CHAPTER 1
INTRODUCTION

1.1 REGRESSION TESTING

Class-based object-oriented programming languages encourage developers to implement a software system as a collection of classes and objects. Some languages such as Java (Gosling et al 1996) and Eiffel (Meyer 1992) enforce pure object-oriented implementation, whereas C++ (Stroustrup 1997) supports a multiparadigm approach where parts of the implementation may be, for example, in procedural style. Without getting into a debate about which approach is preferable, this thesis assumes that the implementation is class-based.

Every good software evolves. Evolution implies that the modified version will be different than the original version in some ways; this difference need not necessarily imply changed functional behavior. The following are the most common reasons for evolution (for example, see McConnell 1993):

- **Bugs are fixed.** Bugs may be discovered in the current release by developer, tester, or the user and will be fixed by appropriately modifying the source code.

- **Performance is enhanced.** Sometimes performance is found to be unacceptable and hence critical portions of the code may be re-written to achieve the desired performance.

- **Code restructuring is performed.** For reasons of maintenance and code reuse, the developer may restructure his code by making changes to existing code. Some examples include changing compositional design to inheritance-based one, re-writing using a well-known design pattern, and changing the access rights of class members.
- **Functionality is altered.** Due to changes in functional requirements, some existing features may be modified. An example is a change in user interface where a modal dialog box is replaced by a nonmodal one.

- **Functionality is enhanced.** The client may require additional functionality in the software.

- **New technology is adopted.** A decision may be taken to reengineer the software to adopt a new technology. One example is to switch from a simple two-tier client-server architecture to a three-tier architecture.

Assuming that the original version was adequately tested, two questions arise:

1) Should new test cases be generated?

2) Should the new version be tested by running the previous test suite?

Answers to these questions depend on the nature of changes causing the program to evolve. Supposing it is known (and can be established) that the only changes made to the program are in the nature of bug fixes, not functionality enhancements, then one can make a case against expending effort in defining (and running) new test cases. What about the other question: Is it necessary to rerun the previous test suite? The typical answer to that question is in the affirmative. The reason is the belief that new bugs may have been introduced in the modified code. Therefore, software testers are faced with the task of regression testing: the process of retesting software after a modification. The aim of regression testing is to provide confidence that the behavior of modified code is as expected and that the unmodified code is not adversely affected.

Why can't the tester decide in favor of retesting the modified software after every modification? In other words, why is regression testing an issue at all? It is, because the process of regression testing may be costly: there may be several thousand test cases that need to be run under identical (controlled) environment requiring substantial effort (resources and time). Research in the area of regression testing has
focused on techniques for minimizing regression test effort. These techniques perform an analysis of the different procedures in the original and changed programs to select portions of the test suite appropriate for the changed version.

Although a lot of research work has been done in the area of software testing, comparatively less work has been done in regression testing. There is even less research related to regression testing object-oriented software. The research reported in this thesis is about minimizing the effort in regression testing object-oriented software by taking advantage of specifics of the implementation language.

Section 1.2 describes some existing approaches to regression testing. Section 1.3 discusses the limitations of existing techniques and section 1.4 introduces the research proposed in this thesis. Section 1.5 summarizes the contributions of this thesis and section 1.6 outlines the thesis structure.

1.2 SURVEY OF EXISTING TECHNIQUES

There are primarily two strategies for regression testing:

1) The retest all strategy reruns every test in the initial test suite. This approach can be impractical as it may require prohibitive amounts of effort and time.

2) The regression test selection strategy reruns only a subset of the initial test suite. This approach may be feasible if the time and effort expended in test selection is less than that required to run the eliminated tests.

Graves et. al. (1998) suggest the following procedure for regression testing:
Let P be a procedure or program, let P' be a modified version of P, and let T be a test suite for P.

1. Select T' ⊆ T, a set of tests to execute on P'.
2. Test P with T', establishing P'’s correctness with respect to T'.
3. If necessary, create T'', a set of new functional or structural tests for P'.
4. Test P' with T'', establishing P''s correctness with respect to T''.
5. Create T''', a new test suite and test history for P' from T, T', and T''.

Step (1) involves the regression test selection problem of selecting a subset T' of T with which to test P'. Step (3) addresses the coverage identification problem of identifying portions of P' requiring additional testing. Steps (2) and (4) address the test suite execution problem of efficiently executing tests and checking the results for correctness. Step (5) address the test suite maintenance problem of updating and storing test information.

Rothermel and Harrold (1996) provide a detailed comparison of the different regression test selection techniques for procedural-language software. In the following, sections 1.2.1 thru 1.2.12 provide an overview of these, based on Rothermel (1996). Section 1.2.13 briefly describes a technique proposed by Rothermel and Harrold (1994) for object-oriented software.

1.2.1 Linear equation techniques

Linear equation techniques (Fischer 1977) use systems of linear equations to express relationships between tests and program segments. The techniques obtain systems of equations from matrices that track program segments reached by test cases, segments reachable from other segments, and def-use information about the segments. The intraprocedural techniques use a 0-1 integer programming algorithms to identify a subset T' of T to be applied on P'.
1.2.2 Symbolic execution technique

This technique (Yau and Kishimoto 1987) uses input partitions and data-driven symbolic execution to select and execute regression tests. First, the technique analyzes code and specifications to derive the input partition for a modified program. Second, the technique eliminates obsolete tests, and generates new tests to ensure that each input partition is exercised by at least one test. Given information on where code has been modified in the new version, the technique determines edges in the control flow graph for the new program from which modified code is reachable. The tests are then symbolically run.

1.2.3 Path analysis technique

The path analysis technique (Benedusi et al 1988) takes as input the set of program paths in \( P' \) expressed as an algebraic expression, and manipulates that expression to obtain a set of cycle-free exemplar paths: acyclic paths from program entry to program exit. The technique then compares exemplar paths from \( P \) to that from \( P' \). Next, the technique analyzes tests to see which exemplar paths they traverse in \( P \). The technique selects all tests that traverse modified exemplar paths.

1.2.4 Dataflow techniques

Dataflow test selection techniques identify def-use pairs that are new in, or modified for, \( P' \), and select tests that exercise these pairs. Some techniques take into account def-use pairs that have been deleted from \( P \). Incremental techniques process a single change, select tests for that change, incrementally update dataflow information and test trace information, and then repeat the process for the next change. Nonincremental techniques consider all modifications simultaneously. This technique in its variations is discussed in Harrold and Soffa (1998, 1989, 1989 a ) and Ostrand and Weyukar (1988).
1.2.5 Program dependence graph techniques

Techniques based on PDG (Bates and Horwitz 1993) use slicing to group PDG components (nodes or flow edges) in $P$ and $P'$ into execution classes, such that a test that executes any component in an execution class executes all components in that class. Next, the techniques identify components that may exhibit different behavior in $P'$ than $P$ by comparing slices of corresponding components in $P$ and $P'$. Finally, the techniques select all tests that exercise components that are in the same execution class as an affected component.

1.2.6 System dependence graph techniques

Binkley et al(1995, 1997) presents a technique for interprocedural regression test selection that operates on the system dependence graph (SDG). Given a program $P$ and its modified version $P'$, this technique uses calling context slicing on SDG for $P$ and $P'$ to identify components that have common execution patterns. The technique identifies new, preserved, deleted and affected components in $P'$ and selects test cases that exercise components in $P$ that have common execution patterns with respect to new or affected components in $P'$.

1.2.7 Modification-based technique

This technique (Sherlund and Korel 1991) differs from the rest of the techniques in that it does not automate the process of selecting $T'$ from $T$. Instead, the technique identifies coverage requirements. The Modification-based technique uses static dependence analysis to determine program components that are data or control dependent on modified code, and thus may be affected by any modification. The technique instruments the modified program, and requires a tester to run the instrumented code. It then performs a dynamic dependence analysis using the execution trace, and detects whether the test executed the new version.
1.2.8 Firewall technique

This selective retest technique proposed by Leung and White (1990) handles both code and specification changes. It determines where to place a firewall around modified code modules and selects unit tests for modified modules that lie within the firewall, and integration tests for groups of interacting modules that lie within the firewall.

1.2.9 Cluster identification technique

Laski and Szermer (1992) present a technique for identifying single-entry, single-exit subgraphs of a control flow graph called clusters that have been modified from one version of a program to the next. The technique computes control dependence information for a procedure and its changed version, and then computes the control scope of each decision statement in the procedure by taking a transitive closure of the control dependence relation. This information is used to identify clusters and establish a correspondence between the CFGs of P and P'. In the process, the technique selects tests that execute new, deleted and modified clusters.

1.2.10 Slicing techniques

Agrawal et al (1993) define a family of selective retest techniques that use slicing. For each test t in T, each technique constructs a slice. The authors discuss four different slice types: execution slice, dynamic slice, relevant slice, and approximate relevant slice. An execution slice for t contains exactly the same statements in P that were executed by t. A dynamic slice for t contains all statements in the execution slice that have an influence on an output statement in the execution slice. A relevant slice for t is like the dynamic slice, except that it also contains predicate statements in t that, if changed, may cause P to produce different output, and statements in t on which these predicates are data dependent. Finally, an approximate relevant slice for t is like the dynamic slice, except that it also contains all predicate statements in the execution slice for t. Given slice sl for test t, constructed by one of the four slicing techniques, if sl
contains a modified statement, the techniques select t. Gupta et al (1992) discuss another slicing-based approach.

1.2.11 Graph walk techniques

Rothermel and Harrold (1997) present an intraprocedural regression test selection technique that builds control flow graphs for P and P', collects traces for tests in T that associate tests with CFG edges, and performs synchronous depth-first traversals of the two graphs, comparing nodes that are reached along prefixes of execution traces. When a pair of nodes N and N' in the graphs for P and P' respectively, are discovered, such that the statements associated with N and N' are lexically identical, the technique selects all tests from T that, in P, reached N.

1.2.12 Modified entity technique

Chen, Rosenblum and Vo (1994) present the modified entity technique, a regression test selection technique that detects modified code entities. Code entities are defined as executable portions of code such as functions, or as nonexecutable components such as storage locations. The technique selects all tests associated with changed entities.

1.2.13 Selecting regression tests for object-oriented software

Rothermel and Harrold (1994) have proposed a technique for selecting regression tests for object-oriented software. Their method constructs dependence graphs for classes (and derived classes) and programs that use classes. A procedure dependence graph (PDG) captures control and data dependence of a procedure. An interprocedural dependence graph (IPDG) represents a collection of PDGs and their contexts. The regression test selection algorithm uses the IPDGs of the certified (P) and modified (P') programs to determine code fragments that may cause P' to produce different output than P. Test history information containing execution trace in terms of various regions of the program dependence graph, is then used to select relevant tests that should be rerun on P'.
1.3 LIMITATIONS OF EXISTING APPROACHES

Of all the techniques mentioned in section 1.2, only that described in section 1.2.13 applies to object-oriented languages. Even this technique fails to explicitly address certain types of class evolutions peculiar to some languages. The PDG does not represent language-dependent information such as whether an instance variable is const (C++), or a member function is synchronized (Java). A study of C++ and Java reveals that these attributes have a role to play in deciding whether or not to retest a modified class. To illustrate this point, three cases are given below.

Case 1 (Java):

**Before change:**

```java
class A {
    private int val = 0;
    synchronized public void setValue(int v) {
        val = v;
    }
    synchronized public int getValue() {
        return val;
    }
}
```

**After change:**

```java
class A {
    private int val = 0;
    public void setValue(int v) {
        val = v;
    }
    public int getValue() {
        return val;
    }
}
```

In this example from Java, qualifying a method as synchronized is to make the class thread-safe. It can therefore be argued that the original class was thread-safe whereas the modified version is not. This implies that the modified version needs to be...
retested in the context of multiple threads. Rothermel and Harrold technique does not handle this case correctly since it would suggest no retest.

Case 2 (C++):

**Before change:**
```cpp
class A {
    int val;
public:
    A(int v) : val(v) { }
    int ff() {
        return val + 89;
    }
};
```

**After change:**
```cpp
class A {
    const int val;
public:
    A(int v) : val(v) { }
    int ff() {
        return val + 89;
    }
};
```

In this C++ example, Rothermel and Harrold technique would suggest retesting using tests from the earlier suite that exercised A::ff() since an instance variable used by the method has been changed. However, it is easy to see that there is no need to retest the method since making the data member `const` does not alter the meaning of the method.

Case 3 (C++ and Java):

**C++ Version -1:**
```cpp
class Base {
public:
    void f() { g(); }
    virtual void g() { }
};
```
class Derived : public Base {
public:
  void g() { }
};
main() {
  Derived d;
  d.f(); // Invokes Derived::g()
  return 0;
}

Java Version - 1:
class Base {
  public void f() { g(); }
  public void g() { }
}
class Derived extends Base {
  public void g() { }
}
class JavaTest {
  public static void main(String args[]) {
    Derived d = new Derived();
    d.f(); // Invokes Derived::g()
  }
}

If the Base class undergoes a minor change in version two as shown below, how does it impact the behavior of the application?

C++ Version - 2:
class Base {
public:
  void f() { g(); }
private:
  virtual void g() { }
};

Java Version - 2:
class Base {
  public void f() { g(); }
}
private void g() { }
}

As can be observed, the only change in the modified version is that a public method has been made private. The surprising result is that there is no change in the C++ version, but in Java, the call to g() in f() gets statically bound to the one defined in Base! This implies that a somewhat trivial change in source code may force regression testing due to changed behavior. Rothermel and Harrold technique does not address this aspect of language dependency.

The technique put forth in this thesis addresses these issues. The general conclusion from these examples is that a technique that is language independent may not be able to minimize retest effort to the desirable extent. Rothermel and Harrold (1996, 1994) use two terms Precision and Safe in this context. Precision measures the ability of a technique to avoid choosing tests that will not cause the modified program to produce different output than the original program. A technique is Safe if it selects all tests from the original test suite that could possibly exhibit different output when run on the modified program. They point out that no technique can both be safe and precise. What is desirable in the context of regression testing is a strategy that is safe and at the same time chooses a minimum number of tests from the original test suite.

1.4 A LANGUAGE-BASED APPROACH

The research described here differs significantly from the current approaches to regression testing. Whereas the others are language-neutral, the proposed technique relies on features particular to the object-oriented language to minimize retest effort. The motivation for this approach comes from the following observations:

• object-oriented languages differ from one another in terms of their feature set, and

• language features directly influence the need for retesting

The research described here consists in studying how a program written in an object-oriented language evolves across versions and then reasoning about the
impact on regression testing based on the evolution pattern. This is accomplished by comparing two versions of a class and determining which elements of the modified software require retesting. How to retest is the primary concern of other existing techniques.

In order to convey the essence of the proposed approach, three of its key concepts are briefly outlined in the following sections. These are further elaborated in the subsequent chapters.

1.4.1 Atomic change:

An atomic change is the fundamental language-specific unit of change in an evolving object-oriented program (Rangarajan and Eswar 1998 a). Every class that changes across versions can be described as a permutation of these units.

For example, consider:

```cpp
class X {
private:
    int a;
    int f();
    void g() {
    
public:
    X(int val) : a(val) {
    void h() {
        g(f());
    
    }
    
    int X::f() {
        return a;
    
    
    
    
Assume that in the next version this class evolves to

```
void h() {
  g( f() );
}
private:
  float fl;
  int f();
  void g() { }
};

int X::f() {
  return a;
}

This evolution can be described as a combination of the following atomic changes:

a) Introduce a new data member `float fl` in the `private` section
b) Change the access level of data member `a` from `private` to `public`
c) Make the data member `a` const
d) Reorder `public` and `private` sections

The primary reason to characterize class evolution in terms of atomic changes is that they have an impact on retesting the class. Atomic changes exhibit the following properties:

1) They are language dependent
2) They are finite in number
3) Some of them preserve class equivalence

The last property is particularly useful in regression testing. Some of the atomic changes do not alter the behavior of the evolving class and hence retesting such a class is not necessary. For instance, consider the following C++ class:

class A {
  private:
    int value;
};
public:
    int getValue() {
        return value;
    }
    A(): value(66) {
    }
};

If this class is changed to

class A {
public:
    int getValue() const {
        return value;
    }
    A(): value(66) {
    }
private:
    const int value;
};

the modified class is behaviorally identical to the original one and hence need not be retested. The following atomic changes are involved in the evolution of the class:

1) Change the order of class members (retesting not needed)
2) Make a data member const (retesting not needed)
3) Make a member function const (retesting not needed)

These three atomic changes, when applied to any class, preserve the behavior of the class and hence such a class need not be tested again after change. In general, whether or not an atomic change will require retesting the evolved class depends on the atomic change and the evolution context. There are three possibilities:

- atomic change requires no retest
- atomic change requires retest
- atomic change may require retest

This thesis proposes a set called Lexical Function Context to derive clues about the need to retest.
1.4.2 Lexical function context

In the case of atomic changes that may require retesting an evolved class, additional clues to decide whether retesting is actually necessary are needed. Such clues are partly derivable from the evolution context by using the concept of *Lexical Function Context* (LFC). LFC of a class member function is the set of all symbols this function binds to *statically*. For example, consider the C++ code fragment shown below:

```cpp
extern void g(int);

class X {
private:
  int a;
  int f();
public:
  X(int val) : a(val) { }
  void h() {
    g(f());
  }
};

int X::f() {
  return a;
}
```

The LFCs of member functions of X are

- \( \text{LFC}(\text{X::void}) = \{ \text{X::int_a} \} \)
- \( \text{LFC}(\text{X::int_f_void}) = \{ \text{X::int_a} \} \)
- \( \text{LFC}(\text{X::void_h}) = \{ \text{X::int_f_void, ::void_g_int} \} \)

The symbol bindings are listed using a name mangling scheme (Appendix 1) that encodes the entity’s signature.

In the context of class evolution, if a member function’s LFC has changed, then that member function needs to be retested (the rule is slightly more detailed, and will be discussed in Chapter 2).
If the above class evolves to

```c++
extern void g(int);
class X {
private:
    int a;
    int f();
    void g(int i) { /* body */ }
public:
    X(int val) : a(val) { }
    void h() {
        g(f());
    }
    int X::f() {
        return a;
    }
}
```

The LFCs of member functions of X are:

- \( \text{LFC}(X::\text{void}) = \{ \text{X::int}_a \} \)
- \( \text{LFC}(X::\text{int}_f:\text{void}) = \{ \text{X::int}_a \} \)
- \( \text{LFC}(X::\text{void}_h) = \{ \text{X::int}_f:\text{void}, \text{X::void}_g:\text{int} \} \)

Since \( X::h \) binds to a different function in the modified version, that method (and hence the class) needs to be retested.

### 1.4.3 Class equivalence categories:

Whether or not an atomic change applied to a class alters the behavior of the class can be answered based on the equivalence preserving property of the atomic change. This thesis proposes four types of equivalence categories namely, L, S, F, and T (Rangarajan and Balasubramaniam 1998).

**L-Equivalence:** Two classes \( C \) and \( C' \) are L-equivalent (L-Eq) if they are lexically equivalent. This essentially means that they are source equivalent. In other
words, running a file differencing utility on the source corresponding to these classes will reveal no differences, implying C and C' are one and the same.

S-Equivalence: Structural equivalence (S-Eq) may be determined by building an abstract syntax tree (AST) for each of the two classes C and C', reducing them to a canonical form, and comparing the trees. If the two ASTs are equivalent, the classes are S-equivalent. An atomic change is said to preserve S-eq if the modified class is S-eq to the original one. Changing the order of elements defined in a class is an example of such an atomic change.

T-Equivalence: A class C' is T-equivalent (T-Eq) to class C if and only if C' is not structurally equivalent to C and the test suite for C runs correctly with respect to C'. Indeed, there is no need to retest C'. The classes C' and C shown below are T-eq.

Class C:
```cpp
class SomeClass { 
public:
    SomeClass( int a, int b ) : i(a), j(b) { 
    }
    int getValue() { 
        return j+i;
    }
private:
    int i, int j;
};
```

Class C:
```cpp
class SomeClass { 
public:
    SomeClass( int a, int b ) : i(a), j(b) { 
    }
    int getValue() const { 
        return j+i;
    }
```
protected:
    const int i;
    const int j;
};

Here, C' differs from C as follows:

1) Data members are protected
2) Data members are const
3) Method getValue() is const

**F-Equivalence:** This is the last type of equivalence. Two classes C and C' are F-Equivalent (F-Eq) if and only if they are not T-equivalent but are functionally equivalent.

**Class C:**

```cpp
class SomeClass {
public:
    void ff( int val ) {
        if( val < 45 ) {
            cout << "Less than 45";
            return;
        }
        cout << "Greater than or equal to 45";
    }
};
```

**Class C':**

```cpp
class SomeClass {
public:
    void ff( int val ) {
        if( val >= 45 )
            cout << "Greater than or equal to 45";
        else
            cout << "Less than 45";
    }
};
```
1.5 KEY CONTRIBUTIONS

The following are the salient contributions of this thesis:

- a new, practical approach to regression testing object-oriented software using language-specific knowledge is suggested;
- the notion of class equivalence categories for OO languages is proposed;
- two popular OO languages namely, C++ and Java, have been studied in detail to document the atomic changes these languages permit;
- a new abstract representation that facilitates reasoning about OO programs is described; and
- finally, a tool called JRegCheck has been implemented to compare two versions of a Java program to enumerate the atomic changes across the versions and is being enhanced to suggest which parts of the modified software require regression testing.

There has been no other attempt to describe class evolution as a permutation of language-specific atomic changes. Whitmire (1997) describes a small number of atomic operations characterizing design changes. The changes proposed in this thesis are more fine-grained than his and occur at the programming level. Palay (1992) describes a C++ system that understands certain compatible changes to an evolving class in order to minimize recompilation. Yang (1991), Horowitz (1990) and Binkley (1997) describe techniques for determining differences between two versions of a procedural program. These differences, however, are not based on predefined categories.
1.6 ORGANIZATION OF THE THESIS

Chapter 2 describes the concept of atomic changes that govern the way a program written in an object-oriented language evolves across versions. Lexical Function Context is also described in detail in the context of atomic changes. The impact of these atomic changes with respect to *equivalence* between the original class and its evolved counterpart is discussed in Chapter 3. The equivalence categories L, S, T, and F and their inter-relationships are explained fully. Chapter 4 documents the different atomic changes possible in C++ and Chapter 5 covers these details for Java. A new abstract graphical representation that facilitates reasoning about classes is proposed in Chapter 6. Other applications of this research and possible future work are covered in Chapter 7. Appendix 1 outlines the name mangling scheme used for C++ programs while Appendix 2 describes the scheme used for Java. Appendices 3 and 4 provide a summary of the atomic changes discussed in detail in Chapters 4 and 5 respectively. Appendix 5 lists the sample output produced by *JRegCheck* on two java files.