under different types of maintenance scenarios.

Acknowledgement

David Olshefski of Thomas J. Watson Research Center, IBM has made valuable comments which have significantly improved the presentation of this paper.

References

1. V. R. Basili and B. T. Perricone, "Software errors and complexity: an empirical investigation," *CACM*, vol. 27(1), pp. 42-52, Jan. 1984.
2. B. Beizer, in *Software system testing and quality assurance*, Van Nostrand, 1984.
3. T. A. Budd, "Mutation analysis: ideas, examples, problems and prospects ," in *Computer Program Testing*, ed. S. Radicchi, pp. 129-148, North-Holland, Amsterdam, 1981.
4. R. Carey and M Bendick, "The control of a software test process," *Proc. COMPSAC 77*, pp. 327-333, Nov. 1977.
5. L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil, "A comparison of data flow path selection criteria," *Proc. 8th Int. Conf. Software Eng*, pp. 244-251, 1985.
6. L. A. Clarke and D. J. Richardson, "Symbolic evaluation methods - Implementations and applications," in *Computer Program Testing*, ed. S. Radicchi, pp. 65-102, North-Holland, Amsterdam, 1981.
7. R. A. DeMillo, R. J. Lipton, and F. G. Sawyer, "Hints on test data selection: help for the practicing programmer," *Computer*, vol. 11, pp. 34-41, April 1978.
8. M. J. Harrold and M. L. Soffa, "An incremental approach to unit testing during maintenance," *Proc. Conf. Software Maintenance*, pp. 362-367, 1988.
9. J. Hartmann and D. J. Robson, "Techniques for selective revalidation," *IEEE Software*, pp. 31-38, Jan. 1990.
10. W. E. Howden, in *Functional program testing and analysis*, McGraw-Hill, 1987.
11. W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Trans. Software Eng.*, vol. SE-8 (2), pp. 371-379, July 1982.
12. D. C. Ince, "The automatic generation of test data," *The Computer Journal*, vol. 30, no. 1, pp. 63-69, 1987.
13. B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, pp. 155-163, Oct. 1988.
14. 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.
15. H. K. N. Leung and L. White, "Insights into regression testing," *Proc. Conf. Software Maintenance*, pp. 60-69, Miami, FL, Oct. 1989.
16. H. K. N. Leung and L. White, "Insights into testing and regression testing global variables," *Technical Report CES-90-18*, Dept. of Comp. Eng. and Sc., Case Western Reserve University, July 1990.
17. 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.
18. L. J. Morell, "Theoretical insights into fault-based testing," *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pp. 45-62, July 1988.
19. S. S. Muchnick and N. D. Jones, in *Program flow analysis: Theory and Applications*, Prentice-Hall International, 1981.
20. G. J. Myers, in *Software reliability: principles and practices*, New York: Wiley-Interscience, 1976.
21. S. Rapps and E. J. Weyuker, "Data flow analysis techniques for program test data selection," *Proceedings Sixth International Conference on Software Eng.*, pp. 272-278, Tokyo, Japan, Sept. 1982.
22. 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.
23. D. J. Richardson and M. C. Thompson, "The RELAY model of error detection and its application," *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pp. 223-230, July 1988.
24. W. P. Stevens, G. Myers, and L. L. Constantine, "Structured Design," *IBM Systems Journal*, vol. 13, 1974.
25. S. S. Yau and Z. Kishimoto, "A method for revalidating modified programs in the maintenance phase," *Proc. COMPSAC 87*, pp. 272-277, 1987.
26. S. J. Zeil, "Testing for perturbations of program statements," *IEEE Trans. Software Eng.*, vol. SE-9 (3), pp. 335-346, May 1983.