Test Case Generation for White-box Unit Testing

A Thesis

Presented to the

Faculty of

California State Polytechnic University, Pomona

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

In

Computer Science

By

Samira Hussain

2007

SIGNATURE PAGE

THESIS: **TEST CASE GENERATION FOR WHITE-BOX UNIT TESTING**

AUTHOR: **Samira Hussain**

DATE SUBMITTED:

Department of Computer Science

Dr. Salam Salloum _____
Thesis Committee Chair
Computer Science

Dr. Chung Lee _____
Computer Science

Dr. Daisy Sang _____
Computer Science

ACKNOWLEDGEMENTS

I wish to thank my thesis advisor Dr. Salam Salloum for his continuous support, guidance and feedback throughout the completion of this project. My sincere thanks to the committee members, Dr. Chung Lee and Dr. Daisy Sang, for their help, support and encouragement.

A very special thank you to my parents, Dr. Hussain Najam and Attira Hussain, for their selfless emotional and financial support, without which I could not have gotten through all these years.

I wish to acknowledge Syed Mohsin for his loving and caring support. Thanks for your incredible patience – this would not have been possible without your help.

Finally, I would like to thank the faculty of Computer Science at California State Polytechnic University, Pomona for keeping me focused.

ABSTRACT

Black-box and white-box testing are the two major techniques for unit testing. In black-box testing, no information about the internal structure of the program under testing is available. However, in white-box testing, a complete source code or the internal structure is available.

Basis path testing is a white-box testing technique that uses a control flow graph (CFG) of a given program to generate a basis set of independent of paths for the CFG. Different techniques have been proposed to generate test data that cover all the paths of a basis set.

In this thesis, we implemented an interactive tool that performs three tasks:

- constructs a control flow graph of a given program based on the pseudocode and information provided by the user;
- computes a basis set of independent paths of the control flow graph;
- generates test data using genetic algorithms to exercise all basis paths.

We evaluated the performance of different mutation operators for the genetic algorithm based on the percentage of basis paths covered by the generated test data. Experiments show that the use of two known mutation operators, input value and one-point crossover, provide the best path coverage for the programs tested.

TABLE OF CONTENTS

1. Introduction

Software testing is the process of evaluating the developed software to ensure that it correctly implements a specific function and is traceable to customer requirements. Testing is a critical element of software quality assurance that ensures the correctness, completeness and quality of the developed computer software.

IEEE defines unit testing as "the testing of individual software or hardware units or groups of related units" [Juristo et al. 2006]. The focus of unit testing is on each component of the software as implemented in the source code.

Three aspects of testing process that could be partially automated are test data generation, test execution and test checking. Automated test generation requires the analysis of some formal object such as source code or a formal specification [Doong and Frankl 1994]. Most research on automated test generation has involved program based or white-box testing techniques (i.e. techniques based on analysis of the source code of the program under test).

Basis path (or path) testing is a white-box testing technique that is widely used during unit testing. Basis path testing uses a control flow graph to depict the logical control flow of program under test. Important control paths are identified and tested to uncover errors within the boundary of the module.

The focus of my research would be to design and develop an interactive tool for basis path testing using genetic algorithms. The tool performs three tasks: it constructs a control flow graph of a given program based on the pseudocode provided by the user, it computes the basis set of independent paths in the control flow graph and it generates test data using genetic algorithms to exercise all the basis paths.

Test case generation support in the tool is based on the application of evolution strategies and genetic algorithms. Genetic algorithms search for optimal test parameter combinations that satisfy a predefined test criterion. This test criterion is represented through a "cost function" that measures how well each of the automatically generated optimization parameters satisfy the given test criterion. Various test criteria are possible according to the goal of the test, such as how well a test covers a piece of code in the case of conventional software testing. This project uses a genetic algorithm to generate input parameter combinations for test cases that achieve high path coverage.

Basis path testing can also be applied to test object-oriented software at the class level [Pressman 2001]. Currently, there is a lack of solid (practical) methodologies for unit testing of object-oriented programs in the literature. Most of the methodologies have their roots in white-box testing used for unit testing of conventional software. The development of a tool that would automatically generate test cases for basis path testing can be modified to generate test cases for object-oriented programs.

The motivation for selecting this research topic lies in the importance of software testing and its implications with respect to software quality. The requirement for higher-quality software demands a more systematic approach to testing. According to Pressman [2001, p. 631], "the testing of object-oriented systems presents a new set of challenges to the software engineer." Hence, my research would look into the feasibility of modifying the developed tool to generate test cases for object-oriented programs.

The thesis report is organized as follows: Section 2 surveys the literature on testing, focusing on conventional and object-oriented testing paradigms. Section 3 describes the research goal of the thesis project. Section 4 gives the details of the methodology used to

achieve the research goal and produce results. An evaluation of the results of the experiments is presented in Section 5. Section 6 discusses the limitations and draws conclusions, followed by future work and reference section.

2. Literature Survey

Testing is the most widely used and accepted technique for verification and validation of software systems. It is applied to measure the extent to which a software system conforms to its original requirements and to demonstrate its correct operation. Testing identifies problems and failures in all the phases of software development. Ideally, software testing guarantees the absence of faults in software, but in reality it only reveals the presence of software faults but never guarantees their absence [Abreu et al. 2004].

Testing involves devising a set of inputs that will cause the software to exercise some portion of its code. These inputs are referred to as test cases. Testing uses a finite collection of test cases. A test case is successful if it detects a new error and unsuccessful otherwise. Test cases should include both valid and invalid data.

Majority of the software practitioners agree with the motto “test early, test often and test enough” [McGregor and Sykes 2001]. This motto helps to uncover problems early in the development process, which in turn reduces the size of effort required to perform adequate system testing by determining what needs to be tested. The recent iterative development processes focus on analyzing a little, designing a little, coding a little and testing what you can [McGregor and Sykes 2001]. It cannot be sufficiently overemphasized that regular testing can detect failures early in the software development and save reworking in subsequent iterations.

There are two types of software's to test: conventional software and object-oriented software.

2.1 Conventional Testing

Conventional testing is used to test systems following a procedural programming approach. These systems rely mostly on walkthroughs and inspections (static verification) to remove faults. Testing of conventional software can be carried out at three different levels:

- **Unit testing:** concentrates on testing each unit (or component) of the software as implemented in the source code.
- **Integration testing:** concentrates on systematic testing of multiple units of software.
- **System testing:** concentrates on testing the software as a whole.

2.2 Object-Oriented Testing

Object-Oriented (OO) testing is used to test systems following an object-oriented programming approach. Object-Oriented (OO) paradigm increases software reusability, extensibility, interoperability and reliability.

Object-oriented testing, in spite of being strategically similar to conventional testing, is tactically very different. With OO systems the three traditional testing phases of unit testing, integration testing and system testing are replaced with the following four levels [Chen et al. 2001, Chen 2003a, Chen 2003b]:

- **Algorithmic/Method level testing:** tests the implementation of each member function in a given class.

- **Class level testing:** tests the interaction between different methods and data in a given class. This is also known as intra-class testing as classes are tested in isolation.
- **Cluster level testing:** tests the interaction between different classes in a given cluster. This is also known as inter-class testing as sets of classes are tested together.
- **System level testing:** tests the interaction between different clusters in a given system. This is similar to conventional software's system level testing where the software is tested as a whole.

OO testing at the algorithmic and system levels is similar to conventional testing. However, OO testing at the class and cluster levels poses new challenges. Unit testing is replaced with class level testing in object-oriented systems. Class level testing involves testing a single object in isolation. This is much more complex than traditional unit testing because a class may have several methods, attributes, possibly some kind of inheritance, and also relationships with other classes.

The advantages of object orientation become potential disadvantages while testing, as object-oriented code is considerably harder to read than conventional source code. Object-oriented testing is much more complex than conventional testing due to the following reasons:

- **Encapsulation:** or data hiding complicates testing because operations must be added to a class interface (by the developer) to support testing. A member function of a class may invoke several other member functions from different object classes to achieve an intended functionality. The implication of long

invocation chains is that a tester has to understand the sequence of member functions and semantics of the class prior to preparing test cases.

- **Abstraction:** of data and code in OO programming complicates the testing process. Layers of testing parallels layers of abstraction in the development process. Testing from the highest level of abstraction provides more effective and accurate set of tests.
- **Cyclic dependency:** makes it difficult to understand a given class in a large OO program if that class depends on many other classes. It is difficult to test inheritance, aggregation, association, template classes, dynamic object creation, polymorphism and dynamic binding relationships. This makes it hard for a tester to know where to start testing in an OO library. It is extremely costly to construct stubs. The impact of a small change may ripple throughout the OO program due to the presence of complex dependencies.
- **State behavior:** documentation for OO testing is either missing or poor. Objects have states and state dependent behaviors. Control flow and state control is distributed over several classes, making it difficult to generate test cases.
- **Inheritance:** provides a mechanism by which bugs propagate from a parent class to each of its descendants. Some of the test cases for superclass can be reused for subclasses. However, the use of inheritance solely for code and test case reuse will lead to difficulties.

- **Polymorphism:** puts more importance on testing a representative sample of runtime/dynamic configurations. Polymorphism leads to more run-time errors; hence requiring explicit unit testing.
- **Tool support:** for OO testing is still in infancy. The tools that support OO testing (Computer Assisted Software Engineering (CASE) tools) are in the development phase. Formal specifications are rarely used for testing. Hence, testing remains a tedious process where the tester has to manually prepare the test cases.

2.3 Unit Testing Techniques for Conventional Software

Software testing techniques are broadly divided into two main categories: black-box testing and white-box testing.

2.3.1 Black-box Testing

Black-box testing is also known as function (or specification) based testing. It generates test data for software based on its specification. Black-box approach tests the software by using the input data and output results; hence, it doesn't need to look into the source code. Some of the black-box testing methods include:

- **Equivalence Partitioning:** It divides the input domain of a program into classes of data from which test cases can be derived. It defines a test case that uncovers a class of errors, thereby reducing the total number of test cases that must be developed. An equivalence class represents a set of valid or invalid states for input conditions. Typically, an input condition is a specific numeric value, a range of values, a set of related values or a Boolean condition.

- **Boundary Value Analysis:** It leads to a selection of test cases that exercise bounding values or "edges". A greater number of errors tend to occur at the boundaries of the input domain rather than in the "center". This test case design technique complements equivalence partitioning.

2.3.2 White-box Testing

White-box testing is also known as structure testing or glass-box testing. It focuses on the procedural details (or internal structure) of the software when generating test data and test cases. The derived test cases guarantee that all independent paths within a module have been exercised at least once and that all logical decisions have been exercised on their true and false sides.

White-box testing is performed early in the testing process whereas black-box testing is applied during the later stages of testing. These two techniques uncover a different class of errors and hence are complementary to each other. White-box testing is mostly used at unit level testing whereas black-box testing is used at integration or system level testing. Some of the white-box testing methods include:

2.3.2.1 Control Flow Testing

Control flow testing is a white-box testing technique that tests the flow of control in a program. The variations of control structure testing that can broaden test coverage and improve quality of white-box testing are:

- **Basis path testing:** is a white-box testing technique for control flow testing. It was first proposed by Thomas McCabe [Pressman 2003] in the mid 1970s. This technique uses the control flow graph of a program module to generate a set of independent paths that must be executed to assure that all statements and branches

in the program have been executed at least once. Basis path testing enables the designer to derive a logical complexity measure, known as cyclomatic complexity, of the procedural design that is used as a guideline for defining a basis set of execution paths. Test cases derived to execute the basis set are guaranteed to execute every statement in the program at least one time during testing. Although basis path testing is simple and highly effective, but it is not sufficient in itself.

- **Condition testing:** is a test case design method that exercises the logical conditions contained in the program module. It focuses on testing each condition in the program. A condition can be made up of relational (<, >, =, !=, <= or >=) or Boolean (AND, OR, NOT) operators. Furthermore, a condition can be simple (only one operator) or compound (two or more operators). A number of condition testing strategies exist:
    - **Branch testing:** executes the true and false branches of a compound condition C and every simple condition in C at least once.
    - **Domain testing:** requires three or four tests to be derived for a relational expression of the form E <relational-operator> F. The three tests are required to make the value of E greater than, equal to or less than that of F. For a Boolean expression with n variables, all of 2^n possible tests are required (n > 0). This strategy is practical only if n is small.

Condition testing strategies have two advantages: measurement of test coverage of a condition is simple and it provides guidance for generating additional tests for the

program. Therefore, the purpose of condition testing is to detect errors both in the conditions of a program and in the program itself.

2.3.2.2 Data Flow Testing

Data flow testing selects test paths of a program according to the location of definitions and subsequent uses of variables in the program. “Data flow testing strategies are useful for selecting test paths of a program containing nested if and loop statements” [Pressman 2003, p. 457]. A number of data flow testing strategies exist. One strategy is definition-use (DU) testing which requires that every DU chain of a variable x be covered at least once.

2.3.2.3 Loop Testing

Loop testing focuses exclusively on the validity of loop constructs. Loops are a major part of all algorithms implemented in the software. Four different classes of loops can be defined:

- **Simple loops:** consist of a single loop with n iterations where n is the maximum number of allowable passes through the loop.
- **Nested loops:** consist of two or more simple loops with each loop having a number of iterations.
- **Concatenated loops:** consist of two or more simple loops that can be dependent or independent of each other.
- **Unstructured loops:** are highly complex and should be redesigned to fit the above classes.

Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons or improper control flow. In summary, basis path and loop testing

are effective techniques for uncovering a broad array of path errors. Test cases that exercise control flow and data values just below, at or above maxima and minima are very likely to uncover errors.

2.4 Unit/Class Level Testing Techniques for Object-Oriented Software

In traditional/conventional systems, unit testing usually means testing just a single function or procedure. But in OO systems, unit/class testing is more complex because a class may have several methods, attributes, possibly some kind of inheritance, and also relationships with other classes. We still perform unit testing but we change the definition of unit.

There are very few practical methodologies in the literature for unit testing of object-oriented programs. Most of the methodologies have their roots in white-box approach used for unit testing of conventional software. Some of the class-level testing strategies that have been proposed in the literature are discussed below. Out of the nine approaches discussed below, only Chen et al.'s [2001] approach has been implemented.

2.4.1 Pressman

According to Pressman [2003], class testing for OO software is equivalent to unit testing for conventional software. The concept of unit changes when OO software is considered. The smallest testable unit is an encapsulated class or object. A class can contain a number of different operations. Therefore, testing operation X in a vacuum is ineffective in the OO context.

"Test case design methods for OO software are still evolving" [Pressman 2003, p.637]. OO testing focuses on designing appropriate sequence of operations to exercise the states of a class. White-box testing methods can be applied to the operations defined for a class.

Basis path, loop testing or data flow techniques can help to ensure that every statement in an operation has been tested.

Random testing and partition testing are two methods that can be used to exercise a class during OO testing.

- **Random testing at the class level:** This tests the behavioral life history of an instance of a class by randomly testing the sequence of operations that can be performed on the instance. Random order tests are conducted to ensure coverage testing of all attributes and operations in a class. Test sequences are designed to ensure that all relevant operations are exercised.
- **Partition testing at the class level:** This reduces the number of test cases required to exercise the class in much the same manner as equivalence partitioning for conventional software. Input and output are categorized and test cases are designed to exercise each category. Partitioning categories can be based on states, attributes or function category of the class:
    a) **State based partitioning:** categorizes operations that change state and those that do not change state separately.
    b) **Attribute based partitioning:** categorizes class operations based on the attributes they use.
    c) **Function based partitioning:** categorizes class functions based on the generic operations that each performs. For example, functions in a class can be categorized into initialization functions, computation functions, query functions and termination functions.

2.4.2 SPECIAL

It is recommended that object-oriented testing be performed on the basis of eight SPECIAL features [Object-Oriented Testing Special Requirements] where this acronym stands for:

S represents State of the object: State testing is required to ensure that an object is in its correct and stable state.

P represents Polymorphism of the object: Polymorphism testing is required to ensure that all possible bindings have been tested.

E represents Encapsulation of the data: Private variables and methods are tested using this feature.

C represents Contract or Interface testing

I represents Inheritance hierarchy of the object: Inherited features require retesting in most circumstances. Retesting is required in most cases because inheritance increases the context of usage.

A represents Association/Abstraction of the class and

L represents Loss of contract during communication between the objects.

2.4.3 Ambler

Full-Lifecycle Object-Oriented Testing (FLOOT) methodology [Ambler 2004] is a collection of testing techniques to verify and validate object-oriented software. It allows thorough testing throughout all stages of software development. Figure 1 illustrates the various tests involved in the FLOOT methodology. These tests include:

a) **Black-box testing:** selects test data based on the software specifications. It ignores program structure. Some types of black-box testing include boundary value testing and equivalence class testing.

b) **Class testing:** ensures that a class and its instances (objects) perform as defined.

c) **Class-integration testing:** ensures that classes and their instances perform as defined.

d) **Code review:** is a form of technical review in which the source code is reviewed thoroughly.

e) **Component testing:** involves validating that a component/module works as defined.

f) **Design review:** is a technical review in which the design model is inspected thoroughly.

g) **Inheritance-regression testing:** involves running the test cases of super classes on a given subclass.

h) **Integration testing:** involves verifying that several portions of software work together.

i) **Method testing:** involves verifying that a member function performs as defined.

j) **Model review:** is an inspection by an independent group of people on the software model under development.

k) **Prototype review:** is used by the end users of the system to test the design of the prototype.

l) **Prove it with code:** involves building the software based on the model. This shows whether the model works or not.

m) **Regression testing:** ensures that previously tested behaviors still work as expected after changes have been made to an application.

n) **Stress testing:** ensures that system performs as expected under high volume of transactions, users etc.

o) **Technical review:** is a quality assurance technique in which the design of application is examined critically by a group of peers. It focuses on accuracy, quality, usability and completeness. Review is also referred to as a walkthrough, inspection or peer review.

p) **Usage scenario testing:** involves validating a model by acting through the logic of usage scenarios.

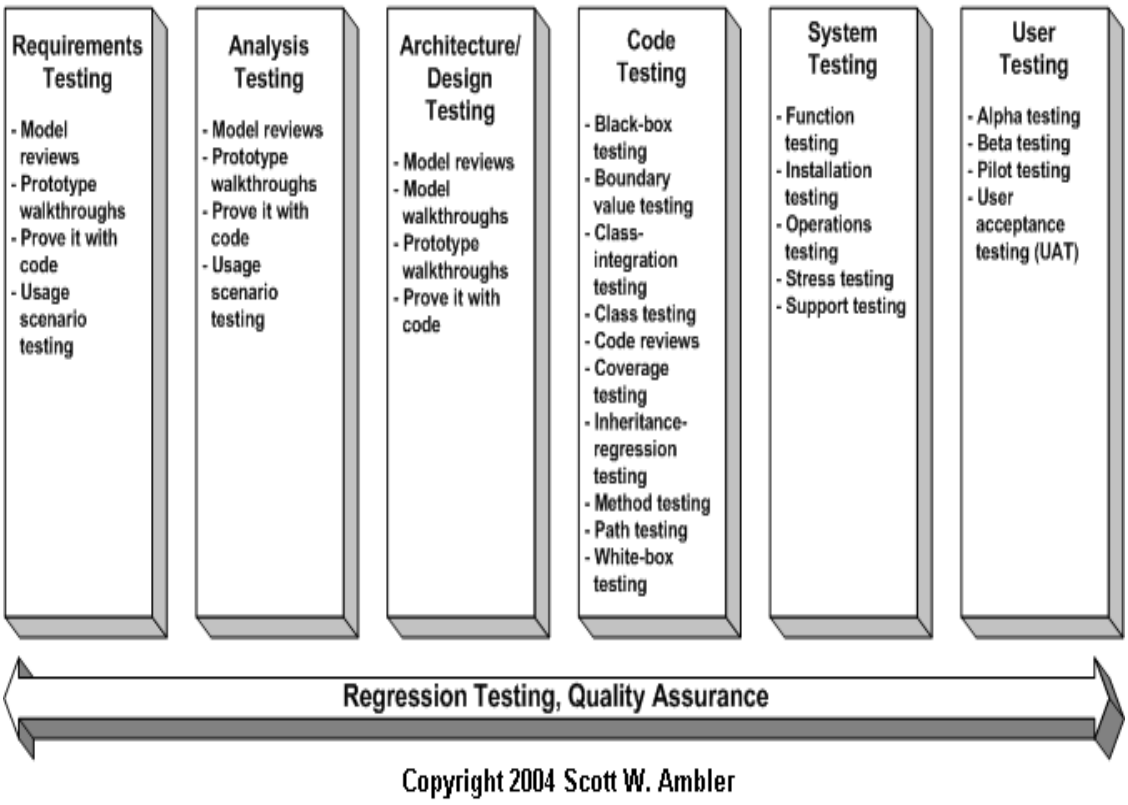

Figure 1: FLOOT methodology

q) **User interface testing:** is also referred to as graphical user interface (GUI) testing. It ensures that user interface follows the accepted standards and requirements defined for it.

r) **White-box testing:** selects test data based on the program's control flow or data flow. It involves designing test cases to test the specific parts of the code/program.

2.4.4 Harrold

The paper by Chen et al. [1998] describes some useful unit test strategies for object-oriented software's. These require finding an order to test the classes so that effort required to construct test stubs is minimum. The unit test strategy proposed by Harrold has six steps:

a) Take base classes with no children. Design test cases to test each member function individually and interactions among member functions.

b) Testing history associates each test case with the attributes it tests. Subclass "inherits" its parents testing history.

c) Inherited testing history is incrementally updated to reflect differences from the parent. Result is a testing history for the subclass.

d) With this technique, new attributes can easily be identified in the subclass that must be tested along with inherited attributes that must be retested.

e) Inherited attributes are retested in subclass by testing their interactions with newly defined attributes in the subclass.

f) Test cases that can be reused in parent class's test suite are subsequently identified.

2.4.5 Kung

The test order finding algorithm defined by Kung is based on a class model called Object Relation Diagram (ORD) [Chen et al. 1998, Durand et al. 2000]. A reverse engineering process generates an ORD by analyzing C++ source code of an object-oriented program.

An ORD is a directed graph (digraph) in which vertices represent the object classes and edges represent the relationships among object classes. ORD shows inheritance, aggregation, association, instantiation, nested and uses relationships. Hence, classes and relationships are displayed diagrammatically. There are two types of ORDs:

- **Acyclic digraph:** means that there exists no cycles in the digraph. Test order is the topological sorting of a set of classes using a given dependence relationship. Computational complexity is the number of classes in the OO program. The effort required to construct test stubs is 0 in this case.
- **Cyclic digraph:** means that there exists one or more cycles. Topological sorting cannot be applied to cyclic digraphs. The solution is to convert the cyclic digraph into an acyclic digraph by treating each strongly connected component as a composite vertex. Topological sorting can then be applied to the resulting acyclic digraph to produce an optimal test order for unit testing of OO programs.

2.4.6 Parrish, Borie and Cordes's

Parrish et al. [Chen et al. 1998] propose a white-box flow-graph based testing approach to test classes. Their approach allows automatic generation of test cases for classes specified in a formal language. The class behavior is modeled by a flow graph (class graph) that is analogous to the concept of control flow graph and data flow graph in

conventional testing. A class is described in a model-based specification language (for e.g. Z or VDM). The flow graph of a class is a directed graph, which is a collection of <N, E, D, U, I>, where N is the set of nodes, E is the set of edges, D denotes the set of definitions of data, U denotes the set of uses of data and I refers to the set of infeasible sub paths. There is one node for each operation in the class. N, E, D, and U are obtained from the class interface in the implementation. Once the flow graph is created for a class, test cases can be systematically selected by traversing the graph. This approach can be automated efficiently. However, some restrictions substantially weaken the degree of testing demanded and reduce the significance of the results.

2.4.7 Chen

In this approach, class behavior is modeled as a directed graph called testgraph [Chen et al. 1998]. The nodes correspond to a state of the class under test. Start node corresponds to the initial state. Each edge corresponds to a sequence of operations that cause the transition from one state to the next state. Only a limited number of representative states constitute the testgraph. Test cases are generated by traversing the testgraph according to the given coverage criteria (such as node coverage, path coverage, branch coverage etc.)

2.4.8 Doong and Frankl

Doong and Frankl [1994] propose A Set of Tools for Object-Oriented Testing (ASTOOT). ASTOOT is an interactive tool for semi-automatically generating test cases for a class from an algebraic specification language called LOBAS. The properties of algebraic specification and the definition of correctness of class implementation form a basis for test case design in the ASTOOT approach. Each test case consists of a tuple (S1,

S2, tag), where S1 and S2 are sequences of operations and tag is either "equivalent" or "unequivalent" indicating whether or not S1 and S2 are observationally equivalent according to the specification. Two objects are observationally equivalent if and only if they produce identical results when subjected to the same sequence of operations and unequivalent otherwise. This approach focuses on automating the unit testing of abstract data types (ADTs) for test data generation, test execution, and test checking in object-oriented programs.

2.4.9 Chen et al.

Chen et al. [2001] propose TACCLE (Testing At the Class and Cluster LEvels) methodology to test object-oriented software at the class level. Their work is motivated by the ASTOOT black-box approach by Doong and Frankl [1994].

White-box testing suffers from certain limitations, such as its inability to generate test cases when certain parts of specification have been inadvertently omitted from the program. This observation has motivated the integration of black and white-box techniques in TACCLE; black-box technique is used to select test cases whereas white-box technique is applied to determine whether two objects resulting from the execution of a test case are observationally equivalent. It is also used to generate test cases afterwards.

TACCLE is more precise than ASTOOT as it is heavily based on mathematical theorems and algebraic specifications. The lack of predetermined test oracles (expected test results) in real life applications has led TACCLE to take sequence of operations as test cases. Message passing among objects is used to determine whether the objects would preserve the behavioral properties defined by the specification. The complete TACCLE methodology consists of 3 components:

- Using equivalent ground terms as class level test cases to determine the observational/behavioral equivalence of objects.
- Using nonequivalent ground terms as class-level test cases.
- Using sequences of message-passing expressions and post-conditions as cluster-level test cases.

These three components are closely related and supplement one another. This section will discuss the first two components of the methodology as they are used for class level testing.

The algorithm to perform class level testing picks up two instances of a class. It then tests whether they are observationally equivalent or non-equivalent. Two objects are observationally equivalent if and only if they produce identical results when subjected to the same sequence of operations. Observationally non-equivalent is the opposite of observationally equivalent behavior. Some concepts related to the TACCLE methodology that are worth discussing here include [Chen et al. 2001]:

a) **Algebraic specification:** is a formal way of specifying that a class C implementing an abstract data type is correct. It has a syntactic part and a semantic part. The syntactic part consists of function names and their signatures (domains and co-domains) according to the input parameters and output of the operations. The semantic part of the specification consists of a list of axioms describing the relation among functions. A specification must be consistent (no contradictory axioms) and sufficiently complete. Some examples of algebraic specification languages are LOBAS and Contract.

b) **Term:** consists of a sequence of operations in an algebraic specification, for e.g. the functions new.push(10).push(20).pop.top in the class of IntegerStack (stack of nonnegative integers).

c) **Ground term:** is a term without variables. They have finite length and are composed of the different operations (creators, constructors and transformers) in the specification, for e.g. new.push(1).push(2).pop.top in the IntegerStack class.

d) **Normal form:** A ground term is in normal form if and only if it cannot be further transformed by an axiom in the specification. Every ground term can be transformed into a unique normal form in a finite number of steps by using the creators/constructors of the class.

e) **Canonical form:** An algebraic specification is in canonical form if and only if the transformation of a ground term reaches a unique normal form in a finite number of steps.

f) **(Normal/Attributive) Equivalence:** Two ground terms are equivalent if and only if they can be transformed into the same normal form by some axioms in the specification (as left-to-right rewriting rules). "Two terms u1 and u2 in a given specification are said to be equivalent if we can use the axioms in the specification to transform u1 into u2" [Chen et al. 2001, p. 63]. A tuple (S1, S2, tag) where S1 and S2 are sequence of messages and tag is "equivalent" if S1 is equivalent to S2 and is "nonequivalent" otherwise. If all observational equivalence checks agree with the tags, then the implementation is correct. Otherwise, it is incorrect.

g) **Observational Equivalence:** Given a canonical specification of a class C, two ground terms u1 and u2 are observationally equivalent if and only if they produce identical results when subjected to the same sequence of transitions/operations.

h) **Fundamental pair:** of equivalent ground terms is formed by replacing all the variables on both sides of an axiom by normal forms. The set of fundamental pairs is a proper subset of the set of equivalent ground terms.

i) **Observational Nonequivalence:** If two objects produce different results when subjected to different sequence of operations, then they are observationally nonequivalent.

j) **Observers:** of class C are the methods that return the values of the attributes of the objects in C, for e.g. empty() and top() in the IntegerStack class.

k) **Creators:** of class C are methods that return initial objects of C, for e.g. pop() in the IntegerStack class.

l) **Transformers:** of class C are methods that transform the states of objects in C. They change the value of at least one attribute of the object, for e.g. push() in the IntegerStack class.

m) **Current State:** of an object is the combination of current values of all attributes of the object.

n) **Observable context:** of a class C is a sequence of operations (possibly an empty sequence) ending with the observer function, for e.g. push().push().pop.top in the IntegerStack class.

In general, the set of all equivalent ground terms in a given specification is infinite. It is impossible to carry out exhaustive testing. Therefore, TACCLE proposes a

mathematically based argument [Chen et al. 2001] for selecting a finite set of fundamental pairs as test cases as it reduces the test case selection domain. This set covers the use of equivalent ground terms as test cases. Hence, TACCLE only concentrates on the testing of fundamental pairs.

2.4.9.1 Selecting Equivalent Ground Terms as Test Cases

The idea of using equivalent ground terms rather than individual operations as test cases in OO testing is because a series of messages are often passed to an object in OO programming. The resulting object is then evaluated for correctness.

Based on the mathematical strategy, Chan et al. [1998] propose an algorithm for Generating a Finite set of fundamental pairs as Test cases (GFT). GFT invokes an algorithm for Determining Observational Equivalence of objects (DOE) resulting from the execution of a test case.

Algorithm: Generating a Finite Set of Fundamental Pairs as Test Cases (GFT)

This algorithm is used to select a finite number of representative test cases from the set of fundamental pairs. Given a canonical specification and a complete class implementation, the algorithm constructs fundamental pairs from each axiom in the canonical specification. Suppose there are n axioms (a_1, a_2, … , a_n) in the specification. For each axiom a_i (i = 1 to n), conduct the following steps:

a) Use the constructors and creators in the given specification to unfold each axiom a_i into several new equations such that the length of the new equation does not exceed some positive integer k. The value of k can be determined by the user or by some white-box technique (for e.g. maximum size of array, boundary values of variables etc.).

b) If the right hand side of the new equation a_{ij} obtained from step a) contains a defined operation f, then partition the input domain of f into subdomains.

c) Randomly select some test points from each subdomain obtained from step b) such that all the test points in each subdomain cause a particular path in the implementation of method to be executed. Use these elements to replace all occurrences of the corresponding input variables in equation a_{ij} to obtain a group of fundamental pairs induced from axiom a_i.

d) If the above group of fundamental pairs reveals an error, exit from the algorithm.

The limitations of GFT algorithm are that it cannot be fully automated because it is difficult to determine the positive integer k in step (a) of algorithm. Also, the algorithm is path-oriented so it inherits problems associated with path testing such as infinite number of paths and identification of infeasible paths.

The complexity of GFT algorithm depends heavily on the actual number of normal forms generated for a given positive integer k. When the boundary values are large, the sizes of test cases may be of the order O (m^k) where m is the number of constructors in the class under test.

Algorithm: Determining Observational Equivalence of Objects (DOE)

It has been formally proven by Chan et al. [1998] that deciding whether two objects are observationally equivalent is undecidable using pure black-box techniques. A heuristic white-box technique, DOE, is used to select a relevant finite subset of the set of observable contexts. This is done with the help of a Data member Relevance Graph (DRG) constructed from the implementation of class C. The notation used in DRG is as follows:

a) Bold rectangle node: represents each data member of class C.

b) Thin rectangle node: denotes some constants coming from the given program.

c) Arc: If the data member d_2 directly affects data member d_1 in the method m_1 under a condition p(…), then there is an arc labeled (p,m_1) from d_2 to d_1.

The authors' call $[d_2, (p, m_1), d_1]$ a segment (Figure 2) of the DRG with d_2 as the start node of arc and d_1 as the end node of arc. (p, m_1) is the output arc of d_2 and (p,m_1) is the input arc of d_1. If d_2 is identical to d_1, the segment is said to be a cycle. Otherwise it is said to be acyclic.

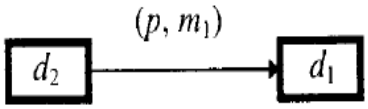

Figure 2: Segment of DRG

Each DRG contains a special node called the observed node, which is the ending node of each arc with the observer node as the second component of its label. This is to determine the observable context of an object after a sequence of operations have been carried out on it. An arc with observed node as an ending node is called an observer arc. Suppose O_1 and O_2 are two objects of the same implemented class C, resulting from the execution of method sequence s_1 and s_2 respectively. Steps for deciding whether O_1 is observationally equivalent to O_2 ($O_1 = O_2$) are as follows:

a) Construct the DRG of class C from the implementation of class C.

b) Suppose the data members of the implemented class C are $d_1, d_2, \ldots, d_n$. Suppose further that $O_t.d_i$ denotes the values of d_{1i} of O_t for i = 1 to n and t = 1, 2. Check

whether tuples ($O_1.d_1$, $O_1.d_2$, …, $O_1.d_n$) and ($O_2.d_1$, $O_2.d_2$, …, $O_2.d_n$) are equal. If yes, we have $O_1 = O_2$. Exit from the algorithm DOE. Otherwise go to step c).

c) Suppose O_1 and O_2 have different values with respect to the data members d_1, d_2,…,d_k. In other words, suppose $O_1.\ d_j\ != O_2.\ d_j$ for j = 1 to k. For each data member d_j for both objects, check whether there is a path from the data member node d_j to the observed node in the DRG. If yes, carry out the following steps:

   1) If data member d_j is a simple data type, traverse every acyclic executable path P once and obtain the OC (observable context) "oc" induced from P. User enters the values for any un-instantiated input variables in oc. If a cycle is encountered when traversing an executable path, the user should manually decide on a ceiling for the number of iterations of the cycle or supply a global ceiling T allowed by the system. Check whether at least one of these OC fails (i.e. produces different results). If so, O_1 is not equal to O_2, exit from the algorithm DOE. Otherwise, data member d_j has successfully passed the check, go to step 3).

   2) If data member d_j is a compound data type (e.g. array or structure), construct OC by the following process. For each element of the array, traverse each executable path (not exceeding the iteration ceilings in the case of cycles) from data member node to the observed node to obtain an OC. User enters the values for any un-instantiated input variables in oc. Check whether at least one of these OC fails (i.e. produces different results). If so, O_1 is not equal to O_2, exit from the algorithm DOE.

Otherwise, if data member has successfully passed the check, go to step 3).

3) If all the data members have successfully passed the checks, then we have $O_1 = O_2$. Exit from the algorithm DOE. Otherwise, continue to check the next data member d_j such that $O_1.\ d_j\ != O_2.\ d_j$.

By using DOE, we can skip the testing of many irrelevant cases i.e. skipping irrelevant observable contexts. DOE can help detect missing paths – parts of specification omitted from the implementation. The limitations of DOE algorithm are that if DRG contains cycles, then the set of OC is infinite. We can only choose a finite subset as test cases. Hence, DOE can't guarantee that all implementation errors will be revealed by a finite set of test cases. Also, if an observable context itself contains an error, then in some cases, no error is reported. This is an inherent limitation of program testing.

The algorithm requires the users to supply the following information manually: number of iterations/global ceiling to traverse cycles, values for un-instantiated variables and two equivalent method sequences corresponding to a selected fundamental pair of equivalent ground terms for the given class under test. Authors are considering the feasibility of adding further heuristics to the algorithm to reduce manual input from the user.

The size of DRG can be represented by a tuple (N, S) where N is the number of nodes in DRG and S is the number of segments. If there are D data members, M methods and P conditions in each method of the implementation, then $N = D + 1$ and $S \leq D^2 . M . P$. This worst case rarely occurs. DRG is simple in most practical situations since DRG models the class, which is relatively low level in OO paradigm.

The complexity of traversing executable paths in algorithm DOE is O (n^L) in the worst case, where n is the maximum number of boolean conditions in the output arcs of any node that are true for the current data member values and L is the maximum length of all acyclic paths from any node to the observed node. Experiments show that the constant L is small in most practical situations.

2.4.9.2 Selecting Nonequivalent Ground Terms as Test Cases

Nonequivalent ground terms are generated as test cases using state-transition diagrams [Chen et al. 2001]. This is the second phase of TACCLE methodology. It is significant because nonequivalent ground terms might erroneously have been implemented as equivalent. Test cases are selected from the set of all pairs of attributively nonequivalent terms, which is infinite in general. Based on the mathematical strategy, Chen et al. [2001] propose an algorithm for Generating Attributively Nonequivalent ground terms as test cases (GAN). GAN constructs a State Transition Diagram (STD) from which a finite number of test cases are selected.

Algorithm: Generating Attributively Nonequivalent Ground Terms as Test Cases

Given a canonical specification of a class C and a set T of all ground terms, suppose T is partitioned into k equivalence classes (T_1, T_2, … , T_k) with respect to the attributive equivalence of terms. The term equivalence class is a discrete math concept where a set is partitioned into disjoint subsets with respect to some equivalence relation. The algorithm constructs STD from the canonical specification of the class. The notation for STD is as follows:

- **State***:* represents an equivalence class of ground terms. It is denoted by a node in STD. Set of all states in C is called the state space of C.

- **Initial state:** is established by the constructor. The node corresponding to the initial state is called the initial node (n_0).
- **Arc/transition***:* represents an operation transforming one state into another. A path is a sequence of contiguous arcs corresponding to a sequence of operations (or a term).

If the state space of C is finite, STD is constructed for class C. However, if the state space of C is infinite, then it can be partitioned into a finite number of subspaces. Each node in the STD denotes a subspace rather than a concrete state. There can be some non-deterministic transitions in the STD (i.e. involving more than one transition arc from the current node.). Dijkstra's guard condition (random execution of non-deterministic paths) or subset construction algorithm (converts non-deterministic finite state automata to deterministic finite state automata by constructing distinct new sets from given states) is used to refine a non-deterministic transition into deterministic one. C*urrent sequence* (CS) is the operation sequence on a current path from the initial node n_0 to the current node n_i. A "non-deterministic" transition from a current node will involve more than one transition arc from the current node.

The following steps generate attributively nonequivalent terms as test cases:

a) Based on the canonical specification, construct STD for the class including guard conditions, if any.

b) Let $\{n_0, n_1, \ldots, n_k\}$ denote the set of nodes in the STD where n_0 is the initial node. For each node n_i other than the initial node n_0, find a path from n_0 to n_i. Every guard condition along the path, if any, must be satisfied according to the

specification. Two terms on two paths from initial node n_0 to different nodes n_i and n_j must be nonequivalent.

c) Take the k (k-1)/2 pairs of attributively nonequivalent terms generated by the STD as test cases where k is the total number of nodes in the STD. Map each operation in the term to a method in the program. If implementation is complete, the mapping is well defined and method sequences should produce different results (nonequivalent objects). Otherwise, there is error in the implementation.

e) If one of the pairs generated in c) reveals an error, then exit from the procedure. Otherwise, randomly generate more paths from the initial node n_0 to every node n_i. If a cycle is encountered in the path, ask the user to determine a ceiling for the number of iterations of the cycle or specify a global ceiling T for the system. The ceiling corresponds to some boundary values in the code.

The following strategy is used to reduce the number of generated paths: paths with lengths corresponding to the boundary values are generated first since boundary values are more sensitive to errors. Once an error is revealed by a pair of paths, the execution of GAN will terminate, so that no other paths will need to be generated. If an error is not detected, then randomly generate some paths with lengths between the boundary values to test the non-boundary cases. If no error is detected from the random paths, report that no error has been revealed and exit from the procedure.

The limitations of algorithm GAN are that STDs are not always deterministic. It is difficult to convert non-deterministic STD to deterministic STD if the given problem is complex. Furthermore, users have to provide a ceiling for maximum number of test paths and iterations (in case of a cycle).

2.5 Test Data/Test Case Generation

Traditionally, a human tester develops test scenarios and writes the test code for software under test manually. Ideally, a testing tool should generate the entire test code automatically, but this is very difficult to achieve, so that only parts of testing process can be automated.

The process of test automation can be subdivided into three main activities:

- **Test data generation:** is to generate test scenarios according to test criteria.
- **Test oracle:** is to generate expected results from software's specifications.
- **Test case execution:** is to combine the above, test scenarios and oracle, to compare executed results with expected ones.

The first activity can be automated with relative ease. However, existing tools apply crude heuristics to find test scenarios. The automation of second activity is much more daunting in practice due to poor quality or low formality of software requirements specification. The last step simply involves the creation of an arbiter that compares the observation from software's execution with the expected observation from the oracle and decides whether the test passes or fails. This poses no difficulty on automation, once the oracle problem has been solved. The work outlined in this report and the related project concentrates on the first testing activity: automated generation of test scenarios.

Automating test data generation is an important step in reducing the cost of software development and maintenance [Abreu et al. 2004]. When automating, it is important to look at two aspects: the test data generation tool and test adequacy criterion.

A test data generation tool is responsible for creating test data, while a test adequacy criterion assures the quality of generated test cases and gives information about the end of

testing process. Many test data generators have been developed, each one using different kinds or variations of existing testing techniques. Test adequacy criterion usually involves path coverage analysis, which can be measured based on different aspects of software like statements, branches, all-uses and paths.

Path testing searches the program domain for suitable test cases that cover every possible path in the software under test. However, it is generally impossible to achieve this goal for several reasons:

- A program with loop constructs may contain an infinite number of paths. The number of paths in a program is exponential to the number of branches in the program.
- Many independent paths in a program may be infeasible (i.e. paths in a program that cannot be executed. This occurs when we cannot assign values to the program's input variables).
- The number of test cases is too large, since each path can be covered by several test cases.

For these reasons, the problem of path testing can become NP-complete problem, making the coverage of all possible paths computationally impractical.

Since it is impossible to cover every path in a software, the problem of path testing selects a subset of paths to execute and finds test data to cover them. Various test data generation methods have been proposed in the literature. They can be classified as:

- **Random test data generators:** select test data randomly from the domain of input variables. Random testing can be used to create a volume of test scenarios, but it does not specifically obey any test coverage criteria. Test tools based on

random testing generate test scenarios that simply measure and illustrate the coverage of software. They cannot generate test scenarios that are "guided" by the coverage.

- **Symbolic evaluators:** analyze a program to obtain a set of symbolic representations of each condition predicate along the selected path. The expressions are obtained by attributing symbolic values to the input variables. Symbolic execution methods are static in nature. If the predicates are linear, then the solution can be obtained by using linear programming. If there are many linear variables, then the problem of finding test cases can become NP complete. Also, the symbolic representation for non-linear predicates becomes complex.
- **Function minimization methods**: perform an exploratory search in which the selected input variables are modified by a small amount and submitted to the program. Hence, function minimization methods are dynamic in nature since they are based on program execution.
- **Pseudo-random generators (Metaheuristics):** is a high-level strategy that guides other heuristics in the search for feasible solutions. Their use to generate test data can provide results in a reasonable time. Several metaheuristics have been suggested for path coverage, besides statement and branch coverage. One of the methods that has recently been used for path coverage is the Genetic Algorithm (GA). GA is a global search heuristic that works on the principle of Darwin's theory of natural evolution i.e. survival of the fittest in the population. Although this methodology is gaining much popularity, but it is heavily based on stochastic processes.

The partial automation of test data generation, test execution and test checking can lead to significant saving of time and more thorough testing.

2.5.1 Genetic Algorithm

Genetic algorithm (GA) is a heuristic that mimics the evolution of natural species (*survival of the fittest*) in searching for the optimal solution to a problem. Genetic algorithms have become increasingly important to researchers in solving difficult problems since they can provide feasible solutions in limited amounts of time. GAs were first proposed by Holland in 1975 and have been successfully applied to the fields of optimization, neural networks, fuzzy logic controllers and many others.

In test-data generation applications, the solution sought by the genetic algorithm is test data that brings about the execution of a given statement, branch, path, or definition-use pair in the program under test.

Genetic algorithms use the following principal genetic operators to produce the successive generation:

- **Selection:** operation chooses some offspring's for survival according to predefined test adequacy criterion (or fitness function).
- **Crossover:** operation generates offspring's from two chosen individuals in the population by exchanging some genes in the two individuals. The offspring thus inherits some characteristics from each parent.
- **Mutation:** operation generates offspring by randomly changing one or several genes in an individual. It is analogous to biological mutation.

In testing applications, genetic algorithms look for test scenarios that cover certain branches of a program. The first generation of test cases is generated at random. Next, the

generated test cases are fed to the program for execution. The executed results are evaluated by a fitness function to determine which test cases should survive to produce the next generation. The crossover and mutation operators generate the next generation of test cases. A simple genetic algorithm uses a single crossover operator and a single mutation operator throughout the entire genetic process. The procedure is repeated until the termination criterion is satisfied. The number of individuals of one generation should be large enough to maintain diversity, yet small enough to avoid excessive number of tests.

Genetic algorithms are well suited to generate test cases for both conventional and object-oriented programs. It is advantageous over more established search-based test case generation approaches because the program under test is represented and altered as a fully functional computer program. Genetic programming uses a tree-shaped data structure which is more directly comparable and suitable for being mapped instantly to abstract syntax trees commonly used in computer languages and compilers. These structures can be manipulated and executed directly. In addition, tree structures make more operations possible compared to linear structures. This speeds up the evolutionary program generation process.

The following algorithm represents a standard simple genetic algorithm that can be used to test populations P, P1, P2 and P3 of feasible test scenarios.

```
1    Initialize_random (P);
2    Fitness_function (P);
3    While (! Stopping_criterion)
4    Do
```

```
5           P1 = selection (P);
6           P2 = recombination (P1);
7           P3 = mutation (P2);
8           Fitness_function (P3);
9           P = merge_population (P3, P);
10      end while
```

The genetic algorithm discussed above begins with a random initial population by selecting the test data randomly from the domain of the input variables used in the program. Fitness function measures how well the initial population satisfies the test criterion. In this case, test criterion is the coverage of program's branches in the control flow graph. A while loop controls the generation of next population until the stopping criterion has been satisfied. Selection function chooses the individuals to be recombined and mutated out of the initial population. Recombination reproduces the selected individuals in order to produce new individuals. This is also called a crossover. Mutation introduces a small and infrequent random change to each newly created individual. Mutation is used to maintain genetic diversity from one generation of population to the next. For the next generation, the old and the new populations are merged, thereby retaining the best individuals. The process of selection, reproduction and evaluation is referred to as one generation, and these steps are repeated until the stopping test criterion is satisfied. Fitter individuals are favored in the recombination and selection process so that in the subsequent generations, the test criterion is more likely to be satisfied. Full branch coverage can be obtained by selecting every branch as the target and solving through an individual search process.

The performance of genetic algorithms is influenced by the choice of mutation operators. A considerable amount of evidence shows that in many applications the mutation operator is the key to the success of the genetic algorithm. Determining which mutation operator to use is quite difficult and is usually learned through experience or by trial-and-error. That is, experiments must be done using all candidate mutation operators to find the best operator for a specific problem, which consumes considerable time and computation resources.

Some of the most common mutation operators used in testing literature of genetic algorithms include [Chen, Hong and Wang 2000]:

- **Mutation of input value:** involves replacing a value with another random value of the same type.
- **One point crossover:** involves taking the midpoint of current and new population.
- **Boundary value:** involves replacing the chosen value with either the upper or lower bound (chosen randomly).
- **Uniform value:** involves replacing the chosen value with a uniform random value selected between upper and lower bound.
- **Non-uniform value:** involves replacing the chosen value with a non-uniform random value selected between upper and lower bound.

The mutation operators for bit values include:

- **1 & 0 exchange:** involves changing the bit values of 0 to 1 and 1 to 0.
- **Swapping:** involves arbitrarily exchanging any two bits.
- **Inversion:** involves inverting the bit order in an interval.

- **One point mutation:** involves changing one bit in a chromosome.

3. Research Goal

When beginning any software project, a goal statement (or thesis statement) is required to provide a charter under which the project will operate. The goal statement must be a direct answer to the problem attempting to be solved.

The strategies discussed above for unit testing of conventional and object-oriented software, although useful, have limited automatic test case generation support in their corresponding testing tools. Most of the strategies use a graphical notation to represent the program's flow of control. Test cases are generated from the corresponding graph using algebraic specification, statistical inferences or in most cases, random number generators.

The focus of this thesis work would be to automate the process of generating test cases for the white-box testing technique known as basis path testing or simply path testing. My research will look into the design and development of a test data generation tool for basis path testing using genetic algorithms. The tool will consist of modules for control flow graph construction, test path selection and test data generation. This interactive tool will construct a dynamic control flow graph from the program's pseudocode entered by the user. The proposed tool will then implement an algorithm [Salloum and Salloum 2006] to compute a basis set of linearly independent control paths from the control flow graph, and lastly, we implement a genetic algorithm [Tonella 2004] to find test data to cover maximum paths in the basis set. We evaluate the performance of different mutation operators in the genetic algorithm on the basis path coverage criterion.

The greatest merit of using genetic algorithm in program testing is its simplicity. The quality of test cases produced by genetic algorithms is higher than the quality of test cases produced by random test data generation [Lin and Yeh 2000].

This thesis work is needed for various reasons:

- Past tools on control flow graph construction e.g. DATRIX [Robillard and Simoneau 1993] use algebraic specifications or statistical inferences to generate program's control flow graph.
- Traditional test case generation algorithms do not perform efficiently with basis path testing.
- There is a need to determine which mutation operators should be used in the genetic algorithm to achieve higher basis path coverage.
- A great deal of work is being done on the transformation of control flow in re-engineering, restructuring, documentation, optimization etc.
- Metric measurement needs a more rigorous framework.

The representation of control flow graph will be concise and readable. It will keep all pertinent control-related information in the pseudocode as it appears. The goal is to represent control flow as it is. Hence, the iconic control flow graph provides an exact transformation of the pseudocode. It is a basis for control flow visualization, path crossing and path computation. The representation is programming-language independent since pseudocode of program is used. The iconic control flow graph construction is dynamic and automated.

Hence, the expected outcome is a unit-testing tool for basis path testing that can be applied to conventional testing and then modified appropriately to suit the class testing of

object-oriented programs. The steps to achieve the research goal are provided in the next section to clearly understand the goal of the thesis work.

4. Methodology

Graphs of programs have been used for many years in several areas of computer science, including software testing and compiler optimization. In these areas of application, any program or program fragment written in some procedural language can be translated into a flow graph.

There is some evidence that the visualization of control flow graph can provide better understanding of the programming process [Robillard and Simoneau 1993]. It can improve testing and validation by making a formal link between algorithm implementation and specification. It can reduce maintenance by showing the complexity of the control flow. It can help establish norms and standards in quality control programs. Furthermore, it can also provide better understanding and definition of the cohesion and coupling of software units.

Despite growing interest in software metrics, control flow representation is still empirical. Although mathematical support exists for studying and manipulating control flow, there is no systematic method either for representing or deriving control flow. Some software metrics are based on the topology of such graphs. Control flow graph construction is a non-trivial process and has usually been underestimated.

Several white-box testing strategies have been proposed in the literature using the control flow graph model: loop testing, branch testing, statement testing, du-path testing and basis set of paths testing. Each strategy is based on generating a set of test vectors for

a given program that exercise a specific set of paths for the corresponding control flow graph.

It is the objective of my research to design and develop an interactive test case generation tool for basis path testing using genetic algorithms. The basis path testing strategy is based on generating a set of maximal linearly independent paths from the control flow graph of a given program (each path starts with start node and ends with the stop node).

All algorithms in the thesis project are implemented in Java, so that they can be integrated in Eclipse Integrated Development Environment (IDE) for future research and development of this work. The tool will generate a control flow graph from the program's pseudocode entered by the user using standard pseudocode notations [Pressman 2003]. The proposed tool will then identify a basis set of linearly independent control paths and generate test data and test cases automatically to force the execution of maximum paths in the basis set.

Basis path testing methodology can be applied to a program's pseudocode. The following series of steps can be defined to achieve the research goal stated above.

a) Using program's pseudocode as a foundation, draw the corresponding control flow graph.
b) Determine the cyclomatic complexity of the resultant graph.
c) Determine a basis set of linearly independent paths.
d) Generate test data to cover all the independent paths in the graph.
e) Prepare test cases that will force the execution of maximum paths in the basis set.

I will roughly follow the above hierarchy in the methodology phase, spending more time in some areas and less in others as fits the problem I am attempting to solve.

4.1 Drawing Control Flow Graph

Basis path testing uses control flow graph (or flow graph) to depict the logical control flow of the program. Control flow graph is a directed graph that is based on nodes and arcs. A node can be a:

- **Procedure node:** that represents one or more sequential (or computation) statements. It is characterized by one edge emanating from it.
- **Predicate node:** that represents a decision point or condition. It is characterized by two or more edges emanating from it.

The arcs (also called edges or links) in the flow graph represent the flow of control in the program. An edge must terminate at a node, even if the node does not represent any procedural statements. Areas bounded by edges and nodes are called regions. The area outside the graph is also counted as a region.

Pressman [2003] suggests the following sequence of activities for basis path testing: Pseudocode -> Flow Chart -> Control Flow Graph. Our program constructs the control flow graph directly from the pseudocode of a program. The intermediate step of drawing the flow chart can be eliminated as our program handles the logic of converting a flow chart into a control flow graph.

The control flow graph of a given program can be built from the basic/prime control flow graph notations, using only two operations: sequencing and nesting [Salloum and Salloum 2006]. Programming constructs can be uniquely represented by the prime

control flow graph notations. The prime control flow graph notations given by Pressman [2003] and Salloum and Salloum [2006] are shown in Figure 3:

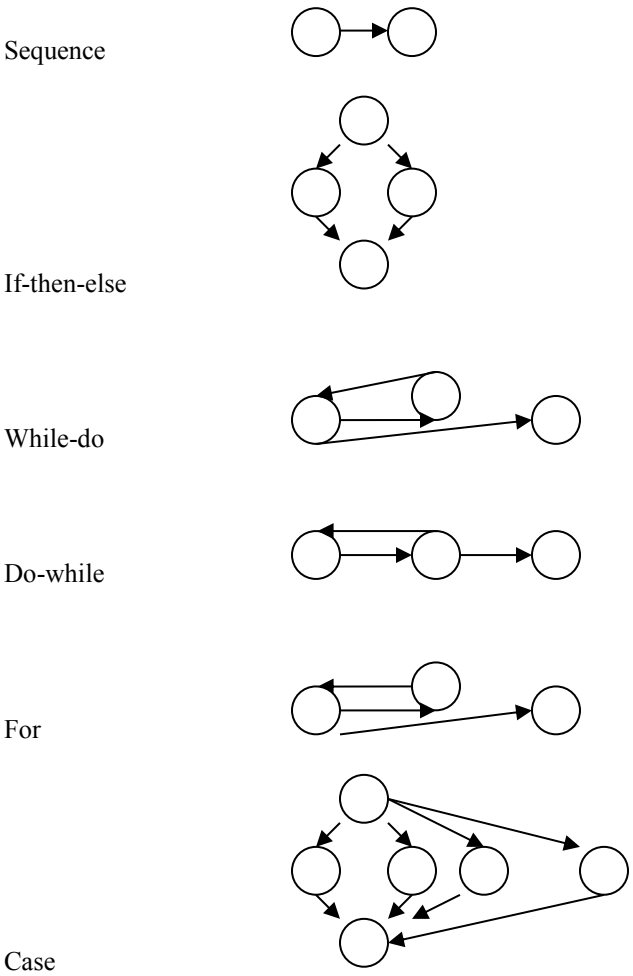

Figure 3: Control flow graph notations

Each circle in the above figure represents one or more non-branching source code statements. Our program implements the following programming constructs: sequence (statements), if-then-else loop, while-do loop, do-while loop and for loop.

When compound conditions are encountered in a procedural design, the generation of control flow graph becomes more complicated. A compound condition occurs when one or more Boolean operators (e.g. logical AND, OR) are present in a conditional statement.

Our program constructs a separate node for each of the compound conditions. The symbols and construction rules of control flow graph have been stated above. It is recommended to draw the control flow graph where the logical control structure of a module is complex. Hence, control flow graph allows tracing of program paths more readily.

4.2 Determining Cyclomatic Complexity

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program [Pressman 2003]. The value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program's control flow graph. This provides us with an upper bound for the number of tests that must be generated and executed to ensure that all statements in a given program have been executed at least once.

Cyclomatic complexity provides us with an answer to the number of paths to test in a given program. Hence, it is related to the branch coverage criterion of white-box testing. Complexity, according to McCabe, is computed in one of the three ways:

- The number of closed regions of the control flow graph.
- Cyclomatic complexity, V(G), for a flow graph G, is defined as

  $$V(G) = E - N + 2$$

  where E is the number of edges in the control flow graph and N is the number of control flow graph nodes.
- Cyclomatic complexity, V(G), for a control flow graph G, is also defined as

  $$V(G) = P + 1$$

  where P is the number of predicate nodes contained in the control flow graph G.

The cyclomatic complexity, V(G), of the resultant control flow graph in the project is determined by applying the last definition given above.

4.3 Determining a Basis Set of Linearly Independent Paths

Basis set of paths are linearly independent paths that are sufficient to express any other paths of the control flow graph. An independent path is a path through the program (from the entry node to the exit node) that introduces at least one new set of processing statements or a new condition. In other words, a collection of paths is called independent if each path has a node or edge not in any of the other paths. Hence, in a flow graph, an independent path moves along at least one edge that has not been traversed before the path is defined. If tests are designed to force the execution of paths in the basis set, then every statement in the program would be guaranteed to execute at least one time and every condition would be executed on its true and false sides. According to Pressman [2003], the basis set is not unique. A number of different basis sets can be derived for a given procedural design.

The value of cyclomatic complexity, V(G), provides the number of linearly independent paths through a program control flow graph. The predicate nodes are identified to aid in the derivation of test cases.

The paper by Salloum and Salloum [2006] presents an efficient and formal algorithm to generate a basis set of paths for a given control flow graph. The thesis project makes use of the algorithm by Salloum and Salloum [2006] to generate a basis set of paths by visiting every node and edge of the control flow graph at least once, as follows:

For a given control flow graph, the algorithm generates a tree whose paths from the root to the leaves represent a basis set of paths. The tree is constructed by visiting every node and edge at least once as follows:

1 Define a tree data structure

2 node_appearance_count = 0;

3 n = head of linked list

4 While (n != end node or node_appearance_count > 1)

5 For every node n in the linked list, create a new node of tree for every node adjacent (directly linked through node_link1 and node_link2) to n in the control flow graph and draw a branch that connects n to its adjacent node.

6 For every node m of the linked list that is not the end node, visit the nodes along any simple path (no cycle) from that node to the end node and create necessary nodes and branches in the tree.

7 node_appearance_count ++; (on the same path)

8 n = n.getnext();

9 endwhile

It has been observed that the above algorithm generates a basis set of paths for a given control flow graph in O (max (n, e)) time complexity where n is the number of nodes and e is the number of edges in the control flow graph. The proof of this observation is given in Salloum and Salloum 2006.

4.4 Generating Test Data and Test Cases

The thesis project uses a genetic algorithm (GA) to generate test cases for basis path testing in a generic usage scenario [Tonella 2004]. The thoroughness of testing can by assessed by means of a coverage criterion. Traditional coverage criteria (white-box

testing) can be used, such as structural (e.g. statement, branch) coverage or data flow (e.g. all-uses) coverage. The genetic algorithm in our program will try to generate test cases for a method under test until a satisfactory level of path coverage (i.e. 95%-100% of method's path coverage from the basis set of paths) is attained.

Test cases are generated to force the execution of maximum number of paths in the basis set. Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested. Each test case is executed according to the condition in the predicate node.

Test cases are generated for every statement in the given pseudocode. It is assumed that the parameters and variables in a method call use the built-in types. A lookup table stores the definition of variables in the given pseudocode. We follow the strongly typed convention of Java language i.e. all variables must first be declared before they can be used. Lookup table holds the following information: variable type, variable name and variable value (if any). Variable type can be any of the eight primitive (built-in) data types in Java: integer, long, float, double, short, byte, boolean and character. String is not a primitive type, as it is made up of characters in double quotes. Variable value, if entered, can be either positive or negative. Our program stores the possible range of values for the primitive data types. This range of values, together with the entered variable value is used to generate the initial test data in the following manner. A number can be randomly chosen in an interval ranging from low to up. The low and up for positive variable value is 0 and (variable value + 1000) respectively. The low and up for negative variable value is (variable value – 1000) and 0 respectively. If no variable value

is specified then a random number from the range of primitive data type constitutes the initial population.

The macro steps of a slightly modified genetic algorithm to generate test data so as to satisfy the given coverage criterion for the method under test is summarized below [Tonella 2004].

```
TestCaseGeneration (basis set of paths of MethodUnderTest: Method)
1       target_paths_to_cover = cyclomatic_complexity of Method
2       target_nodes_to_cover = size of linked list
3       current_population = generate_random_population (number of variables in Method)
4       while (target_paths_to_cover != 0)
5               t = select_target_path()
6               attempts = 0
7               while (!path_covered (t) and attempts < maxAttempts)
8                       execute testcases in current_population
9                       if (path_covered(t))
10                              break
11                      compute fitness(t) for test cases in current_population
12                      extract new_population from current_population according to fitness(t)
13                      mutate new_population
14                      current_population = new_population
15                      attempts = attempts + 1
16              endwhile
17              target_paths_to_cover--;
18      endwhile
```

The basis set of paths of the method under test is passed to the genetic algorithm as a parameter. The very first step is to specify a set of targets (i.e. paths) to be covered. This is equal to the cyclomatic complexity of the control flow graph. The next step is to randomly generate initial population (current_population) of the genetic algorithm. The random generator takes the respective range of different variables present in the method under test to generate the initial population as discussed above.

The steps at line 4 and 5 are responsible for selecting the next node that is not covered in a given path and applying the genetic algorithm to it. The variable maxAttempts on line 7 indicates the maximum number of evolution cycles that can be run for each target hence restricting the amount of time to find the best results. The value of maxAttempts used by our program is 50.

The evolution cycle occurs between lines 7 and 16. In each cycle, the test cases in current population are executed, possibly covering the uncovered targets and then the uncovered targets are updated based on the results of the run. The inner loop exits after the target nodes in a given path have been covered. Otherwise, the level of fitness against the target is calculated for each of the test cases. A new population (new_population) is created from the current one. Each member has a probability of being selected, which is proportional to its fitness with respect to the current target. Mutation is then applied and another attempt to cover the target is started.

Each time a target is covered, the test case covering it is added to the result set as one of those necessary to achieve the final level of coverage. Thus the resulting set might be redundant. In other words, the test cases that are added later to the set might cover some

of the previous targets also. Performing a post processing of the resulting test cases can minimize this redundancy.

The parameter that can be set externally is maxAttempts. It can be augmented if the achieved coverage level is not satisfactory.

The critical choice in the above algorithm is the level of fitness that determines the probability of each individual test case to survive and participate in an evolved form in the next population. There are several possible options for calculating the fitness but the one that is used here is based on path coverage criterion. The fitness of a test case is obtained from the control flow graph edges that are traversed during its execution. Specifically, given the transitive set of control flow dependencies that lead to the given target, the proportion of edges that are exercised during the execution of a test case measure its fitness. Thus the fitness will be closer to 1 for the test cases that traverse most of the control flow graph edges that leads to the target, while it will be close to 0 when the execution follows a path that does not intersect with the edges leading to the target.

The choice of mutation operator is also critical to the success of the genetic algorithm since it diversifies the search directions and avoids convergence to local optima. The earliest genetic algorithms use only one mutation operator in producing the next generation.

We tested the performance of each of the following common mutation operators individually on the genetic algorithm's maximum path coverage criterion:

- Mutation of input value
- One point crossover
- Boundary value

- Uniform value
- Non-uniform value

The rules for input generation used by the mutation of input value include:

- Integer, Long, Float, Double: are randomly chosen in the interval [-2147,483,648, 2147,438,648] (inclusive).
- Boolean: values of true and false are randomly chosen with equal probability (0.5).
- Byte: values are randomly chosen in the interval [-128, 127] (inclusive).
- Short: values are randomly chosen in the interval [-32768, 32767] (inclusive).
- Character: values are randomly chosen in the interval [\u0000,\uffff] or 0 to 65535 inclusive.

The rules for input generation used by the boundary value include:

- Integer, Long, Float, Double: are randomly given the lower bound (-2147,483,648) or upper bound (2147,438,648) values.
- Boolean: are randomly given the lower bound (0) or upper bound (1) values.
- Byte: are randomly given the lower bound (-128) or upper bound (127) values.
- Short: are randomly given the lower bound (-32768) or upper bound (32767) values.
- Character: are randomly given the lower bound (0) or upper bound (65535) values.

The rules for input generation for the uniform value use equal probability (0.5,0.5) for random value generation. The rules are:

- Integer, Long, Float, Double: are given uniform random value chosen in the interval [-2147,483,648, 2147,438,648] (inclusive).
- Boolean: are given uniform random value of true and false chosen with equal probability (0.5).
- Byte: are given uniform random value chosen in the interval [-128, 127] (inclusive).
- Short: are given uniform random value chosen in the interval [-32768, 32767] (inclusive).
- Character: are given uniform random value chosen in the interval [\u0000,\uffff] or 0 to 65535 inclusive.

The rules for input generation for the non-uniform value use unequal probability (0.7,0.3) for random value generation. The rules are:

- Integer, Long, Float, Double: are given non-uniform random value chosen in the interval [-2147,483,648, 2147,438,648] (inclusive).
- Boolean: are given non-uniform random value of true and false.
- Byte: are given non-uniform random value chosen in the interval [-128, 127] (inclusive).
- Short: are given non-uniform random value chosen in the interval [-32768, 32767] (inclusive).
- Character: are given non-uniform random value chosen in the interval [\u0000,\uffff] or 0 to 65535 inclusive.

Sometimes, a single mutation operator does not achieve the best mutation results; instead several mutation operators may have to be applied to achieve the best

performance. This observation led us to test the performance of any two mutation operators (out of the five operators) applied simultaneously on the genetic algorithm's maximum path coverage criterion. The set of mutation operators that provides the best results for path coverage criterion are mutation of input value and one point crossover when applied simultaneously.

The test suite produced by the above algorithm provides a satisfactory level of path coverage. When the generated test cases do not reach a satisfactory level of coverage using default values, the maxAttempts parameter can be increased.

4.5 Pseudocode of the Program

The pseudocode of the thesis project's program is as follows:

```
1       While (data != "end" or "END")
2               Read one line at a time from the program;
3               If (data ends with semicolon)
4                       Statement_procedure ();
5               If (data starts with "if" or "IF")
6                       IF_procedure ();
7               If (data starts with "then" or "THEN")
8                       THEN_procedure ();
9               If (data starts with "else" or "ELSE")
10                      ELSE_procedure ();
11              If (data starts with "else if" or "ELSE IF")
12                      ELSEIF_procedure ();
13              If (data starts with "while" or "WHILE")
14                      WHILE_procedure ();
15              If (data starts with "do" or "DO")
```

```
16                      DO_procedure ();
17              If (data starts with “for” or “FOR”)
18                      FOR_procedure ();
19              If (data == “endif” or “ENDIF”)
20                      ENDIF_procedure ();
21              If (data == “enddo” or “ENDDO”)
22                      ENDDO_procedure ();
23              If (data == “endwhile” or “ENDWHILE”)
24                      ENDWHILE_procedure ();
25              If (data == “endfor” or “ENDFOR”)
26                      ENDFOR_procedure ();
27              Display_linklist();
28              Showtree();
29      endwhile
30      If (data == “end” or “END”)
31              END_procedure ();
32              Display_linklist();
33              Showtree();
34              Testpaths();
35              Testdata();
```

Statement_procedure ()

1 Store the variable type, variable name and variable value in a lookup table

2 If data read immediately before this ended with semicolon, then concatenate new data with data read immediately before.

3 Make a new node in linked list

4 Store the data in the node

IF_procedure ()

1 Save “if” substring;

2 Skip any leading whitespaces

3 string = index of “if” + 2 to len – 1;

4 Find_and_or (string);

THEN_procedure ()

1 Save “then” substring;

2 Skip any leading whitespaces

3 Statement_procedure (); //gathers statements that end with semicolon in one node

ELSE_procedure ()

1 Save “else” substring;

2 Skip any leading whitespaces

3 Statement_procedure (); //gathers statements that end with semicolon in one node

ELSEIF_procedure ()

1 Save “elseif” substring;

2 Skip any leading whitespaces

3 string = index of “elseif” + 6 to len – 1;

4 Find_and_or (string); // involves the same steps as IF_procedure ()

WHILE_procedure ()

1 Save “while” substring;

2 Skip any leading whitespaces

3 string = index of “while” + 5 to len – 1;

4 Find_and_or (string); // involves the same steps as IF_procedure ()

DO_procedure ()

1 Save “do” substring;

2 Skip any leading whitespaces

3 Statement_procedure (); //gathers statements that end with semicolon in one node

FOR_procedure ()

1 Save “for” substring;

2 Skip any leading whitespaces

3 Separate for statement in 3 parts by looking for semicolons

4 Store variables in lookup array (type, name, value)

5 Make a node for the whole for statement

ENDIF_procedure ()

1 Create the link in the linked list by traversing linked list backwards to look for first “if”.

ENDDO_procedure ()

1 Create the link in the linked list by traversing linked list backwards to look for first “do”.

ENDWHILE_procedure ()

1 Create the link in the linked list by traversing linked list backwards to look for first “while”.

ENDFOR_procedure ()

1 Create the link in the linked list by traversing linked list backwards to look for first “for”.

END_procedure ()

1 Stop reading input data

Find_and_or (data)

1 index1 = look for index of “and” in data //returns first occurrence of and

2 index2 = look for index of “or” in data //returns first occurrence of or

3 case 1: if index1 and index2 contain nothing: //and & or not found

4 make a new node in linked list

5 save data in the new node

6 case 2: if only index1 returns a value: //and found

7 make a new node in linked list

```
8                   skip leading whitespaces
9                   string1 = store data substring from index 0 to index1-1 in the new node
10                  string2 = store substring index1+4 to len-1
11                  Find_and_or (string2);
12      case 3: If only index2 returns a value:          //or found
13                  make a new node in linked list
14                  skip leading whitespaces
15                  string1 = store data substring from index 0 to index2 –1 in the new node
16                  string2 = store substring index2 + 3 to len – 1
17                  Find_and_or (string2);
18      case 4: If both index1 and index2 return a value:  //both and & or found
19                  make a new node in the linked list
20                  index3 = min (index1, index2);
21                  string1 = store data substring from index 0 to index3 – 1 in new node
22                  string2 = store substring index3 + 3 to len – 1
23                  Find_and_or (string2);
```

Display_linklist ()

```
1       Node current = head;
2       while  (current != null)
3                   output node_number, node_link1, node_link2, marker, text;
4                   current = current.getNext();
5       endwhile
```

Showtree()

```
1       Node current = head;
2       Node p = head;
3       while (current != null)
```

```
4               while (p != null)
5                       if (p.node_link1 != 0 or -1)
6                               Node r = head;
7                               while (r != null)
8                                       if (p.node_link1 == r.node_number)
9                                               draw arrowed line between the nodes p and r
10                                      r = r.getNext();
11                              endwhile
12                      endif
13                      if (p.node_link2 != 0 or –1)
14                              Node r = head;
15                              while (r != null)
16                                      if (p.node_link2 == r.node_number)
17                                              draw arrowed line between the nodes p and r
18                                      r = r.getNext();
19                              endwhile
20                      endif
21              p = p.getNext();
22              endwhile
23              draw circle for current node
24              write the node_number in the circle
25              curent = current.getNext();
26      endwhile
```

Testpaths()

```
1       Node current = head;
2       String t = String.valueOf(current.node_number);
```

```
3       Paths(current,t);
```

Paths(Node n, String t)

```
1       Node current = null;
2       if(n.node_number == listCount)
3               print t;
4       else
5               if(n.node_link1 != 0 && n.node_link1 != -1)
6                       current = getNode(n.node_link1);
7                       String rough = String.valueOf(current.node_number);
8                       String t1 = t.concat(rough);
9                       if(current != null && count < 3)
10                              paths(current, t1);
11              endif
12              if(n.node_link2 != 0 && n.node_link2 != -1)
13                      current = getNode(n.node_link1);
14                      String rough = String.valueOf(current.node_number);
15                      String t2 = t.concat(rough);
16                      if(current != null && count < 3)
17                              paths(current, t2);
18              endif
19      endelseif
```

Testdata()

```
1       Operator1 = randomvalue(lower_bound, upper_bound);
2       Operator2 = midpoint(old_value, new_value);
3       Operator3 = boundary(lower_bound, upper_bound);
4       Operator4 = uniform(lower_bound, upper_bound);
```

```
5       Operator5 = nonuniform(lower_bound, upper_bound);
6       if (variable_type == integer)
7               if (variable_value != null)
8                       if (count == 0)
9                               int n = operator1;
10                      if(count == 1)
11                              int n = operator2;
12              endif
13      endif
14      if(variable_type == long)
15              Repeat lines 7 to 13 with modified variable range in all operators
16      if(variable_type == double)
17              Repeat lines 7 to 13 with modified variable range in all operators
18      if(variable_type == float)
19              Repeat lines 7 to 13 with modified variable range in all operators
20      if(variable_type == byte)
21              Repeat lines 7 to 13 with modified variable range in all operators
22      if(variable_type == short)
23              Repeat lines 7 to 13 with modified variable range in all operators
24      if(variable_type == char)
25              Repeat lines 7 to 13 with modified variable range in all operators
26      if(variable_type == boolean)
27              Repeat lines 7 to 13 with modified variable range in all operators
```

4.6 Program Facts

Some facts relating to the thesis program are discussed in this section.

4.6.1 Control Flow Graph

The focus is on constructing control flow graph from the pseudocode of a program. Our approach does not make control flow graph from code directly because code can be logically wrong and we would not have anything to verify it. Our program accommodates the following looping constructs: if-then-else, do-while, while-do and for loop. It works on the keywords in pseudocode i.e. if, then, else, elseif, endif, do, while, enddo, endwhile, for, endfor, and, or and end.

4.6.2 Syntax Errors

We are not limiting our program to any specific language hence no syntax errors are considered.

4.6.3 Semantic Errors

We are using pseudocode of program (line by line) in order to avoid semantic errors. Only the logical flow of functions is looked at.

4.6.4 Data Structure

The procedure for deriving the control flow graph and even determining a set of basis paths can be automated. To develop a software tool that assists in basis path testing, a data structure called a linked list is used. Linked list is used to store both the predicate and procedure nodes created. Mathematical algorithms when applied to linked lists can be used to partially or fully automate the design of test cases. Linked list can store many parameters hence allowing full automation of test case generation.

Linked list is much more powerful than the graph matrix data structure proposed by Pressman [2003] to construct control flow graphs.

Linked list is one of the fundamental data structures. It consists of a sequence of nodes, each containing arbitrary data fields of same type and one or two references (or pointers) pointing to the next and/or previous nodes. Linked list permits insertion and removal of nodes at any point in the list in constant time, but does not allow random access. On the other hand, a graph matrix is a square matrix whose size (number of rows and columns) is equal to the number of nodes in the flow graph. The matrix entries correspond to connections (an edge) between nodes. Matrix entries can be accessed randomly by specifying the index. Graph matrix is nothing more than a tabular representation of a flow graph.

The principle benefit of linked list over a graph structure is that the order of linked items may be different to the order that data items are stored in memory or on disk, allowing the list of items to be traversed in a different order. Elements can be inserted into linked lists indefinitely, while a matrix will eventually either fill up or need to be resized, an expensive operation that may not even be possible if memory is fragmented. Similarly, a matrix from which many elements are removed may become wastefully empty or need to be made smaller.

Linked list structure is often preferred to represent sparse graphs, that is, graphs with few edges, as it has smaller memory requirements and it does not use any space to represent edges which are not present. Graph structures on the other hand provide faster access for some applications but can consume huge amounts of memory if the graph is very large.

In general, linked list is beneficial for a dynamic collection, where elements are frequently being added and deleted, and the location of new elements added to the list is significant.

The complexity to insert/delete data nodes in linked list is O (1). The complexity to index elements in linked list is O (n).

The control flow graph in our program is not a fully connected graph. Hence, it justifies the use of a linked list. Statements in the pseudocode are stored systematically in the nodes of the linked list. Every node in the linked list has the following data members: text, marker, node number, next node, node link1 and node link2. Text and marker are stored in string formats. Text field holds the statements entered by the user whereas marker identifies the keyword (e.g. if, while, do etc.) used in the corresponding text field. The text field is parsed to separate the definition of variables in the program. The type, name and value of variables are stored in a lookup table. Linked list is updated dynamically as user enters the pseudocode statements. This is unlike the static graph matrix data structure that requires size declaration at the beginning. Linked list allows efficient traversal of data nodes when constructing the control flow graph in the shape of a tree.

4.6.5 Nested Loops

The pseudocode entered by the user may contain nested loops. Our program will prompt/remind the user to enter the corresponding loop end constructs (for example ENDWHILE, ENDFOR etc.) but it cannot force this due to the nature of nested loops. Our program is recursive due to the presence of nested loops.

4.6.6 Display

The program provides a screen for user to enter pseudocode (one line at a time). Data read from the terminal is then passed to the main program. The main program analyzes the data entered by the user, breaks the data based on the language keywords, stores the fragmented data in linked list (predicate or procedure nodes), extracts variables in the data, constructs control flow graph and generates test data at the end.

Our program displays the program nodes (both procedure nodes and predicate nodes) using circles of fixed radius. The circles are numbered according to the node number in the linked list, which in turn is taken from the pseudocode entered by the users. The control flow graph updates dynamically after each input from the user, hence making it interactive. Simpler output notation is used in the program to make it easy to understand.

The program contains separate windows for user input in the form of program pseudocode, control flow graph in the form of a tree diagram and program feedback in the form of test paths and test data. User can see the control flow graph update dynamically as the input is entered.

The program can also run in a Server/Client mode where a socket/network connection is opened between a server and a client. Many clients can attach to the server program at one time by opening separate connections. This is useful for processing simultaneous requests from the clients (users), in which case the server can perform heavy computations and send results to the respective client.

4.6.7 Test Data

Test data is generated from the genetic algorithm proposed by Tonella [2004]. Initial population is generated using random number generator in the Java language. Test data is

derived from the range of the variable types (for e.g. int, float, char, etc.) in Java language. Five mutation operators (mutation of input value, one point crossover, boundary value, uniform value and non-uniform value) are used in the genetic algorithm to introduce a small and random change in the next population.

4.7 Program Assumptions

Some assumptions used in the program are as follows:

- All small or all capital keywords
- A simple statement ends with a semicolon.
- A simple statement can have at most one next link.
- A conditional statement can have at most 2 next links.
- Program operates on keywords of the language – if, for, and, etc.
- Test data generation operates on range of primitive data types in the programming language – e.g. int, boolean, char, etc.

5. Evaluation of Results

Testing is one of the most time-consuming parts of software development process. Program testing cannot thoroughly expose all the errors in the program under consideration [Chen, H. Y., Chen, T. Y. and Tse, T. H. 2001]. In this sense, testing in general is an incomplete and undecidable problem. The development project under research cannot avoid this inherent limitation of testing.

The outcome of the research would be evaluated by verifying the independent control paths and test cases generated by the tool. Executed results from the test cases will be compared with the expected results (test oracle).

5.1 Control Flow Graph Construction

Control flow graph construction is straightforward and easy to apply to small programs. Larger programs need a software tool to compute and draw the control graph. The algorithm presented in this thesis gives acceptable results for drawing the control flow graphs from the pseudocode of a given program. Some of the programs used to test control flow graph construction include bubble sort, insertion sort, selection sort, quick sort's partition algorithm and sum of squares from Standish [1998]. It has been observed that control flow graph for large programs (with four or more predicate nodes) is generated in a reasonable time. The control flow graph updates dynamically and shows all the pertinent control flow information. The algorithm proposed in this thesis can be further improved and extended to apply to class level testing for object-oriented software testing.

5.2 Basis Set of Paths Selection

The algorithm implemented in the tool for generating a basis set of paths for a given control flow graph by using Salloum and Salloum's [2006] approach yields results in O (max (n, e)) time where n is the number of nodes and e is the number of edges in the control flow graph. The proof of the time complexity is based on induction and decomposition theory and is given in greater detail in Salloum and Salloum [2006].

5.3 Test Data/Test Case Generation

Genetic algorithms have been successfully applied to the problem of generating test cases for procedural programs. A population of individuals, representing the test cases, is evolved in order to increase a measure of fitness to satisfy a coverage criterion of choice.

Genetic programming approach achieves a much higher level of path coverage compared to random test case generation. Also, the genetic algorithm is much more complex compared to random test data generation method. The fitness function used for the experiments is based on the concept of control flow graph's path coverage.

Sometimes, a single mutation operator does not achieve the best mutation results; instead several mutation operators may have to be applied to achieve the best performance. Figure 4 shows a table comparing the performance of the five common mutation operators individually and in combinations of two. The table gives a list of test programs along with the number of paths in the basis set of each program. These test programs were run on our tool. The mutation operators are numbered 1 to 5 in the table where 1 is mutation of input value, 2 is one point crossover, 3 is boundary value, 4 is uniform value and 5 is non-uniform value.

Programs/Operators	Paths	1	2	3	4	5	1&2	1&3	1&4	1&5	2&3	2&4	2&5	3&4	3&5	4&5
Bubble Sort	4	2	2	1	1	1	4	2	2	1	1	2	1	1	0	2
Insertion Sort	6	2	3	1	1	1	6	2	2	2	2	2	2	2	1	2
Selection Sort	4	2	2	1	1	1	4	2	2	1	1	2	1	1	1	1
Partition Algorithm	4	2	2	1	1	1	4	2	2	1	2	2	1	1	0	2
Merge Algorithm	6	2	3	1	1	1	6	2	2	1	1	2	1	1	1	1
Binary Search	4	1	2	1	1	1	3	1	2	1	1	2	1	0	0	2

Programs/Operators	Paths	2&1	3&1	4&1	5&1	3&2	4&2	5&2	4&3	5&3	5&4
Bubble Sort	4	2	1	2	1	1	2	1	1	1	2
Insertion Sort	6	3	2	1	2	2	2	2	2	1	2

Selection Sort	4	1	2	2	1	1	1	1	1	1	1
Partition Algorithm	4	1	2	2	1	2	2	1	1	0	2
Merge Algorithm	6	3	2	3	1	1	2	1	1	1	1
Binary Search	4	2	1	2	1	1	1	1	1	0	2

Figure 4: Comparison of mutation operators

The performance of these five mutation operators when applied individually and when applied in combinations of two in achieving maximum path coverage is summarized in figure 4. Results from the experiments show that maximum (100%) level of path coverage is achieved when mutation of input value and one-point crossover mutation operators are applied simultaneously.

The genetic algorithm implemented in this thesis project for generating test data achieves optimal coverage of a method's branches within a reasonable computation time. The resulting test suites are generally compact. Initial test cases are generated randomly. The upper limit for execution time is between a few minutes and one hour typically.

The input values that are inserted into the test cases exercise the code under test in a sophisticated way. In order to achieve branch coverage, they explore all sides of the boundary conditions in alternative execution flows. Some of the conditions are hard to satisfy and a programmer expected to produce a test case for them would have a hard job.

Experimental results have been obtained by employing our tool for the unit testing of the following methods [Standish 1998]:

- Bubble sort
- Insertion sort
- Selection sort
- Partition algorithm of quick sort
- Merge algorithm
- Sum of squares
- Binary search

5.4 Test Results Evaluation

Currently, the most human-intensive activity in the testing process is related to the definition of test oracle for each test case. Such an activity is conducted after the test case generation phase. This step exposes faults in the code by executing the selected path with the test cases found above and determining whether their outputs are correct or not. The thesis project does not implement this step.

6. Conclusion and Limitations

Testing involves the identification of a limited number of tests out of nearly unlimited number of possible test scenarios. Identifying test cases typically follows predefined testing criteria such as path coverage criteria.

Unit testing of conventional or object-oriented software makes heavy use of white-box testing techniques, specifically basis path testing. Basis path testing uses the control flow graph of a program to generate a set of independent control flow paths.

This thesis focuses on combining basis path testing with genetic algorithms to generate test cases for basis path testing. In this thesis work, the basic steps followed are:

a) Control flow graph construction

b) Basis set of paths selection

c) Test case generation

Experiments on our tool show that the algorithms implemented for control flow graph construction, basis set of paths selection, and test case generation for a given program yield results in a reasonable time. The control flow graph updates dynamically and shows all the pertinent control flow information. Furthermore, the use of genetic algorithms for basis path testing proves extremely powerful. The simultaneous use of two known mutation operators, mutation of input value and one point crossover, achieves 100% basis path coverage.

The use of genetic algorithm with basis path testing has shown satisfying results in terms of path coverage, time taken to generate test cases and compactness of the resulting test suites. The solution presented is a simple one, and can be applied to any object-oriented programming language. A software tool has been implemented and the power of the approach demonstrated.

This work deals strictly with the internal control flow of a program, which constitutes only a part of the information contained in the pseudocode. The testing of a set of independent paths may not be sufficient in many cases. Additionally, control flow graph obscures important information present in the code, specifically feasible and infeasible paths.

7. Proposed Work for the Future

The techniques presented in this report are an initial step towards a more extensive test case generation application.

Future work would be to look into the feasibility of modifying this tool to generate test cases for object-oriented programs from sequence diagrams. In this scenario, the control flow graph can be modified to accommodate several functions in one class. The predicate nodes in the control flow graph will be converted to function nodes (function calls). When this strategy is applied to the sequence diagram, then it becomes class level testing (OO paradigm) instead of unit level testing (procedural programming).

This tool can be extended to incorporate data flow definition-use (DU) testing. Our program stores the definition of variables present in the pseudocode in a lookup table. This together with the subsequent use of variables in the control flow graph's predicate nodes can be applied to select test paths for data flow testing.

Recently, researchers are employing genetic algorithms to test programs. This work may be seen as an initial step towards unit testing of classes in a generic usage scenario. The mutation operators used in this thesis work can be applied in several other combinations in the genetic algorithm to check their impact on program's path coverage criterion. Furthermore, the use of other mutation operators not discussed in this report can be explored.

8. References

Abreu, B. T., Martins, E. and Sousa, F. L. 2004. *Automatic Test Data Generation for Path Testing using a New Stochastic Algorithm*. http://www.sbbd-sbes2005.ufu.br/arquivos/16-%209523.pdf (accessed August 3, 2007.)

Ali, S., Gu, D. and Zhong, Y. 1994. On testing of classes in object-oriented programs. *ACM Proceedings of the 1994 Conference of the Center for Advanced Studies on Collaborative Research* (October), 22-30.

Ambler, S. W. *The Full Life Cycle Object-Oriented Testing (FLOOT) Method*. Ambysoft, 2004-2006. http://www.ambysoft.com/essays/floot.html (accessed August 3, 2007).

Binder, R. V. *Testing Object-Oriented Systems: A Status Report*. http://www.rbsc.com/pages/ootstat.html (accessed August 3, 2007).

Bucanac, C. *Object-Oriented Testing Report*. IDE, University of Karlskrona/Ronneby, 1998. http://www.bucanac.com/documents/Object_Oriented_Testing_Report.pdf (accessed August 3, 2007).

Cha, S, D., Hong, H. S. and Kwon, Y. R. 1995. Testing of object-oriented programs based on finite state machines. *Proceedings of the IEEE 1995 Asia Pacific Software Engineering Conference* (December), 234-241.

Chan, F. T., Chen, H. Y., Chen, T. Y. and Tse, T. H. 1998. In black and white: an integrated approach to class-level testing of object-oriented programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 7, 3 (July), 250-295.

Chan, W. K., Lau, F. C. M., Liu, P. C. K., Luk, C. K. F. and Tse, T. H. Testing of object-oriented industrial software without precise oracles or results. *Communications of the ACM (accepted for publication)*.

Chatterjee , S. *Testing Java in an Object-Oriented Way*. Sun Microsystems, 1995-2006. O'Reilly and CollabNet. http://today.java.net/pub/a/today/2006/03/28/testing-java-object-oriented.html (accessed August 3, 2007).

Chen, H. Y. 2000. A dynamic approach for object-oriented cluster-level tests by program instrumentation. *IEEE International Conference on Systems, Man and Cybernetics* 2 (October), 1030-1035.

Chen, H. Y. 2002. The design and implementation of a prototype for data flow analysis at the method-level of object-oriented testing. *IEEE International Conference on Systems, Man and Cybernetics* 6 (October), 6-11.

Chen, H. Y. 2003a. Algorithm MSEL for determining observational equivalence in object-oriented class level testing. *IEEE International Conference on Systems, Man and Cybernetics* 2 (October), 1059-1063.

Chen, H. Y. 2003b. An approach for object-oriented cluster-level tests based on UML. *IEEE International Conference on Systems, Man and Cybernetics* 2 (October), 1064-1068.

Chen, H. Y. 2005. The refined algorithm ReCDRG to construct DRG graph for object-oriented class-level testing. *IEEE Proceedings of the 38th Annual Hawaii International System Sciences HICSS'05* (January), 317-321.

Chen, H. Y., Chen, T. Y. and Tse, T. H. 2001. TACCLE: a methodology for object-oriented software testing at the class and cluster levels. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 10, 1 (January), 56-109.

Chen, C., Gao, J., Hsia, P., Kim, Y. S., Kung, D., Song, Y. K., and Toyoshima, Y. 1995. Developing an object-oriented software testing and maintenance environment. *Communications of the ACM* 38, 10 (October), 75-87.

Chen, C., Gao, J., Hsia, P., Kung, D. and Toyoshima, Y. 1995. A test strategy for object-oriented programs. *IEEE Proceedings of the 19th Annual International Computer Software and Applications Conference* (August), 239-244.

Chen, C., Gao, J., Hsia, P., Kung, D. C. and Toyoshima, Y. 1998. Object-oriented software testing- some research and development. *Proceedings of the 3rd IEEE International High-Assurance Systems Engineering Symposium* (November), 158-165.

Chen, W. C., Hong, T.P., and Wang, H. S. 2000. Simultaneously applying operators in genetic algorithms. *ACM Journal of Heuristics* 6, 4 (September), 439-455.

Chen, M. H. and Kao, H. M. 1999. Testing object-oriented programs- an integrated approach. *10th International Symposium on Software Reliability Engineering, IEEE Conference Proceeding* (November), 73-82.

Chen, H. Y. and Tse, T. H. 2000. *Prototypes and Initial Experimentation on the Tools of the TACCLE Methodology*. http://www.cs.hku.hk/~tse/Papers/staccSupp.pdf (accessed August 3, 2007).

Cheston, G. A. and Tremblay, J. P. 2002. *Data Structures and Software Development in an Object-Oriented Domain.* Java Edition. Prentice Hall.

Choi, E. M. and Seo, K. I. 2006. Comparison of five black-box testing methods for object-oriented software. *IEEE 4th International Conference on Software Engineering Research, Management and Applications* (August), 213-220.

Dasiewicz, P. 2005. Design patterns and object-oriented software testing. *Canadian Conference on Electrical and Computer Engineering, IEEE Conference Proceeding* (May), 904-907.

Doong, R. K. and Frankl, P. G. 1994. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 3, 2 (April), 101-130.

Durand, M. H., Labiche, Y., Thevenod-Fosse, P. and Waeselynck, H. 2000. Testing levels for object-oriented software. *ACM Proceedings of the 22nd International Conference on Software Engineering* (June), 136-145.

Dwivedi, N. and Iyer, K. *Object-Oriented Testing @ NIIT*. Software Solutions Business. NIIT Ltd. http://www.softwaredioxide.com/community/paper/OOtesting@niit.doc (accessed August 3, 2007).

Fitzpatrick, K. J., Harrold. M. J. and McGregor, J.D. 1992. Incremental testing of object-oriented class structures. *Proceedings of the 14th International Conference on Software Engineering* (June), 68-80.

Gao, J., Hsia, P. and Kung, D. C. 1994. An object oriented testing and maintenance environment. *ACM Proceedings of the 1994 Conference for Advanced Studies on Collaborative Research* (October), 37-49.

Gross, H. G. and Seesing. 2006. *A Genetic Programming Approach to Automated Test Generation for Object-Oriented Software*. http://www.st.ewi.tudelft.nl/~gross/Publications/Seesing_2006.pdf (accessed August 3, 2007).

Harrold, M. J., Pargas, R. P. and Peck, R. R. 1999. Test data generation using genetic algorithm. *The Journal of Software Testing, Verification and Reliability 9, 4* (September), 263-282.

Hsia, P. and Kung, D. 1997. An object-oriented testing and maintenance environment. *IEEE Proceedings of the 19th International Conference on Software Engineering* (May), 608-609.

Johnson, M. S. 1996. A survey of testing techniques for object-oriented systems. *ACM Proceedings of the 1996 Conference of the Center for Advanced Studies on Collaborative Research* (November), 17-24.

Juristo, N., Moreno, A. M., and Strigel, W. 2006. Software testing practices in industry. *IEEE Software* 23, 4 (July-August), 19-21.

Kolling, M. and Rosenberg, J. 1998. Support for object-oriented testing. *IEEE Proceedings of Technology of Object-Oriented Languages* (November), 204-215.

Lin, J.C. and Yeh, P.L. 2000. Using genetic algorithms for test case generation in path testing. *IEEE Proceedings of the Ninth Asian Test Symposium* (December), 241-246.

Lin, Y.C., Pai, W.C., Shih, T.K. and Wang, C.C. 1997. An automatic approach to object-oriented software testing and metrics for C++ inheritance hierarchies. *IEEE Proceedings of 1997 International Conference on Information, Communications and Signal Processing 2* (September), 934-938.

McGregor J. D. and Sykes, D. A. 2001. *A Practical Guide to Testing Object-Oriented Software*. Addision-Wesley.

Object-Oriented Testing. http://courses.cs.tamu.edu/cpsc606/lively/OOTesting.ppt (accessed August 3, 2007).

Object-Oriented Testing Special Requirements. http://www.cs.bham.ac.uk/~pxc/se/2002/kv/Lecture5/OOTestingSpecialReq5.pdf (accessed August 3, 2007).

Pezz, M. and Young, M. 2004. Testing object oriented software. *IEEE Proceedings of the 26th International Conference on Software Engineering ICSE'04* (May), 739-740.

Pollock, L. L. and Souter, A. M. 2000. OMEN: A strategy for testing object-oriented software. *ACM SIGSOFT Software Engineering Notes, Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis ISSTA'00* 25, 5 (August), 49-59.

Pressman, R. S. 2003. *Software Engineering: A Practitioner's Approach.* Fifth Edition. McGraw-Hill, New York.

Robillard, P. N. and Simoneau, M. 1993. Iconic control graph representation. *ACM Software – Practice and Experience* 23, 2 (February), 223-234.

Salloum, M. and Salloum, S. 2006. Efficient algorithm for generating a basis set of path for white-box testing. *International Conference on Computer Science and Applications* (June), 27-31.

Standish, T. A. 1998. *Data Structures in Java*. Addison Wesley, California.

Tonella, P. 2004. Evolutionary testing of classes. *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis* (July), 119-128.

Xie, T. 2004. Automatic identification of common and special object-oriented unit tests. *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (October), 324-325.

Appendix A: Contents of CD

The CD contains three directories. The Source directory contains the source code files for the project. The Test Directory contains the test programs that I used to test the project. The Documents directory contains various documents related to the project.

Source Directory

The Source directory contains a package named PathTesting and several subdirectories. This package contains the source code files for the thesis project. The file Server.java is the Graphical User Interface (GUI) for the application. The file LinkedList.java contains the definition and methods of the linked list data structure used by the application. The file TreeWindow.java displays the control flow graph output of the program. The package is entirely portable, not relying on any platform specific code. The thesis project has approximately 10000 lines of code (LOC) and 2K lines of documentation/comments.

There are several requirements to build this application. Firstly, Java Development Kit (JDK) is required. The project is built in JDK 5. Secondly, Eclipse Integrated Development Environment (IDE) needs to be installed. This can be downloaded for free from http://www.eclipse.org/downloads. The version of Eclipse software used in this project is 3.2. Thirdly, Java Runtime Environment (JRE) is required. This project uses Java 5 JRE. Java's modern object-oriented development environment allows easier genetic programming based test case generation.

Test Directory

The Test directory contains a set of programs I used to test the thesis project. The programs, taken from Standish [1998], include algorithms of bubble sort, insertion sort, selection sort, quick sort's partition algorithm, sum of squares etc.

Documents Directory

The Documents directory contains several documents of interest. First, Thesis.doc is this document. Test Case Generation for White-box Unit Testing.ppt is the presentation I used at the thesis defense. The results.doc sheet contains the validation run results. Finally, the failures.doc sheet categorizes each of the failures.

Appendix B: Experimental Results

Some experimental results obtained from the program include:

```
Void Compare()

1       int i = 10;
2       int j = 100;
3       if i < j
4               then i++;
5       endif
6       end
```

```
Please enter a new node:
{1, procedure, int i = 10;}

Please enter a new node:
{1, procedure, int i = 10;int j = 100;}

Please enter a new node:
Enter ENDIF/endif to exit if-then-else loop
{2, if, i < j}

Please enter a new node:
Enter ENDIF/endif to exit if-then-else loop
{3, then, then i++;}

Please enter a new node:
{4, endif, endif}

Please enter a new node:
{5, end, end}

The nodes in the tree are:
{1, 2, procedure, int i = 10;int j = 100;}
{2, 3, 4, if, i < j}
{3, 4, then, then i++;}
{4, 5, endif, endif}
{5, end, end}

Cyclomatic Complexity: 2

Test Paths:
1 2 3 4 5
1 2 4 5

i   10
j   100

i<j
```

Path 1 2 3 4 5 covered.

i 73
j 56

i<j

Path 1 2 4 5 covered.

Void BubbleSort() [Stanley 1998, p. 413]

```
1       int array [10];
2       int temp = 0;
3       boolean notdone = 0;
4       while notdone == 0
5               for int i = 0; i < 10; i++
6                       if array [i] > array [i + 1]
7                               temp = array [i];
8                               array [i] = array [i + 1];
9                               array [i + 1] = temp;
10                              notdone = 1;
11                      endif
12              endfor
13      endwhile
14      end
```

Please enter a new node:
{1, procedure, int array [10];}

Please enter a new node:
{1, procedure, int array [10];int temp = 0;boolean notdone = 0;}

Please enter a new node:
Enter ENDWHILE/endwhile to exit while-do loop
{2, while, notdone == 0}

Please enter a new node:
Enter ENDFOR/endfor to exit for-loop
{3, for, i < 10 ; int i = 0; i ++}

Please enter a new node:
Enter ENDIF/endif to exit if-then-else loop
{4, if, array > array + 1}

Please enter a new node:
{5, procedure, temp = array;}

Please enter a new node:
{5, procedure, temp = array;array = array + 1;}

Please enter a new node:

{5, procedure, temp = array;array = array + 1;array + 1 = temp;}

Please enter a new node:
{5, procedure, temp = array;array = array + 1;array + 1 = temp;notdone = 1;}

Please enter a new node:
{6, endif, endif}

Please enter a new node:
{7, endfor, endfor}

Please enter a new node:
{8, endwhile, endwhile}

Please enter a new node:
{9, end, end}

The nodes in the tree are:
{1, 2, procedure, int array [10];int temp = 0;boolean notdone = 0;}
{2, 3, 8, while, notdone == 0}
{3, 4, 7, for, i < 10 ; int i = 0; i ++}
{4, 5, 6, if, array > array + 1}
{5, 6, procedure, temp = array;array = array + 1;array + 1 = temp;notdone = 1;}
{6, 7, endif, endif}
{7, 2, endfor, endfor}
{8, 9, endwhile, endwhile}
{9, end, end}

Cyclomatic Complexity: 4

Test Paths:
1 2 3 4 5 6 7 2 8 9
1 2 3 4 6 7 2 8 9
1 2 3 7 2 8 9
1 2 8 9

array 215 207 19 224 128 916 345 929 620 949
temp 0
notdone 0
i 0

notdone == 0

array 215 207 19 224 128 916 345 929 620 949
temp 0
notdone 1.0
i 0

Path 1 2 8 9 covered

i < 10

```
array  20  3  44  41  49  12  34  43  4  3
temp  0
notdone  0.0
i  1
```

Path 1 2 3 7 2 8 9 covered

array [i] < array [i +1]

Path 1 2 3 4 6 7 2 8 9 covered

Path 1 2 3 4 5 6 7 2 8 9 covered

Void InsertionSort() [Stanley 1998, p. 391]

```
1       int array [10];
2       int j = 0;
3       int k = 0;
4       boolean notfinished = 0;
5       for int i = 1; i < 10; i++
6               k = array [i];
7               j = i;
8               if array [j - 1] > k
9                       then notfinished = 1;
10                      else notfinished = 0;
11              endif
12              while notfinished == 1
13                      array [j] = array [j - 1];
14                      j--;
15                      if j > 0
16                              if array [j - 1] > k
17                                      then notfinished = 1;
18                              endif
19                      else notfinished = 0;
20                      endif
21              endwhile
22              array [j] = k;
23      endfor
24      end
```

Please enter a new node:
{1, procedure, int array [10];}

Please enter a new node:
{1, procedure, int array [10];int k = 0;}

Please enter a new node:
{1, procedure, int array [10];int k = 0;int j = 0;}

Please enter a new node:

{1, procedure, int array [10];int k = 0;int j = 0;boolean notfinished = 0;}

Please enter a new node:
Enter ENDFOR/endfor to exit for-loop
{2, for, i < 10 ; int i = 0; i ++}

Please enter a new node:
{3, procedure, k = array [i];}

Please enter a new node:
{3, procedure, k = array [i];j = i;}

Please enter a new node:
Enter ENDIF/endif to exit if-then-else loop
{4, if, array [j - 1] > k}

Please enter a new node:
Enter ENDIF/endif to exit if-then-else loop
{5, then, then notfinished = 1;}

Please enter a new node:
Enter ENDIF/endif to exit if-then-else loop
{6, else, else notfinished = 0;}

Please enter a new node:
{7, endif, endif}

Please enter a new node:
Enter ENDWHILE/endwhile to exit while-do loop
{8, while, notfinished == 1}

Please enter a new node:
{9, procedure, array [j] = array [j - 1];}

Please enter a new node:
Enter ENDIF/endif to exit if-then-else loop
{10, if, j > 0}

Please enter a new node:
Enter ENDIF/endif to exit if-then-else loop
{11, if, array [j - 1] > k}

Please enter a new node:
Enter ENDIF/endif to exit if-then-else loop
{12, then, then notfinished = 1;}

Please enter a new node:
Enter ENDIF/endif to exit if-then-else loop
{13, else, else notfinished = 0;}

Please enter a new node:
{14, endif, endif}

Please enter a new node:
Enter ENDIF/endif to exit if-then-else loop
{15, else, else notfinished = 0;}

Please enter a new node:
{16, endif, endif}

Please enter a new node:
{17, endwhile, endwhile}

Please enter a new node:
{18, endfor, endfor}

Please enter a new node:
{19, procedure, array[j] = k;}

Please enter a new node:
{20, end,end}

The nodes in the tree are:

{1, 2, procedure, int array [10];int k = 0;int j = 0;boolean notfinished = 0;}

{2, 3, 18, for, i < 10 ; int i = 0; i ++}

{3, 4, procedure, k = array [i];j = i;}

{4, 5, 6, if, array [j - 1] > k}

{5, 7, then, then notfinished = 1;}

{6, 7, else, else notfinished = 0;}

{7, 8, endif, endif}

{8, 9, 17, while, notfinished == 1}

{9, 10, procedure, array [j] = array [j - 1];}

{10, 11, 15, if, j > 0}

{11, 12, 13, if, array [j - 1] > k}

{12, 14, then, then notfinished = 1;}

{13, 14, else, else notfinished = 0;}

{14, 15, endif, endif}

{15, 16, else, else notfinished = 0;}

{16, 8, endif, endif}

{17, 18, endwhile, endwhile}

{18, 19, endfor, endfor}

{19, 20, procedure, array[j] = k;}

{20, end, end}

Cyclomatic Complexity: 6

Test Paths:

1 2 18 19 20

1 2 3 4 5 7 8 9 10 11 12 14 15 16 8 17 18 19 20

1 2 3 4 7 8 9 10 11 12 14 15 16 8 17 18 19 20

1 2 3 4 5 7 8 17 18 19 20

1 2 3 4 5 7 8 9 10 11 13 14 15 16 8 17 18 19 20

1 2 3 4 5 7 8 9 10 15 16 8 17 18 19 20

array 9 24 23 21 21 8 31 8 5 46
k 0
j 0
notfinished 0.0
i 1

i < 10

Path 1 2 18 19 20 covered

array [j - 1] > k

notfinished == 0

j > 0

array [j - 1] > k

array 13 48 46 13 21 44 20 33 16 11
k 12
j 3
notfinished 1.0

i 4

Path 1 2 3 4 5 7 8 9 10 11 12 14 15 16 8 17 18 19 20 covered

Path 1 2 3 4 7 8 9 10 11 12 14 15 16 8 17 18 19 20 covered

Path 1 2 3 4 5 7 8 9 10 11 13 14 15 16 8 17 18 19 20 covered

array 32 49 15 19 14 44 13 40 42 0
k 11
j 10
notfinished 1.0
i 9

Path 1 2 3 4 5 7 8 9 10 15 16 8 17 18 19 20 covered

array 44 18 14 32 1 3 19 37 2 13
k 3
j 2
notfinished 0.0
i 6

array 47 10 49 49 40 42 19 16 18 9
k 5
j 5
notfinished 0.0
i 0

array 38 16 8 27 49 3 25 23 47 37
k 7
j 9
notfinished 1.0
i 3

array 1 27 37 14 12 11 23 17 30 38
k 14
j 4
notfinished 0.0
i 13

array 35 42 9 17 26 41 23 27 33 0
k 9
j 14
notfinished 0.0
i 15

array 19 18 15 30 20 21 5 31 34 8
k 14
j 1
notfinished 0.0
i 15

```
array  13  44  11  3  7  30  22  23  14  47
k  8
j  10
notfinished  1.0
i  9

Path 1 2 3 4 5 7 8 17 18 19 20 covered

Void SelectionSort()   [Stanley 1998, p. 379]

1       int array [10];
2       int i = 9;
3       int j = 0;
4       int temp = 0;
5       while i > 0
6               j = i;
7               for int k = 0; k < i; k++
8                       if array[k] > array[j]
9                               then j = k;
10
11                      endif
12              endfor
13              temp = array [i];
14              array [i] = array [j]
15              array [j] = temp;
16              i--;
17      endwhile
18      end

Please enter a new node:
{1, procedure, int array [10];}

Please enter a new node:
{1, procedure, int array [10];int i = 9;}

Please enter a new node:
{1, procedure, int array [10];int i = 9;int j = 0;}

Please enter a new node:
{1, procedure, int array [10];int i = 9;int j = 0;int temp = 0;}

Please enter a new node:
Enter ENDWHILE/endwhile to exit while-do loop
{2, while, i > 0}

Please enter a new node:
{3, procedure, j = i;}

Please enter a new node:
Enter ENDFOR/endfor to exit for-loop
```

{4, for, k < 10 ; int k = 0; k ++}

Please enter a new node:
Enter ENDIF/endif to exit if-then-else loop
{5, if, array [k] > array [j]}

Please enter a new node:
Enter ENDIF/endif to exit if-then-else loop
{6, then, then j = k;}

Please enter a new node:
{7, endif, endif}

Please enter a new node:
{8, endfor, endfor}

Please enter a new node:
{9, procedure, temp = array [i];}

Please enter a new node:
{9, procedure, temp = array [i];array [i] = array [j];}

Please enter a new node:
{9, procedure, temp = array [i];array [i] = array [j];array [j] = temp;}

Please enter a new node:
{9, procedure, temp = array [i];array [i] = array [j];array [j] = temp;i--;}

Please enter a new node:
{10, endwhile, endwhile}

Please enter a new node:
{11, end, end}

The nodes in the tree are:

{1, 2, procedure, int array [10];int i = 9;int j = 0;int temp = 0;}

{2, 3, 10, while, i > 0}

{3, 4, procedure, j = i;}

{4, 5, 8, for, k < 10 ; int k = 0; k ++}

{5, 6, 7, if, array [k] > array [j]}

{6, 7, then, then j = k;}

{7, 8, endif, endif}

{8, 9, endfor, endfor}

{9, 2, procedure, temp = array [i];array [i] = array [j];array [j] = temp;i--;}

{10, 11, endwhile, endwhile}

{11, end, end}

Cyclomatic Complexity: 4

Test Paths:

1 2 10 11

1 2 3 4 8 9 2 10 11

1 2 3 4 5 6 7 8 9 2 10 11

1 2 3 4 5 7 8 9 2 10 11

array 40 18 2 25 37 35 46 1 29 0
i 9
j 0
temp 0
k 0

i > 0

array 40 18 2 25 37 35 46 1 29 0
i 0
j 0
temp 0
k 0

Path 1 2 8 9 covered

k < 10

array 40 18 2 25 37 35 46 1 29 0
i 10
j 2
temp 0
k 2

Path 1 2 3 4 8 9 2 10 11 covered

array [k] > array [j]

array 44 2 15 49 20 44 38 44 42 28
i 4
j 8
temp 6

k 7

Path 1 2 3 4 5 6 7 9 2 10 11 covered

Path 1 2 3 4 5 7 8 9 2 10 11 covered

Void Partition() [Stanley 1998, p. 386]

```
1       int array [10];
2       int i;
3       int j;
4       int pivot = 0;
5       int temp = 0;
6       int middle = 0;
7       int p = 0;
8       middle = i + j / 2;
9       pivot = array [middle];
10      array [middle] = array [i];
11      array [i] = pivot;
12      p = i;
13      for int k = i + 1; k <= j; k++
14              if array [k] > pivot
15                      then temp = array [++p];
16                      array [p] = array [k];
17                      array [k] = temp;
18              endif
19      endfor
20      temp = array [i];
21      array [i] = array [p];
22      array [p] = temp;
23      end
```

Void sumSquares()

```
1       int n = 10;
2       int partialsum = 0;
3       int i = 1;
4       int temp = 0;
5       while i <= n
6               temp = i * i;
7               partialsum = partialsum + temp;
8               i++;
9       endwhile
10      end
```

Please enter a new node:
{1, procedure, int n = 0;}

Please enter a new node:
{1, procedure, int n = 0;int partialsum = 0;}

Please enter a new node:
{1, procedure, int n = 0;int partialsum = 0;int i = 1;}

Please enter a new node:
{1, procedure, int n = 0;int partialsum = 0;int i = 1;in temp = 0;}

Please enter a new node:
Enter ENDWHILE/endwhile to exit while-do loop
{2, while, i <= n}

Please enter a new node:
{3, procedure, temp = i * i;}

Please enter a new node:
{3, procedure, temp = i * i;partialsum = partialsum + temp;}

Please enter a new node:
{3, procedure, temp = i * i;partialsum = partialsum + temp;i++;}

Please enter a new node:
{4, endwhile, endwhile}

Please enter a new node:
{5, end, end}

The nodes in the tree are:
{1, 2, procedure, int n = 0;int partialsum = 0;int i = 1;in temp = 0;}
{2, 3, 4, while, i <= n}
{3, 2, procedure, temp = i * i;partialsum = partialsum + temp;i++;}
{4, 5, endwhile, endwhile}
{5, end, end}

Cyclomatic Complexity: 2

Test Paths:
1 2 3 2 4 5
1 2 4 5

n 0
partialsum 0
i 1

i<=n

n 28
partialsum 34
i 17

i<=n

Path 1 2 3 2 4 5 covered.

n 22
partialsum 82
i 41

i<=n

Path 1 2 4 5 covered.

Void BinarySearch()

```
1       int keys [10];
2       int low = 0;
3       int high = 9;
4       int middle = 0;
5       int x = 5;
6       while low < high
7               middle = (low + high) /2;
8               if x > keys[middle]
9                       then low = middle + 1;
10                      else high = middle;
11              endif
12      endwhile
13      keys[low] = x;
14      end
```

Please enter a new node:
{1, procedure, int keys [10];}

Please enter a new node:
{1, procedure, int keys [10];int low = 0;}

Please enter a new node:
{1, procedure, int keys [10];int low = 0;int high = 9;}

Please enter a new node:
{1, procedure, int keys [10];int low = 0;int high = 9;int middle = 0;}

Please enter a new node:
{1, procedure, int keys [10];int low = 0;int high = 9;int middle = 0;int x = 5;}

Please enter a new node:
Enter ENDWHILE/endwhile to exit while-do loop
{2, while, low < high}

Please enter a new node:
{3, procedure, middle = low + high / 2;}

Please enter a new node:

```
Enter ENDIF/endif to exit if-then-else loop
{4, if, x > keys [middle]}

Please enter a new node:
Enter ENDIF/endif to exit if-then-else loop
{5, then, then low = middle + 1;}

Please enter a new node:
Enter ENDIF/endif to exit if-then-else loop
{6, else, else high = middle;}

Please enter a new node:
{7, endif, endif}

Please enter a new node:
{8, endwhile, endwhile}

Please enter a new node:
{9, procedure, keys [low] = x;}

Please enter a new node:
{10, end, end}

The nodes in the tree are:
{1, 2, procedure, int keys [10];int low = 0;int high = 9;int middle = 0;int x = 5;}
{2, 3, 8, while, low < high}
{3, 4, procedure, middle = low + high / 2;}
{4, 5, 6, if, x > keys [middle]}
{5, 7, then, then low = middle + 1;}
{6, 7, else, else high = middle;}
{7, 2, endif, endif}
{8, 9, endwhile, endwhile}
{9, 10, procedure, keys [low] = x;}
{10, end, end}

Cyclomatic Complexity: 3

Test Paths:
1 2 3 4 5 7 2 8 9 10
1 2 3 4 6 7 2 8 9 10
1 2 8 9 10

keys  185  24  903  742  354  790  973  537  547  263
low   0
high   9
middle   0
x   5

low < high
x > keys [middle]
```

keys 276 307 358 800 103 821 412 138 761 180
low 6
high 8
middle 9
x 58

Path 1 2 3 4 6 7 2 8 9 10 covered.

keys 928 668 433 403 602 358 180 648 999 924
low 57
high 103
middle 54
x 70

low<high

keys 962 613 40 877 896 435 148 461 31 999
low 7
high 36
middle 17
x 103
low<high

keys 414 907 437 786 528 852 640 779 811 835
low 34
high 8
middle 38
x 69
low<high

Path 1 2 8 9 10 covered.

keys 562 834 370 649 679 383 825 428 973 68
low 8
high 9
middle 9
x 69

Path 1 2 3 4 5 7 2 8 9 10 covered.