CHAPTER 7
RESULTS, DISCUSSION AND CONCLUSION

This chapter presents the conclusions of this research and offers suggestions for further extensions. Certain interesting design differences between C++ and Java are also briefly discussed.

7.1 REVIEW OF CONCLUSIONS

This thesis has described investigation into selective regression testing using a model based on programming language. This model is built on the notion of atomic changes, which are the fundamental and syntactically acceptable units of code change. These atomic changes are language-dependent and have well-defined properties. One interesting property of some of these atomic changes is that they do not alter the behavior of a class significantly to warrant a retest. Hence, if a class evolves through a permutation of such neutral atomic changes, the evolved class need not be retested. To characterize the property that an evolved class may be equivalent to the original class, four types of equivalence categories namely, Lexical (L), Structural (S), Test (T), and Functional (F), have been proposed and discussed. If two classes are equivalent, the equivalence must belong to one of these four categories. It is shown that if an evolved class is L, S, or T equivalent to its original version, regression testing is not needed.

Two popular object-oriented languages in use today, C++ and Java, have been studied in detail and their respective atomic changes documented. Finally, a new representation called Class Annotated Graph has been proposed as an extension to existing representations in order to capture language-specific information of a given object-oriented program.

Further empirical investigation is required to determine the effectiveness of the language-based approach in minimizing regression testing in real-world applications. Such a study could not be undertaken as part of this research since programmers do not keep track of the atomic changes involved in the evolution, and
determining this manually is a formidable task. To this end, prototype of a tool called \textit{JRegCheck} that analyzes two Java programs and lists some of the atomic changes between them has been built. Sample output from this tool on two versions of a Java program is presented in Appendix 5. Work is in progress to enhance the tool to cover all possible atomic changes and to determine which methods of an evolved class require retesting. Such an automated tool will be of great value to developers and testers.

7.2 OTHER APPLICATIONS OF ATOMIC CHANGES

The primary use of atomic changes, as espoused in this thesis, is to minimize regression test effort. However, the author would like to suggest two other areas in which it may be possible to apply this notion. These require further study and validation.

One application is measuring the complexity of a programming language. This is a difficult task, since complexity is usually considered to be an external attribute, not directly measurable. It is generally accepted that a language is considered complex if, among other things, it supports many features. For example, C++ is considered more complex than C in view of its additional features to support object-orientation. It was pointed out in Chapter 2 that the number of atomic changes a language permits is proportional to the number of language features. Hence, if a language L1 has more number of atomic changes compared to another language L2, it is true that L1 has more features than L2, and hence is perceived more complex. It is the author's practical experience that programming in C++ is more involved than programming in Java. This fact is also corroborated by the number of atomic changes each permits (Java has 58, whereas C++ has 61).

Secondly, since atomic changes are the units of program change, it is now possible to study an evolving program to understand the types of changes made by programmers. Such a study can provide clues to developers as to where additional work in designing new test cases needs to be expended. This information may also prove useful in estimating program maintenance effort.
7.3 A COMPARISON OF C++ AND JAVA

In order to determine the applicability of the notion of atomic changes to object-oriented languages, two popular languages C++ and Java were studied in detail. During the course of the study, some interesting differences between the two languages came to light.

7.3.1 Atomic changes

The number of atomic changes in C++ (61) is only slightly more than that in Java (58). This is surprising since C++ is generally considered far more complex than Java in terms of the number of features supported. One reason appears to be the definition of the atomic change itself. In view of the requirement that the code should correctly build after applying an atomic change, certain low level programmatic changes do not qualify as atomic changes. For instance, replacing a pointer with a reference, though a valid code change in C++, cannot be an atomic change since the code will not compile after such a change. Consider the following C++ code fragment:

```cpp
class A {
    int *ival;
public:
    void func() {
        *ival = 90;
    }
};
```

If the instance variable is converted to a reference, the code becomes

```cpp
class A {
    int &ival;
public:
    void func() {
        *ival = 90;
    }
};
```

The code fragment after change is syntactically incorrect. Since the atomic changes are defined at a higher level of granularity, there is no substantial difference in the number.
7.3.2 Accessibility and dynamic binding

Whereas a virtual function can be private in C++, it cannot be in Java. Though this feature has been highlighted in the discussion of atomic changes in Java, it is worthwhile to recall it here. Consider

```java
class Base {
    private void f() { /* .. */}
    public void g() {
        f(); // Statically bound to Base.f()
    }
}
class Derived extends Base {
    private void f() { /* ... */} // Does not override
        // Base.f()
}
class Test {
    public static void main(String args[]) {
        Derived d = new Derived();
        d.g(); // Calls Base.f() and Base.g()
    }
}
```

If this code fragment is written in C++, the behavior changes.

```java
class Base {
    private: void f() { /* .. */}
    public: void g() {
        f(); // Dynamically bound to appropriate f()!
    }
};
class Derived : public Base {
    private: void f() { /* ... */ } // Overrides Base::f()!
};
main() {
    Derived d;
    d.g(); // Calls Base::g() and Derived::f()
}
```

The author personally believes that the C++ model is better than that of Java, since making a method private can have a subtle behavior change in the latter.
7.3.3 Overloading across scopes

The next difference concerns the scope rule in respect of overloading. C++ disallows overloading of functions across scopes, whereas Java permits it. First, consider a C++ program.

```cpp
class Base {
public:
    void f(int i) { /* ... */}
};
class Derived : public Base {
public:
    void f(const char *p) { /* ... */}
};

main() {
    Derived d;
    d.f("Hello"); // Calls Derived::f()
    d.f(99); // Syntax error!
}
```

Thus, `Base::f(int)` is hidden by `Derived::f(const char *)`. Now, consider the following version of Java code:

```java
class Base {
    public void f(int i) { /* ... */}
}
class Derived extends Base {
    public void f(const char *p) { /* ... */}
}
class Test {
    public static void main(String args[]) {
        Derived d = new Derived();
        d.f("Hello"); // "Calls Derived.f()"
        d.f(99); // "Calls Base.f()!"
    }
}
```
7.3.4 Dynamic class loading

Java's execution model is much more dynamic than that of C++. A C++ program is typically built by compiling several independent modules and linking together to form a single executable. In sharp contrast, a Java program comprises several class files (and optional resources) that need not be bundled into a single file. However, for ease of distribution, these may be packaged as a jar or zip file. This physical representation apart, an executing Java program may dynamically load classes on demand from arbitrary (at least in principle) sources. One consequence of this dynamic feature is that a Java compiler can easily be fooled, such that a compile-time detectable problem may manifest at run-time. Consider the following two Java classes:

```java
// File: A.java
class A {
    public void ff() { /* ... */}
}

// File: B.java
class B {
    A aobj;
    public void gg() { aobj.ff(); }
    public static void main(String[] args) {
        B b1 = new B();
        b1.gg();
    }
}
```

Assume that these two files are compiled, perhaps A first and then B. The compiler while compiling B can validate access to A's elements. Once B is compiled, suppose the programmer changes A.java as follows:

```java
// File: A.java
class A {
    private void ff() { /* ... */}
}
```

If B.java is recompiled at this point, this will result in compilation error since a private element of A is being accessed. However, if the programmer compiles A.java alone and then executes the program, there will an exception thrown at the point where the private member of A is accessed from B! Such a scenario is unlikely in the
context of C++ since all changed files will automatically be rebuilt by the compilation system, for instance, through a *make* process.

### 7.3.5 Genericity

One of the strengths of C++ is its support for generic programming through templates. This means that a Stack abstraction may be implemented without undue concern on the type of elements it will hold. The compiler is encouraged to synthesize appropriate code to make the abstraction work for concrete types. Another example is writing a sort function without tying the logic of sorting to a specific collection of data. Programming with templates is one of the biggest challenges in mastering C++. Java, as of this writing, does not support this notion. One, not always acceptable, solution is to employ polymorphic behavior by using a super class. In the author’s personal opinion, this is a limitation of Java.

### 7.3.6 Control of inlining methods

C++ supports a keyword *inline* that may be used by the programmer to recommend to the compiler inline expansion of a function (at the place of call) for efficiency reasons. It is clearly specified in the language that this inline specification may be ignored by a compiler for various reasons. Many a time, this results in code bloat. This explicit, programmer specifiable, inline expansion of method bodies is not available in Java. Instead, a Java compiler might, based on contextual knowledge (e.g., a private method with only a few statements in its body), automatically inline the method. This latter approach appears advantageous, since the possibility of a compiler overruling the programmer does not arise.

### 7.3.7 Language grammar

A formal syntax specification of Java language is much more manageable than that of C++. This observation is based on the author’s personal experience in implementing a few static code analyzers for Java.
7.3.8 Language complexity

It is the author's opinion (this view is shared by many programmers) that Java is much simpler a language to learn and use than C++. Part of this is due to the simplified syntax and fewer language features of Java. One major contributor to the overwhelming complexity of C++ is that it is a multiparadigm language, whereas Java is strictly an object-oriented language. The following are major contributors to Java's relative simplicity:

- Support for a single programming paradigm, namely object-orientation
- Absence of pointers
- Lack of multiple inheritance
- Support for garbage collection
- Lack of genericity
- In-built support for mutithreaded programming

7.3.9 Ease of static analysis

Java language, due to its inherent simplicity and uniform grammar, makes it easier for performing static analysis on a piece of code. This is borne out by the author's experience in implementing a couple of such tools. One of these is a code critiquing system that identifies potential problems with Java code without executing it (Rangarajan 1997).