Review of Regression Testing on Object-Oriented Programs

Sudhir Mohapatra¹, Pradipta Kumar Mishra² and B.K.S.S Pattanaik¹

¹Gandhi Institute for Technological Advancement, Bhubaneswar.
²Centurion University of Technology and Management, Bhubaneswar.

ABSTRACT

The purpose of regression testing is to ensure that bug fixes and new functionality introduced in a new version of a software do not adversely affect the correct functionality inherited from the previous version. Regression testing is an expensive and frequently executed maintenance process used to revalidate modified software. It is costly but crucial problem in software development. The paper try to do the survey of current research on regression testing and current practice in industry and also try to find out whether there are gaps between them.

Introduction

Changes in software is inevitable when computer based system are built, therefore we have see that functionalities that were working in the previous version are still working in the new version. Mistaken and changed requirements cause the software to be reworked. New uses of old software yield new functionality not originally conceived in the requirements[1]. The management of this change is critical to the continuing usefulness of the software. Regression testing attempts to revalidate the old functionality inherited from the old version.

The new functionality added to a system may be accommodated by the standard software development processes. Regression testing attempts to revalidate the old functionality inherited from the old version. The new version should behave exactly as the old except where new behavior is intended. Therefore, regression tests for a system may be viewed as partial operational requirements for new versions of the system.

Fig 1

Figure 1 shows a typical example of a sequence of time intervals during the life of a software system. Note that the regression testing intervals occupy a significant fraction of the system's lifetime. Unfortunately, complete regression testing can not always be accommodated during frequent medications and updates of a system as it is often time consuming. This may result in the escape of costly, improper changes into the field. Clearly, omitting or arbitrarily reducing the regression testing interval is not an acceptable solution to the problem of software revalidation.

After a program has been modified, we must not only ensure that the modifications work correctly but also check that the unmodified parts of the program have not been adversely affected by the modifications. This is necessary because small changes in one part of a program may have subtle undesired effects in other seemingly unrelated parts of the program. Even though the modified program may yield correct outputs on test cases specifically designed to test the modifications, it may produce incorrect outputs on other test cases on which the original program produced correct outputs. Thus, during regression testing, the modified program is executed on all existing regression tests to verify that it still behaves the same way as the original program, except where change is expected.

During the maintenance of a software system or as the software evolves, the regression testing is the expensive but definitely necessary task. Since the cost of the software[3,4] maintenance account for about two-thirds of the whole software, both project managers and researchers have to pay more and more attention to the regression testing. There have been a lot of researches on regression testing.

Regression testing techniques are categorized as: regression test selection techniques, test prioritization techniques, test suite reduction and hybrid techniques. The regression test selection technique chooses the tests from the old test suite to execute on the modified version of the software. Regression test prioritization techniques reorder test suite with a goal to increase the effectiveness of testing in terms of achieving code coverage earlier, checking frequently used features of software and early fault detection. Test suite may contain more than enough test cases for satisfying the requirements. Test suite reduction say, when some test cases are removed from the test suite, the test suite may still satisfy all the requirements that can be satisfied by the original test suite. Hybrid techniques combine both selection and prioritization for regression testing.

Basic Regression Testing Technique

Let P be a procedure or program, let P’ be a modified version of P, and let T be a test suite for P. As described in [1], a typical regression test procedures is:

1. Select T’ T, a set of test cases to execute on P’.
2. Test P’ with T’, establishing P’’s correctness with respect to T.

© 2014 Elixir All rights reserved.
3. If necessary, create T', a set of new functional or structural test cases for P'.
4. Test P' with T', establishing P'’s correctness with respect to T'.
5. Create T'''', a new test suite and test execution profile for P', from t, T', and T'''.

**Topic Involve In Regression Testing**
Researchers had addressed the wide variety of topics in their year of research on regression testing. Here we briefly describe several techniques and give a representative example of each.

**Regression test selection**
A safe regression-test-selection technique is one that, under certain assumptions, selects every test case from the original test suite that can expose faults in the modified program. Regression-test-selection techniques are particularly effective in environments in which changed software is tested frequently [16]. For example, consider an environment in which nightly builds of the software are performed and a test suite is run on the newly built version of the software. In this case, regression test selection can be used to select a subset of the test suite for use in testing the new version of the software. The main benefit of this approach is that, in many cases, a small subset of the test suite is selected, which reduces the time required to perform the testing. For another example, consider a development environment that includes such a regression-test-selection component. In this case, after developers modify their software, they can use the regression test selector to select a subset of the test suite to use in testing. With this approach, developers can frequently test their software as they make changes, which can help them locate errors early in development [15]. The techniques are also effective when the cost of test cases is high. An example is the regression testing of avionics software. In this case, even the reduction of one test case may save thousands of dollars in testing resources.

Software testing is the activity of executing a given program P with sample inputs selected from the input space for P, to try to reveal failures in the program. The underlying idea is that, under the hypothesis that a test case is well-designed, its inability to reveal failures increases our confidence in the tested code. To perform testing, it is necessary to select a set of input data for P, to determine the expected behavior of P for such data with respect to a given model, and to check the results of P’s execution against the expected behavior. A test case consists of the input data that is provided to P, together with the corresponding expected output. A test suite is a set of test cases, and a test run is the execution of P with respect to a test suite T. For each test case t in T, a test run for T consists of the following three steps: (1) initialization of the environment, (2) execution of P with input specified in t, and (3) checking of the output of the execution with respect to the expected output. To assess the adequacy of a given test suite T, we measure the level of coverage achieved by the test run. Although functional coverage can be measured as well, coverage is usually computed for structural entities (i.e., entities in the code). Common examples of entities considered for structural coverage are statements, branches, and conditions. Coverage is measured as a percentage. For example, statement coverage is defined as the percentage of statements covered by the test run with respect to the total number of executable statements. The coverage information can be obtained in several ways. One method produces an instrumented version of P such that when this instrumented version of P is executed with a test case t, it records the entities in the program, such as statements or branches, which are executed with t. An alternative method for obtaining the coverage information modifies the runtime environment so that when P is executed with t, the environment gathers the information about the entities covered.

Let P' be a modified version of P, and T be the test suite used to test P. During regression testing of P', T, and information about the testing of P with T are available for use in testing P'. In attempting to reuse T for testing P', two problems arise. First, which test cases in T should be used to test P' (the regression-test-selection problem). Second, which new test cases must be developed to test parts of P' such as new functionality (the test-suite augmentation problem). Although both problems are important, in this paper we concentrate on the regression-test-selection problem. Regression-test-selection techniques attempt to reduce the cost of regression testing by selecting T', a subset of T, and using T' to test P'.

![Diagram](image)

A number of safe regression-test-selection techniques that vary in precision and efficiency have been presented. We can view these techniques as a family of regression-test-selection techniques that use information about the program's source code to select T'. Figure 2 illustrates a general regression-test-selection system. In this system, a program P is executed with a test suite T. In addition to the results of the execution the pass/fail information the system records coverage information about which entities in P are executed by each test case t. The types of entities recorded depend on the specific regression-test-selection technique. After all test cases have been run, the coverage information is compiled into a coverage matrix that associates each t in T with the entities that it executes. In addition to computing coverage information, these techniques compare P and P', and identify in P a set of dangerous entities [4]. We define P(i) as the execution of P with input i. A dangerous entity is a program entity e such that for each input i causing P to cover e, P(i) and P'(i) may behave differently due to differences between P and P'. The technique ensures that any test case that does not cover a dangerous entity will behave in the same way in both P and P', and thus, cannot expose new faults in P'. Thus, it is safe to select only those test cases for which the coverage matrix indicates coverage of a dangerous entity.

Several regression test selection techniques have been investigated. Here we briefly describe several techniques and give a representative example of each.

**Re-test-all Technique:**
This method reruns all test cases in T. It may be used when test effectiveness is the utmost priority with little regard for cost.

**Random/Ad-Hoc Technique**
Testers often select test cases randomly or rely on their prior knowledge or experience. One such technique is to randomly select a percentage of test cases from T.

**Minimization Technique**
This approach [5], [6] aims to select a minimal set of test cases from $T$ that covers all modified elements of $P'$. One such technique randomly selects test cases from $T$ until every program statement added or modified to create $P'$ is exercised by at least one test case.

Dataflow Techniques. Dataflow-coverage-based regression test selection techniques [8] select test cases that exercise data interactions that have been affected by modifications.

Safe Technique

These techniques [4],[7] select, under certain conditions, every test case in $T$ that covers hanged program entities in $P'$. One such technique selects every test case in $T$ that exercises at least one statement that was added or modified to create $P'$, or that has been deleted from $P'$.

Test case prioritization [9]

This technique of regression testing prioritize the test cases so as to increase a test suite’s rate of fault detection that is how quickly a test suite detects faults in the modified program to increase reliability. This is of two types: (1) General prioritization which attempts to select an order of the test case that will be effective on average subsequent versions of software . (2) Version Specific prioritization which is concerned with particular version of the software.

Test Case Prioritization problem Rothermel et al. [17] define the test case prioritization problem as follows:

Given: $T$, a test suite; $PT$, the set of permutations of $T$; $f$, a function from $PT$ to the real numbers. Problem: Find $T' \in PT$ such that $(f(T')) (T' \in PT) (T' \neq T') [f(T') \geq f(T')]$. Here, $PT$ represents the set of all possible prioritizations (orderings) of $T$ and $f$ is a function that, applied to any such ordering, yields an award value for that ordering.

Test Case Prioritization Techniques There are 18 different test case prioritizations techniques [14] numbered P1-P18 which are divided into three groups as shown in figure 2.

Comparator techniques:

P1: Random ordering: in which the test cases in test suite are randomly prioritized.

P2: Optimal ordering: in which the test cases are prioritized to optimize rate of fault detection. As faults are determined by respective test cases and we have programs with known faults, so test cases can be prioritized optimally.

Statement level techniques: (Fine Granularity)

P3: Total statement coverage prioritization: in which test cases are prioritized in terms of total number of statements by sorting them in order of coverage achieved. If test cases are having same number of statements they can be ordered pseudo randomly.

P4: Additional statement coverage prioritization: which is similar to total coverage prioritization, but depends upon feedback about coverage attained to focus on statements not yet covered. This technique greedily selects a test case that has the greatest statement coverage and then iterates until all statements are covered by at least one test case. The moment all statements are covered the remaining test cases undergo Additional statement coverage prioritization by resetting all statements to “not covered”.

P5: Total FEP prioritization: in which prioritization is done on the probability of exposing faults by test cases. Mutation analysis is used to approximate the Fault-Exposing-Potential (FEP) of a test case. The cost of calculating FEP using mutation analysis is quite high which motivates the search of cost effective approximators of FEP.

P6: Additional FEP prioritization: the total FEP prioritization is extended to Additional FEP prioritization as the total statement coverage prioritization is extended to Additional statement coverage prioritization.

Function level techniques: (Coarse Granularity)

P7: Total function coverage prioritization: it is similar to total statement coverage but instead of using statements it uses functions. As it has got coarse granularity so the process of collecting function level traces is cheaper than the process of collecting statement level traces in total statement coverage.

P8: Additional function coverage prioritization: it is similar to Additional statement coverage prioritization with only difference that instead of statements, it is considering function level coverage.

P9: Total FEP prioritization (function level): it is analogous to Total FEP prioritization with only difference that instead of using statements it is using functions.

P10: Additional FEP prioritization (function level): this technique is similar to Additional FEP prioritization with only difference that instead of using statements it is using functions.

P11: Total Fault Index (FI) prioritization: fault proneness is a measurable software attribute which is used for this technique. Some functions are likely to contain more faults than others, so the fault index is generated using following steps: (1) a set of measurable attributes [15] for each function. (2) the metrics are standardized. (3) principal component analysis [16] which reduces the set of standardized metrics. (4) finally they are combined to a linear function to obtain one fault index per function.

Now for each test case all the fault indexes for every function are added to get total fault index for each test case. Then sort the test cases in decreasing order of these sums to get the result for Total Fault Index (FI) prioritization.

P12: Additional Fault Index (FI) prioritization: as Total function coverage prioritization is extended to Additional function coverage prioritization similarly the Total Fault Index (FI) prioritization is extended to Additional Fault Index (FI) prioritization.

P13: Total FI with FEP coverage prioritization: this technique combines both Total FI and FEP coverage prioritization to achieve a better rate of fault detection.

P14: Additional FI with FEP coverage prioritization: as Total function coverage prioritization is extended to Additional function coverage prioritization similarly Total FI with FEP coverage prioritization is extended to Additional FI with FEP coverage prioritization.

P15: Total Diff prioritization: this technique is similar to Total Fault Index (FI) prioritization with the difference that Total FI prioritization require collection of metrics whereas Total Diff prioritization require only the calculation of syntactic differences.
between the program and the modified program. Diff means that only syntactic differences are given consideration.

P16: Additional Diff prioritization: Total Diff prioritization is extended to Additional Diff prioritization in a similar way as Total function coverage prioritization is extended to Additional function coverage prioritization.

P17: Total Diff with FEP prioritization: is exactly similar to Total FI with FEP coverage prioritization, except that it is dependent upon changed data derived from diff.

P18: Additional Diff with FEP prioritization: Total Diff with FEP prioritization is extended to Additional Diff with FEP prioritization in a similar way as Total function coverage prioritization is extended to Additional function coverage prioritization.

Test Suite Reduction[12]

The test-suite reduction problem may be stated as follows:

"Test suite T, a set of test-case requirements r1, r2, ..., rn that must be satisfied to provide the desired test coverage of the program, and subsets of T, T1, T2, ..., Tn, one associated with each of the ris such that any one of the test cases belonging to Ti can be used to test ri."

Step 1: Decompose and Reduce System Version n.

Construct the decomposition slices for the system under consideration. Note that the decomposition slice graph does not need to be constructed (although it would be nice to have); we only need the decomposition slices and the slices equivalences.

Step 2: Match Tests with Code. Use techniques like those of Vokolos and Chen to connect test cases to decomposition slices.

More precisely, the problem may be stated as follows:

Given a program, P, its regression test set, T, and a “new” program version, P’, nd, T’, the minimal subset of T necessary to revalidate the part of P’’s functionality that is inherited from P.

In other words, if P and P’ may be viewed as functions over sets of test cases, then determine the smallest subset, T*, of T, such that:

P’(T*) = P(T’) => P’(T) = P(T)

Note that this yields a decomposition of the test cases, just as the slices form a decomposition of the software; an amalgamation of equivalent test cases are also obtained by matching the test cases with the equivalent decomposition slice cluster.

Step 3: Decompose and Reduce System Version n+1.

As in step 1. The systems will readily compare. Obtain the tests for decomposition slice clusters that remain unchanged.

Step 4: Use tests that remain after removing those obtained in step 3. Any tests for unchanged code are not needed.

Regression testing technique

Incremental regression testing[10]

In this method as suggested [10], researchers propose some methods using which the test cases in the regression test suite whose outputs may be affected by the changes to the program may be identified automatically. Only these test cases need to be rerun during regression testing. Furthermore, the bulk of the cost of determining these tests is relegated to off-line processing, as depicted in Figure 2. They refer to the problem of determining the test cases in a regression test suite on which the modified program may differ from the original program as the incremental regression testing problem. The solution to this problem requires that the answer be determined solely on the basis of the analysis of the original program, the modifications, the regression test cases, and the original program's execution on these test cases and the modified program may not be executed on any test case.
1. If a statement is not executed under a test case, it cannot affect the program output for that test case.
2. Not all statements in the program are executed under all test cases.
3. Even if a statement is executed under a test case, it does not necessarily affect the program output for that test case.
4. Every statement does not necessarily affect every part of the program output.

**Behavioral Regression Testing**

Regression testing that relies only on existing test suites can result in limited checking of the changed code because of one of two issues, or both: (1) the lack of test cases that exercise a changed behavior; (2) the lack of an oracle that can identify such changed behavior. To address these issues, in this paper we propose **Behavioral Regression Testing (BERT)**, a novel approach that is meant to complement existing regression testing techniques. The goal of BERT is to accurately and automatically identify behavioral differences between two versions of a program by means of dynamic analysis. Given information on which parts of the code have changed between $P$ and $P'$, BERT operates in three main phases. In the first phase, BERT leverages automated test generation techniques to create a large number of test cases targeted at each of the changed classes. In its second phase, BERT considers each changed class $c$ and each test case $t$ created for $c$, runs $t$ on the old and new versions of $c$, and compares the outcome of $t$ in the two cases. The technique performs this comparison by checking several aspects of the test executions: the state of $c$ after the execution of $t$, the values returned by the methods of $c$ invoked by $t$, and the various outputs produced by $t$. Finally, in its third phase, BERT analyzes any difference in test outcomes identified in the previous phase to abstract away some of the information and factor together related differences.

```java
public class SavingAcc {
    private double balance;
    public boolean deposit(double amount) {
        01 if (amount > 0.00) {
            02 balance = balance + amount;
            03 return true;
        } else {
            04 System.out.println("amount cannot be negative");
            05 return false;
        }
    }
    public boolean withdraw(double amount) {
        06 if (amount <= 0) {
            07 System.out.println("amount cannot be negative");
            08 return false;
        }
        09 if (isOverdraft) {
            10 System.out.println("account is overdraft");
            11 return false;
        }
        12 balance = balance - amount;
        13 if (balance < 0) {
            14 isOverdraft = true;
        }
        15 return true;
    }
}
```

**Figure 1: Version V 0 of the bank account example.**

Before presenting the details of our technique, we introduce a small example that we use in the rest of the paper to show the limitations of existing regression testing approaches, motivate behavioral regression testing, and illustrate our approach. The example consists of a single class, SavingAcc, which implements the main functionality of a bank account and that we assume to be part of a larger bank management system. Figures 1 and 2 show the code of two consecutive versions of the class. Class SavingAcc contains two methods: deposit and withdraw. Method deposit is the same in V 0 and V 1. It allows for depositing funds in the account. When called, the method first checks whether the deposit amount is positive. If so, it adds amount to field balance and returns true; otherwise, it leaves the account balance unchanged, prints an error message, and returns the value false. Method withdraw allows for withdrawing funds from the bank account and is different in the two versions. In V 0, the method first checks whether the withdrawal amount is negative. If so, it prints an error message and returns false. Otherwise, it checks the value of balance. If balance is negative, it reports that the account is overdraft and returns false. Conversely, if balance is positive, the method subtracts the amount from the account balance and returns a value true. Assume that the developers decide to make the overdraft status of the account explicit. To this end, they make three changes to class SavingAcc, which are shown in boldface font in Figure 2. First, they add a boolean field, isOverdraft, which keeps track of whether the account is in an overdraft state. Then, they modify the conditional at Line 9 of method withdraw so that it checks the value of field isOverdraft instead of balance. Finally, they add to method withdraw instructions to set isOverdraft to true if the balance becomes negative (Lines 13–14). Although these changes to method withdraw are correct, there is a fault in the new version of the code. The developers forgot to reset the value of field isOverdraft when a deposit causes the balance to become positive after an overdraft. The practical effect of this omission fault is that an account that reaches an overdraft state will never leave it. To be able to identify the regression fault introduced in version V 1 of SavingAcc, a regression test suite would need to contain a test case that (1) performs a withdraw that causes the account to enter an overdraft state, (2) performs a deposit that causes the account to exit the overdraft state, (3) performs a withdraw with
an amount greater than zero, (4) checks whether the last withdraw was successful. Figure 3 shows a possible test case that would satisfy these requirements.

```java
public void testBehavioralDifference()
{
    SavingAcc acc = new SavingAcc();
    acc.deposit(10.00);
    acc.withdraw(20.00);
    acc.deposit(50.00);
    boolean result = acc.withdraw(20.00);
    assertEquals(result, true);
}
```

**Scenario-Based Functional Regression Testing[13]**

Most of the existing regression test selection techniques are code-based using program slicing, program dependence graphs, data flow and control flow analysis. Here the author presents a test scenario-based methodology for functional regression testing in the context of End-to-End (E2E) integration testing, which focuses on the functional correctness of large integrated system from the end users’ point of view. Different to program slicing, which is performed on source code, test scenario slicing in this paper is performed on test scenarios and enables global change impact analysis based on dependency and traceability analysis. Using test scenario slicing level-by-level, Ripple Effect Analysis technique directs the iterative process of regression test selection and revalidation.

**Test Scenario Slicing**

Definition 1. Slicing Criterion: A slicing criterion is a 2-tuple, denoted as ? | ?, meaning that upon the given attribute ? , slice corresponding attribute ?. The attributes in the two fields are identical, except that when a tester analyzes the input/output dependence, where the input attribute and output attribute flip over in the two fields.

Definition 2. Slice: Given test scenario s, and slicing criterion ls. ? is. ? i (s ? si), search scenario(s) si that match with the given scenario’s attribute s. ? . The results of a scenario slice are a set of scenario(s), which may full-match, or semi-match the slicing criterion, dependent on that the slicing criterion is a simple attribute or a complex attribute.

**Ripple Effect Analysis**

Ripple Effect Analysis (REA) is used to analyze and eliminate side effects due to changes and to ensure consistency and integrity after changes are made to software. It is an iterative process of change request, software modification, impacts identification and validation. It ends when there are no more ripples. The notion of ripple effect analysis, originally studied in the context of software programs, can be extended to all software artifacts including specifications, design, and test cases. Also, it can be extended to different artifacts across different phases of the software life cycle. In particular, the REA process is not specific to any particular programming language or design paradigm.

In this research, two perspectives characterize the REA process:

?? Keep consistence among scenarios, software components, and requirements, no matter where the change are initiated. Whenever there is an inconsistence, modifications and impact analysis are required.

?? At each iterate of the process, impacts are identified and validated using test scenario slicing with various slicing criteria. The following steps illustrate the characterized REA process for regression test selection.

1. Associate a change request, wherever it is originated from, with a set of test scenarios. If changes come from requirements and implementations, a tester can identify the test scenarios through the traceability analysis.
2. Trace each of the test scenarios identified in step 1 to software components through the traceability analysis.
3. Identify and revalidate the affected software components correspondingly.
4. Revalidate the test scenarios identified with respect to inputs, outputs, conditions, test data and configurations. For each of the test scenarios that need to be changed, check out the corresponding test cases.
5. For each of the test scenarios, determine slicing criteria based on its changes. Perform slicing to identify the potentially affected test scenarios with the criteria. The following are some policies for determining slicing criteria:

?? Slicing with all of the defined slicing criteria. This is to identify all the dependent test scenarios.

?? Slicing with only some slicing criteria. If an experienced test engineer is sure that changes on some parts of the test scenario will not affect the other parts and will not propagate along certain directions, he can perform impact analysis with a subset of slicing criteria. For example, suppose only the inputs of a test scenario need to be changed such as a layout of the interface, the tester may slice with only input slicing criterion and backward input/output slicing criterion.

?? Slicing by whole set of attribute values. This is to select all the test scenarios that are dependent with respect to the attribute.

?? Slicing by a subset of values. This is to select all the test scenarios that are dependent with respect to some specific aspects of the attribute.

?? Hybrid approach. For some test criteria, the tester may perform slicing by whole set of attribute value, while for the others, slicing by a selected subset of attribute values.

6. If there is any scenario identified in step 5, go to step 2; otherwise prepare for the following regression testing. The results of this process are:

a. A set of modified software components;

b. A set of affected test scenarios; and

c. A set of test cases corresponding to the affected test scenarios.

**Hybrid Regression Testing**

Incorporating other testing criteria, this subsection also provides the following hybrid strategies for selecting and scheduling regression tests to balance cost and reliability.

?? Risk-Based Strategy: Selects high-risk test scenarios, which have high probabilities to fail and/or cause serious consequences in case of fail, and to exercise them with high priority.

?? Usage-Based Strategy: Selects highly used test scenarios as candidates for regression testing. This strategy has been advocated in the Cleanroom methodology [15].

?? Random Testing: Selects a sample set of test scenarios. If the sample size is large enough, it is possible to achieve reasonable statistical confidence. This is the basis for reliability testing and assurance based testing.

?? Time-Wise Strategy: Regression testing may be performed at different stages of software development, such as daily build, weekly build, milestone, and major release.

**Regression Testing Using Slicing[14]**

Now we discuss about regression testing using the concept of a program slice. A backward program slice at a program point P for variable v consists of all statements in the program, including conditionals, that might affect the value of v at p, whereas a forward program slice at a program point p for variable v consists of all statements in the program, including
conditionals, that might be affected by the value of \( v \) at \( p \). The technique uses two slicing algorithms to determine directly and indirectly affected def-use associations.

The slicing algorithms are efficient in that they detect the def-use associations without requiring either the data flow history or the complete recomputation of data flow for the entire program. These algorithms are based on the approach taken by Weiser\cite{20} that uses the control flow graph representation of the program and only requires the computation of partial data flow information. Unlike previous regression testing techniques that require either a test suite or data flow information to select tests for regression testing, this technique explicitly identifies all affected def-use associations. Thus, the technique requires neither a test suite nor complete data flow information to enable selective retesting. If a test suite is maintained, fewer tests may be executed since only those tests that may execute affected def-use associations are rerun.

**Background**
This section overviews data flow testing, the basis of the regression testing technique. The technique also uses control dependence information to identify which affected def-use associations in a program to retest. Thus, this section also briefly discusses control dependence.

**Data Flow Testing**
Several data flow testing techniques\cite{6, 12, 13} have been developed to assist in detecting program errors. All of these techniques use the data flow information in a program to guide the selection of test data. Traditional data flow analysis techniques\cite{1}, based on a control flow graph representation of a program, are used to compute def-use associations. In a control flow graph, each node corresponds to a statement and each edge represents the flow of control between statements. Definitions and uses of variables are attached to nodes in the control flow graph, and data flow analysis uses these definitions and uses to compute def-use associations. Uses are classified as either computation uses (c-uses) or predicate uses (p-uses). A c-use occurs whenever a variable is used in a computation statement; a p-use occurs whenever a variable is used in a conditional statement. Def-use associations are represented by triples, \((s, u, v)\), where the value of variable \( v \) defined in statement \( s \) is used in statement or edge \( u \). In Figure 1, definitions are shown with the variable on the left side of the assignment and uses are shown with the variable on the right side of an assignment or in a predicate. In the figure, node 7 contains a c-use of the definition of \( X \) in node 4. The triple \((4, 7, X)\) represents this def-use association. Node 7 also contains a c-use of the definition of \( X \) in node 6, and the triple \((6, 7, X)\) represents this def-use association. Figure 1 contains a p-use in node 5 of the definitions of \( A \) in nodes 2 and 3, and there are def-use associations between the definitions in nodes 2 and 3 and each of the edges leaving conditional node 5; triples \((2, (5,6), A)\), \((2, (5,7), A)\), \((3, (5,6), A)\) and \((3, (5,7), A)\) represent these def-use associations.

Test data adequacy criteria are used to select particular def-use associations to test. One criterion, alldupaths, requires that each definition of a variable be tested on each loop-free subpath to each reachable use. Another criterion, alldupaths, requires that each definition of a variable be tested on some subpath to each of its uses. Other criteria, such as all defs, require that fewer def-use associations be tested. For a complete discussion of data flow testing, see references\cite{6, 16}.

**Definition-c-use Associations**
\((4, 7, X)\), \((6, 7, X)\)

**Definition-p-use Associations**
\((2, (5,6), A)\), \((2, (5,7), A)\), \((3, (5,6), A)\), \((3, (5,7), A)\)

**Control Dependence**
To identify def-use associations that may be affected by a program change, the slice-based approach uses control dependence information. Informally, a node (statement) in a control flow graph is control dependent on another node (statement) \( Y \) if there are two paths out of \( Y \), such that one path necessarily reaches \( X \) and the other path may not reach \( X \). This definition of control dependence, given by Ferrante, Ottenstein and Warren\cite{5}, is essentially the same as the definition of direct strong control dependence given by Clarke and Podgurski\cite{15}.

In Figure 1, the execution of statement 2 depends on statement 1 evaluating to \( true \), whereas the execution of statement 3 depends on statement 1 evaluating to \( false \). Thus, statements 2 and 3 are control dependent on statement 1. Also, statements 6 and 7 are control dependent on statement 5. However, statements 1, 4, and 5 are only control dependent on reaching the point immediately before statement 1.

**Detecting Affected Definition-Use Associations**
To satisfy a data flow testing criterion after making a program change requires identifying the def-use associations affected by the change. This section discuss the different types of affected def-use associations. Then, it discusses the slicing algorithms for forward and backward walks that identify the definitions and uses that are affected by a program edit. Next, the section presents the algorithm to handle the different types of program edits, and finally, it discusses the way in which higher level edits are translated to lower level edits for processing.

**Types of Affected Def-Use Associations**
Affected def-use associations fall into two categories: (1) those affected directly because of the insertion/deletion of definitions and uses (new associations) and (2) those affected indirectly because of a change in either a computed value (value associations) or a path condition (path associations).

**New Associations:** A program edit creates new def-use associations that must be tested. For example, consider the following code segment:

1. if \( A > 1 \) then
2. \( Y := X + 5 \)
3. else
4. \( Y := X - 5 \)
5. endif
6. \( X := 2 \) /* replace with “X := 2 + Y” */

If statement 6 is replaced with “X:=2+Y”, a new use of variable Y is introduced. Def-use associations consisting of those definitions of X that reach the new use of Y must be tested. These new associations are (2, 6, Y) and (4, 6, Y).

Value Associations Value associations are defuse associations whose computed values may have changed because of the program edit and therefore, require retesting. For example, consider the following code segment:

1. \( X := 2 \) /* replace with “X := 3” */
2. if \( A > 1 \) then
3. \( Y := X + 5 \)
4. else
5. \( Z := X - 5 \)
6. endif
7. \( T := Y + 6 \)
8. \( U := Z + B \)

If statement 1 is replaced with “X := 3”, no new def-use associations are created. However, the def-use associations that depend on the new value of X are retested since they are affected by the change. Since both statements 3 and 5 use the definition of X in statement 1, def-use associations (1, 3, X) and (1, 5, X) are value associations. The new value of X in statement 3 affects the computed value of Y at that statement, which causes value association (3, 7, Y) to be identified. Likewise, value association (5, 8, Z) is found because of the affected value of Z in statement 5. Now, the values of T in statement 7 and U in statement 8 are affected, and the process of identifying value associations continues with uses of these variables.

Path Associations The def-use associations that are affected on a path whose path condition has changed must be retested. A path condition can be altered because of an explicit change in an operator in the predicate statement. For example, consider the following code segment:

1. \( X := 2 \) /* replace with “X := 3” */
2. if \( A > 1 \) then /* replace with “A < 1” */
3. \( Y := X + 5 \)
4. endif
5. if \( X > A \) then
6. \( Y := X - 5 \)
7. endif
8. write Y

If statement 2 is changed to “if A < 1”, then no new def-use associations are created, but any def-use association that is control dependent on statement 2 may be affected by the change. Here, statement 3 is control dependent on statement 2, and so path association (3, 9, Y) is identified.

A path condition can also be altered because of a change in the value of a p-use in a predicate. For example, in the above code segment, if statement 1 is replaced with “X := 3”, then the value computed in statement 6 is affected, and the path condition to statement 7 is changed. Thus, path association (7, 9, Y) is found.

Backward and Forward Walk Algorithms

Algorithms for backward and forward walks identify the definitions and uses that are affected by a program edit. Both algorithms use a control flow graph representation of the program in which each node represents a single statement. These algorithms compute data flow information to identify affected def-use associations but require no past history of data flow information. Furthermore, the algorithms are slicing algorithms in that they examine only relevant parts of the control flow graph to compute the required data flow information. These algorithms are designed based on the approach taken by Weiser for computing slices[20]. This approach lets relevant program slices be computed without exhaustively computing the def-use information for the program. This discussion assumes that only scalars are being considered; the technique is easily extended to include arrays by adding a new condition for halting the search along paths.

6.4.7. Result of forward and backward walk

\[
\text{BackwardWalk(7,\{X\})}
\]

Variable use def-use associations

use of X at statement 7 (6,7,X), (4,7,X)

ForwardWalk((2,Y), false) variable def-use association path taken

Y (2,4,Y) 2,3,4
J (4,10,J) 4,9,10
ForwardWalk((3,X),(3,Y), true) variable def-use association path taken

X (1,6,X) 3,5,6
Y (2,4,Y) 2,3,4
J (4,10,J) 4,9,10
Z (7,8,Z) 7,8
(7,10,Z) 7,8,9,10

Conclusion:

We have presented a selective regression technique based on a function dependence relationship. Our approach properly uses a high level of abstraction instead of involving the complex relationship among statements. Therefore, it can achieve a good compromise between efficiency and precision. Moreover, we use the Function Calling Graph to represent the execution history of each test case, which can be processed efficiently and precisely. Our method can also provide useful information to facilitate the designing of new test cases.

References

[1]. A survey of software testing methodology, Tao Feng, Kasturi Bidarkar, University of Wisconsin-Madison/Department of Electrical and Computer Engineering, 1415 Engineering Drive, Madison WI 53706 USA

[2]. A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments, Jung-Min Kim Adam Porter, Department of Computer Science, University of Maryland, College Park, MD 20742, USA


[11] Reducing Regression Test Size by Exclusion, Keith Gallagher, Tracy Hall, Sue Black


