Test Sequence Generation and Optimization

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering

In

Software Engineering

Thapar University, Patiala

By:

Dharmender Kamat

(80631003)

Under the supervision of

Mr. Rajesh Kumar Bhatia

Assistant Professor

JUNE 2008

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

THAPAR UNIVERSITY

PATIALA – 147004

Chapter 1
Introduction

Almost 50%-70% of the software projects fail due to the improper or inefficient testing. Almost 50% of the software production development cost is expended in software testing. It uses resources and in returns it adds nothing to the product in terms of its functionality. Software testing remains the primary concern and technique used to gain consumer's confidence in the software. Therefore, much effort has been spent in the development of software testing tools in order to significantly reduce the cost of developing software. A test path generator is a tool, which supports and helps the program tester to produce test data for software.

Software is like other physical processes where inputs are received and outputs are produced. Where software differs is in the manner in which it fails. Most physical systems fail in a fixed (and reasonably small) set of ways. By contrast, software can fail in many bizarre ways. Detecting all of the different failure modes for software is generally infeasible [39].

Unlike most physical systems, most of the defects in software are design errors, not manufacturing defects. Software does not suffer from corrosion, wear-and-tear generally it will not change until upgrades, or until obsolescence. So once the software is shipped, the design defects or bugs will be buried in and remain latent until activation.

Software bugs will almost always exist in any software module with moderate size, not because programmers are careless or irresponsible, but because the complexity of software is generally intractable and humans have only limited ability to manage complexity. It is also true that for any complex systems, design defects can never be completely ruled out.

Ideally, testing software guarantees the absence of errors in the software, but in reality it only discloses the presence of software errors but never guarantees their absence. Even, systematic testing cannot prove absolutely the absence of errors, which are detected by discovering their effect. One objective of software testing is to find errors and program structure faults. However, a problem might be to decide when to stop testing the software, e.g. if no errors are found or, how long does one keep

looking, if several errors are found.

Test data that are good for one program are not necessarily appropriate for another program even if they have the same functionality. Therefore, an adaptive testing tool for the software under test is necessary. Adaptive means that it monitors the effectiveness of the test data to the environment in order to produce new solutions with the attempt to maximize the test effectiveness. In the proposed work optimized test sequence is generated for object oriented programs.

1.1 Class in Object Oriented Paradigm

In object-oriented programming, a class is a programming language construct that is used as a blueprint to create objects. This blueprint includes attributes and methods that the created objects all share. More technically, a class is a cohesive package that consists of a particular kind of metadata. It describes the rules by which objects behave; these objects are referred to as instances of that class. A class has both an interface and a structure. The interface describes how the class and its instances can be interacted with via methods, while the structure describes how the data is partitioned into attributes within an instance. A class is the most specific type of an object in relation to a specific layer. Programming languages that support classes all subtly differ in their support for various class-related features. Most support various forms of class inheritance. Many languages also support features providing encapsulation, such as access specifiers [31].

1.1.1 Reasons for using classes

- Classes, when used properly, can accelerate development by reducing redundant code entry, testing and bug fixing. If a class has been thoroughly tested it will reduce, if not eliminate, the possibility of bugs propagating into the code.
- Another reason for using classes is to simplify the relationships of interrelated data. For example rather than writing code to repeatedly call a GUI window drawing subroutine on the terminal screen (as would be typical for structured programming), it is more intuitive to represent the window as an object and tell it to draw itself as necessary. With classes, GUI items that are similar to windows (such as dialog boxes) can simply inherit most of their functionality

and data structures from the window class. The programmer then need only add code to the dialog class that is unique to its operation.

- GUIs are a very common and useful application of classes, and GUI programming is generally much easier with a good class framework.

1.1.2 Structure of a class

Class defines the abstract characteristics of a thing or object, including the thing's characteristics (its attributes, fields or properties) and the thing's behaviors (the things it can do, or methods, operations or features). It contains the full details of its attributes and methods used. For example class person may contain person name and person id as its attributes and getname () and getid () as its methods

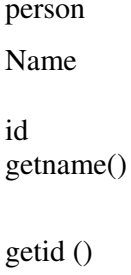


Figure 1.1 Class structure

- **Attributes**

  In object-oriented programming attributes are the different kind data used by the class to represent its properties. For example in class person name and id are its attributes which can be string and integer data types [32].

- **Method**

  In object-oriented programming, the term method refers to a subroutine that is exclusively associated either with a class called class methods, static methods, or factory methods or with an object called instance methods. For example in person class getname () and getid () are the methods [37].

1.1.3 Categories of classes

- **Concrete classes**

  A concrete class is a class that can be instantiated.

- **Abstract classes**

  An abstract class, or abstract base class (ABC), is a class that cannot be instantiated. Such a class is only meaningful if the language supports inheritance. An abstract class is designed *only* as a parent class from which child classes may be derived. Abstract classes are often used to represent abstract concepts or entities. The incomplete features of the abstract class are then shared by a group of subclasses which add different variations of the missing pieces .

  Abstract classes are super classes which contain abstract methods and are defined such that concrete subclasses are to extend them by implementing the methods. The behaviors defined by such a class are "generic" and much of the class will be undefined and unimplemented. Before a class derived from an abstract class can become concrete, i.e. a class that can be instantiated, it must implement particular methods for all the abstract methods of its parent classes.

  Most object oriented programming languages allow the programmer to specify which classes are considered abstract and will not allow these to be instantiated. For example, in Java, the keyword *abstract* is used. In C++, an abstract class is a class having at least one abstract method a pure virtual function in C++ parlance.

- **Sealed classes**

  A sealed class cannot be used as a base class. For this reason, it cannot also be an abstract class. Sealed classes are primarily used to prevent derivation. They add another level of strictness during compile-time, improve memory usage, and trigger certain optimizations that improve run-time efficiency.

- **Local and inner classes**

  In some languages, classes can be declared in scopes other than the global scope. There are various types of such classes.

  An inner class or nested class is a class defined within another class. Since it involves two classes, this can also be treated as another type of class association. The methods of an inner class can access static methods of the enclosing classes. An inner class is typically not associated with instances of the enclosing class, i.e. an inner class is not instantiated along with its enclosing class. Depending on language, it may or may not be possible to refer to the class from outside the enclosing class. A related concept is inner types (inner data type, nested type), which is a generalization of the concept of inner classes. C++ is an example of a language that supports both inner classes and inner types via typedef declarations.

  A local class, which is a class defined within a procedure or function. This limits references to the class name to within the scope where the class is declared. Depending on the semantic rules of the language, there may be additional restrictions on local classes compared non-local ones. One common restriction is to disallow local class methods to access local variables of the enclosing function. For example, in C++, a local class may refer to static variables declared within its enclosing function, but may not access the function's automatic variables.

- **Metaclasses**

  Metaclasses are classes whose instances are classes. A metaclass describes a common structure of a collection of classes. A metaclass can implement a design pattern or describe a shorthand for particular kinds of classes. Metaclasses are often used to describe frameworks .

  In some languages such as Python, Ruby, Java, and Smalltalk, a class is also an object; thus each class is an instance of the unique metaclass, which is built in the language. For example, in Objective-C, each object and class is an

instance of NSObject. The Common Lisp Object System (CLOS) provides metaobject protocols (MOPs) to implement those classes and metaclasses.

- **Partial classes**

  Partial classes are classes that can be split over multiple definitions (typically over multiple files), making it easier to deal with large quantities of code. At compile time the partial classes are grouped together, thus logically make no difference to the output. An example of the use of partial classes may be the separation of user interface logic and processing logic. A primary benefit of partial classes is allowing different programmers to work on different parts of the same class at the same time. They also make automatically generated code easier to interpret, as it is separated from other code into a partial class.

  Partial classes have been around in Smalltalk under the name of *Class Extensions* for considerable time. With the arrival of the .NET framework 2, Microsoft introduced partial classes, supported in both C# 2.0 and Visual Basic 2005.

1.1.4 Associations between classes

In object-oriented design and in UML, an association between two classes is a type of a link between the corresponding objects. A (two-way) association between classes A and B describes a relationship between each object of class A and some objects of class B, and vice versa.

- **Composition** between class A and class B describes a "part-of" relationship where instances of class B have shorter lifetime than the lifetime of the corresponding instances of the enclosing class. Class B is said to be a part of class A. This is often implemented in programming languages by allocating the data storage of instances of class A to contain a representation of instances of class B.
- **Aggregation** is a variation of composition that describes that instances of a class are part of instances of the other class, but the constraint on lifetime of the instances is not required. The implementation of aggregation is often via a pointer or reference to the contained instance. In both cases, method

implementations of the enclosing class can invoke methods of the part class. A common example of aggregation is a list class. When a list's lifetime is over, it does not necessarily mean the lifetimes of the objects within the list are also over.

- **Inheritance**

  Inheritance is a class association which involves subclasses and super classes, also known respectively as *child classes* (or *derived classes*) and *parent classes* (or *base classes*). For example if [car] was a class, then [station wagon] and [mini-van] might be two subclasses. If [Button] is a subclass of [Control], then all buttons are controls. Subclasses usually consist of several kinds of modifications (customizations) to their respective superclasses: addition of new instance variables, addition of new methods and overriding of existing methods to support the new instance variables.

  Some programming languages (for example C++) allow multiple inheritance - they allow a child class to have more than one parent class. This technique has been criticized by some for its unnecessary complexity and being difficult to implement efficiently, though some projects have certainly benefited from its use. Java, for example has no multiple inheritance, as its designers felt that it would add unnecessary complexity. Java instead allows inheriting from multiple pure abstract classes (called interfaces in Java).

  Sub- and superclasses are considered to exist within a hierarchy defined by the inheritance relationship. If multiple inheritance is allowed, this hierarchy is a directed acyclic graph (or DAG for short), otherwise it is a tree. The hierarchy has classes as nodes and inheritance relationships as links. The levels of this hierarchy are called layers or levels of abstraction. Classes in the same level are more likely to be associated than classes in different levels.

1.2 Cyclomatic complexity

Cyclomatic complexity is a software metric (measurement). It was developed by Thomas McCabe and is used to measure the complexity of a program. It directly measures the number of linearly independent paths through a program's source code.

Cyclomatic complexity is computed using a graph that describes the control flow of the program. The nodes of the graph correspond to the commands of a program. A directed edge connects two nodes if the second command might be executed immediately after the first command [35, 34].

- **Definition**

  $M = E - N + P$

  where

  M = cyclomatic complexity
  E = the number of edges of the graph
  N = the number of nodes of the graph
  P = the number of connected components.

  "M" is alternatively defined to be one larger than the number of decision points (if/case-statements, while-statements, etc) in a module (function, procedure, chart node, etc.), or more generally a system.

- **Alternative definition**

  $v(G) = e - n + p$
  G is a program's flow graph
  e is the number of edges (arcs) in the flow graph
  n is the number of nodes in the flow graph
  p is the number of connected components

- **Alternative way**

  There is another simple way to determine the cyclomatic number. This is done by counting the number of closed loops in the flow graph, and incrementing the number by one.

  i.e. M = Number of closed loops + 1

  where M = Cyclomatic number.

- **Implications for Software Testing**

  (1) M is a lower bound for the number of possible paths through the control flow graph.

  (2) M is an upper bound for the number of test cases that are necessary to achieve complete branch coverage.

A module with a high complexity number requires more testing effort than a module with a lower value since the higher complexity number indicates more pathways through the code. This also implies that a module with higher complexity is more difficult for a programmer to understand since the programmer must understand the different pathways and the results of those pathways.

1.3 A Brief Review of ACO Algorithm

Ant Colony Optimization (ACO) algorithm is a paradigm for designing meta-heuristic algorithms for combinatorial optimization problems. The first algorithm which can be classified within this framework was presented in 1991 [2, 7] and, since then, many diverse variants of the basic principle have been reported in the literature. The essential trait of ACO algorithms is the combination of a priori information about the structure of a promising solution with a posteriori information about the structure of previously obtained good solutions.

Meta-heuristic algorithms are algorithms which, in order to escape from local optima, drive some basic heuristic: either a constructive heuristic starting from a null solution and adding elements to build a good complete one, or a local search heuristic starting from a complete solution and iteratively modifying some of its elements in order to achieve a better one. The meta-heuristic part permits the low level heuristic to obtain solutions better than those it could have achieved alone, even if iterated. Usually, the controlling mechanism is achieved either by constraining or by randomizing the set of local neighbor solutions to consider in local search as is the case of simulated annealing [24] or tabu search [5]), or by combining elements taken by different solutions as is the case of evolution strategies[27] and genetic [3] or bionomic [29] algorithms.

The characteristic of ACO algorithms is their explicit use of elements of previous solutions. In fact, they drive a constructive low-level solution, as Greedy randomized adaptive search procedures [26] does, but including it in a population framework and randomizing the construction in a Monte Carlo way. A Monte Carlo combination of different solution elements is suggested also by Genetic Algorithms [8], but in the case of ACO the probability distribution is explicitly defined by previously obtained solution components.

The particular way of defining components and associated probabilities is problem-specific, and can be designed in different ways, facing a trade-off between the specificity of the information used for the conditioning and the number of solution which need to be constructed before effectively biasing the probability distribution to favor the emergence of good solutions. Different applications have favored either the use of conditioning at the level of decision variables, thus requiring a huge number of iterations before getting a precise distribution, or the computational efficiency, thus using very coarse conditioning information.

1.3.1 Theory

ACO [12, 13] is a class of algorithms, whose first member, called Ant System, was initially proposed by Colorni, Dorigo and Maniezzo [11, 24, and 14]. The main underlying idea, loosely inspired by the behavior of real ants, is that of a parallel search over several constructive computational threads based on local problem data and on a dynamic memory structure containing information on the quality of previously obtained result. The collective behavior emerging from the interaction of the different search threads has proved effective in solving combinatorial optimization (CO) problems.

Following [16], the following notations are used. A combinatorial optimization problem is a problem defined over a set **C** = c*1*... c*n* of basic components. A subset **S** of components represents a solution of the problem; **F** ≤ 2^***C*** is the subset of feasible solutions, thus a solution **S** is feasible if and only if **S** € **F**. A cost function *z* is defined over the solution domain, z: 2^***C*** € **R**, the objective being to find a minimum cost feasible solution S*, i.e., to find S*: S* € **F** and z(S*) ≤ z(**S**), **S** € **F**.

Given this, the functioning of an ACO algorithm can be summarized as follows [15]. A set of computational concurrent and asynchronous agents a colony of ants moves through states of the problem corresponding to partial solutions of the problem to solve. They move by applying a stochastic local decision policy based on two parameters, called trails and attractiveness. By moving, each ant incrementally constructs a solution to the problem. When an ant completes a solution, or during the construction phase, the ant evaluates the solution and modifies the trail value on the components used in its solution. This pheromone information will direct the search of the future ants.

Furthermore, an ACO algorithm includes two more mechanisms: trail evaporation and, optionally, daemon actions. Trail evaporation decreases all trail values over time, in order to avoid unlimited accumulation of trails over some component. Daemon actions can be used to implement centralized actions which cannot be performed by single ants, such as the invocation of a local optimization procedure, or the update of global information to be used to decide whether to bias the search process from a non-local perspective.

More specifically, an ant is a simple computational agent, which iteratively constructs a solution for the instance to solve. Partial problem solutions are seen as state*s*. At the core of the ACO algorithm lies a loop, where at each iteration, each ant moves (performs a step) from one state to another one, corresponding to a more complete partial solution. That is, at each step each ant *k* computes a set of feasible expansions to its current state, and moves to one of these in probability. The probability distribution is specified as follows. For ant *k*, the probability of moving from state one to another state depends on the combination of two values:

- The attractiveness of the move, as computed by some heuristic indicating the a priori desirability of that move;
- The trail level of the move, indicating how proficient it has been in the past to make that particular move: it represents therefore an *a* posteriori indication of the desirability of that move.

Trails are updated usually when all ants have completed their solution, increasing or decreasing the level of trails corresponding to moves that were part of "good" or "bad" solutions, respectively.

1.3.2 Ant Colony System

AS was the first algorithm inspired by real ants behavior. AS was initially applied to the solution of the traveling salesman problem but was not able to compete against the state-of-the art algorithms in the field. On the other ACO algorithms has been introduced to show the potentiality of using artificial pheromone and artificial ants and to drive the search of always better solutions for complex optimization problems. The next researches were motivated by two goals: the first was to improve the performance of the algorithm and the second was to investigate and better explain its behavior. Gambardella and Dorigo proposed in 1995 the Ant-Q algorithm [16], an extension of AS which integrates some ideas from Q-learning and in 1996 Ant Colony System (ACS) [9] a Simplified version of Ant-Q which maintained approximately the same level of performance, measured by algorithm complexity and by computational results. Since ACS is the base of many algorithms defined in the following years, the attention is focused on ACS other than Ant-Q. ACS differs from the previous AS because of three main aspects:

1.3.2.1 Pheromone

In ACS once all ants have computed their tour (i.e. at the end of each iteration) AS updates the pheromone trail using all the solutions produced by the ant colony. Each edge belonging to one of the computed solutions is modified by an amount of pheromone proportional to its solution value. At the end of this phase the pheromone of the entire system evaporates and the process of construction and update is iterated. On the contrary, in ACS only the best solution computed since the beginning of the computation is used to globally update the pheromone. As was the case in AS, global updating is intended to increase the attractiveness of promising route but ACS mechanism is more effective since it avoids long convergence time by directly concentrate the search in a neighborhood of the best tour found up to the current iteration of the algorithm.

In ACS, the final evaporation phase is substituted by a local updating of the pheromone applied during the construction phase. Each time an ant moves from the current city to the next the pheromone associated to the edge is modified in the following way: $\tau_{ij}(t) = r \cdot \tau_{ij}(t-1) + (1 - r) \cdot t\ 0$ where $0 \leq \rho \leq 1$ is a parameter (usually

set at 0.9) and τ_0 is the initial pheromone value. τ_0 is defined as $\tau_0 = (n \cdot L_{nn})^{-1}$, where L_{nn} is the tour length produced by the execution of one ACS iteration without the pheromone component. The effect of local-updating is to make the desirability of edges change dynamically: every time an ant uses an edge this becomes slightly less desirable and only for the edges which never belonged to a global best tour the pheromone remains τ_0. An interesting property of these local and global updating mechanisms is that the pheromone $\tau_{ij}(t)$ of each edge is inferior limited by τ_0. A similar approach was proposed with the Max-Min-AS [28] that explicitly introduces lower and upper bounds to the value of the pheromone trials.

1.3.2.2 State Transition Rule

During the construction of a new solution the state transition rule is the phase where each ant decides which is the next state to move to. In ACS a new state transition rule called pseudo-random-proportional is introduced. The pseudorandom-proportional rule is a compromise between the pseudo-random state choice rule typically used in Q-learning and the random-proportional action choice rule typically used in Ant System. With the pseudo-random rule the chosen state is the best with probability q (exploitation) while a random state is chosen with probability $1-q$ (exploration). Using the AS random-proportional rule the next state is chosen randomly with a probability distribution. The ACS pseudo-random-proportional state transition rule provides a direct way to balance between exploration of new states and exploitation of a priori and accumulated knowledge.

The best state is chosen with probability q_0 (that is a parameter $0 \leq q_0 \leq 1$ usually fixed to 0.9) and with probability $(1-q_0)$ the next state is chosen randomly with a probability distribution based on η_{ij} and τ_{ij} weighted by α (usually equal to 1) and β (usually equal to 2) .

1.3.2.3 Hybridization and performance improvement

ACS was applied to the solution of big symmetric and asymmetric traveling salesman problems. For these purpose ACS incorporates an advanced data structure known as candidate list. A candidate list is a static data structure of length cl which contains, for a given city i, the cl preferred cities to be visited. An ant in ACS first uses candidate list with the state transition rules to choose the city to move to. If none of the cities in

the candidate list can be visited the ant chooses the nearest available city only using the heuristic value hij.

ACS for TSP/ATSP has been improved by incorporating local optimization heuristic (hybridization): the idea is that each time a solution is generated by the ant it is taken to its local minimum by the application of a local optimization heuristic based on an edge exchange strategy, like 2-opt, 3-opt or Lin-Kernighan [25]. The new optimized solutions are considered as the final solutions produced in the current iteration by ants and are used to globally update the pheromone trails.

This ACS implementation combining a new pheromone management policy, a new state transition strategy and local search procedures was finally competitive with state-of-the-art algorithm for the solution of TSP/ATSP problems. This opened a new frontier for ACO based algorithm. Following the same approach that combines a constructive phase driven by the pheromone and a local search phase that optimizes the computed solution, ACO algorithms were able to break several optimization records, including those for routing and scheduling problems

1.4 A Brief Review of Software Testing

Testing [19, 20, and 21] is the most critical phase in the software development life cycle. The testing phase is the final filter for all errors of omission and commission. Testing software is far more complex than exercising a program to see if it works. Each review, inspection, audit, walk-through, group code read, all is in reality a form of test. The more effective that can make early is static testing, the fewer problems will encounter in the dynamic stages of testing. IT has shown again and again that the earlier a fault can be detected and removed, the lower the additional development cost associated with removing the error. Preparation for testing should begin as soon as each software product is defined.

The increasing visibility of software as the system element and the attendant “costs” associated with a software failure are motivating forces for well-planned, thorough testing. It is not unusual for a software development organization to expend between 30 and 40 percent of total project effort on testing. In the extreme, testing of human-rated software can cost three to five times as much as all other software engineering activities combined.

1.4.1 Objective

The main objective of testing is to prove that the software product as a minimum meets a set of pre-established acceptance criteria under a prescribed set of environmental circumstances. There are two components to this objective. The first component is to prove that the requirements specification from which the software was designed is correct. The second component is to prove that the design and coding correctly respond to the requirements [1]. Correctness means that function, performance, and timing requirements match acceptance criteria.

Software testing is further complicated by the fact that system acceptance criteria usually involve hardware, procedures, and operators so that acceptance tests involve more than just the software. Software tests are designed to force failures. In that regard, software testing is intrinsically destructive.
Following are the objectives that software testing follows:

- Testing is a process of executing a program with the intent of finding an error.

- A good test case is one that has a high probability of finding an as-yet undiscovered error.

A successful test is one that uncovers an as-yet undiscovered error.
Our objective is to design tests that systematically uncover different classes of errors and do so with a minimum amount of time and effort. If testing is conducted successfully it will uncover errors in the software. As a secondary benefit, testing demonstrates that software functions appear to be working according to the specifications and that performance requirements appear to have been met. In addition, data collected as testing is conducted provides a good indication of software quality as a whole. But there is one thing that testing cannot do:
Testing cannot show the absence of defects, it can only show that software errors are present**.**

1.4.2 Testing Principles

Following are principles of Software Testing [23]:

- All tests should be traceable to customer requirements. The objective of system testing is to uncover errors. It follows that the most severe defects are those that cause the program to fail to meet its requirements.

- Tests should be planned long before testing begins. Test planning can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.

- Testing should begin “in the small” and progress toward testing “in the large”.

The first test planned and executed generally focus on individual program modules. As testing progresses, testing shifts focus in an attempt to find errors in integrated clusters of modules and ultimately in the entire system.

- Exhaustive testing is not possible. The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing.

- To be most effective, testing should be conducted by an independent third party.

By “most effective “, means testing that has the highest probability of finding errors. For this reason, the software engineer who created the system is not the best person to conduct all tests for the software.

1.4.3 Characteristics of a "Good" Test

Following are the characteristics of a good test:

- A good test has a high probability of finding an error. To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail.

- A good test is not redundant. Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose.

- A good test should be "best of breed". In a group of tests that have a similar intent, time and resource limitations may militate for the execution of only a subset of these tests. In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.

- A good test should be neither too simple nor too complex. Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

1.4.4 Testing Process

The IEEE829 standard [1] describes a framework within which the entire testing process can be managed. The framework allows easy communication between members of a testing project, organizes the testing process, and outlines the documents that should be made part of any compliant testing process.

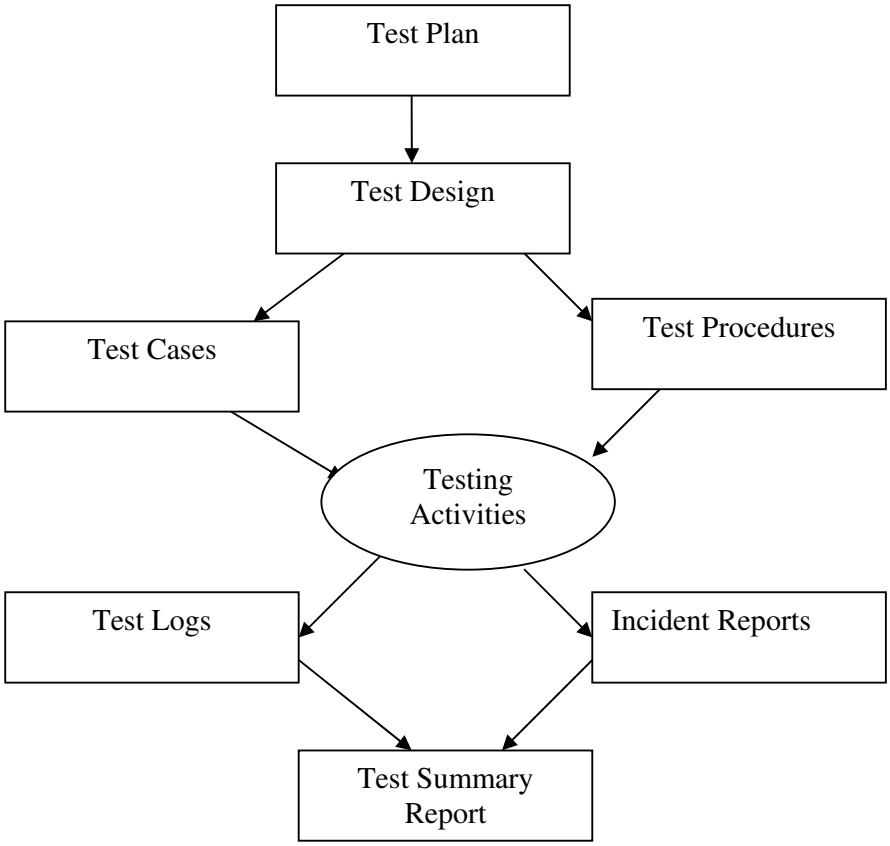


Figure 1.2: Testing activities

1.4.4.1 Test Plan

Test plan describes the scope, approach, resources, and schedule of testing activities in a given project, and identifies the items to be tested, the features of those items to be tested, the individual testing tasks that are to be performed, and personnel responsible for those tasks, along with the risks associated with the plan.

Test plan should have the following structure:

- Test plan identifier
- Introduction
- Items to be tested
- Features to be tested
- Features not to be tested
- Testing approach
- Test item pass/fail criteria

- Test suspension criteria
- Test resumption requirements
- Testing tasks
- Environmental needs
- Responsibilities
- Staffing and training requirements
- Schedule
- Risks and contingencies
- Approvals

1.4.4.2 Test Design

Test design, or test specification as it is sometimes known, further refines the testing approach, and identifies the test cases and test procedures, and test item pass/fail criteria.

Test design specification should have the following structure:

- Test design identifier
- Features to be tested
- Approach refinements
- Test identification
- Pass/fail criteria

1.4.4.3 Test Cases

Individual test cases document the values that will be used as input to individual tests, together with the associated expected outputs. A test case identifies any constraints on the test procedure resulting from the use of the test case, and separated from the test design to allow easy reuse in other situations.

Test cases should have the following structure:

- Test case identifier
- Items to be tested
- Input specifications
- Expected output specifications
- Environmental prerequisites
- Special procedural requirements

- Inter-case dependencies

1.4.4.4 Test Procedures

Test procedure describes the exact steps required to operate the system and execute test cases in order to implement the test design. Test procedure is kept separate from the test design as it is followed step by step, and does not contain irrelevant detail.

Test procedure should have the following structure:

- Test procedure identifier
- Purpose
- Special requirements
- Procedure steps

1.4.4.5 Test Logs

Test logs are used to record what occurred during execution of a test or set of tests. Test logs may either be manually created as tests are executed, or automatically by the system as testing processes.

Test logs should have the following structure:

- Test log identifier
- Description
- Activity and event entries

1.4.4.6 Incident Reports

Incident reports are used to provide a description of any events that occur during testing that require further investigation.

Incident reports should have the following structure:

- Incident report identifier
- Summary
- Incident description
- Impact of the incident

1.4.4.7 Test Summary Report

Test summary report summarizes the testing activities associated with one or more test designs.

Test summary report should have the following structure:

- Test summary report identifier
- Summary
- Variances
- Comprehensive assessment
- Summary of results
- Evaluation
- Summary of activities
- Approvals

1.4.5 Software Testing Approaches

1.4.5.1 Top-Down Approach

The top-down approach is based on establishing the top-level control structure first. In top-down strategy, testing starts from the top of the hierarchy, and then incrementally adds modules that it calls and tests the new combined system. This approach of testing requires stubs to be written. A stub is a dummy routine that simulates a module [23]. In the top-down approach, a module or a collection cannot be tested in isolation because they invoke some other modules. To allow the modules to be tested before their subordinates have been coded, stubs simulate the behavior of the subordinates.

1.4.5.2 Bottom-up Approach

The bottom-up approach starts from the bottom of the hierarchy. First the modules at the very bottom, which have no subordinates, are tested. Then these modules are combined with higher-level modules for testing. At any stage all the subordinate modules exist and have been tested earlier. To perform bottom-up testing, drivers are needed to set up the appropriate environment and invoke the module. It is the job of the driver to invoke the module under testing with the different set of test cases.

1.4.6 Software Testing Techniques

1.4.6.1 Static Testing

The term static testing refers to testing the software requirement specification (SRS), software design specification (SDS) and other non-executable items through requirement analysis, audits, desk checks, inspections, walk-through, etc [1]. Static

testing is employed to verify the correctness of requirements, designs, and code before execution of test cases. Static testing will also audit newly developed or reused code for adherence to established standards. A successful static test of a software module depends upon several things going right:

- A correct allocation of requirements to the software components.
- A correct partitioning and sub allocation of software requirements to the module.
- Successful (correct) module design.
- A successful translation of the intermediate code (pseudo-code, POL, etc.) into programming language statements. However, true representative test cases must be successfully executed before the testing and integration team certifies a software module. Usually this step is not a part of static testing.

The purposes of the static testing are:

- Validating the requirement specifications.
- Looking for omissions, inconsistencies, redundancies in all documents and source code.
- To ensure that the documents of design and coding conform to the specification.

1.4.6.2 Dynamic Testing

Dynamic testing is a term that describes the development of test cases and test procedures, the execution of test cases, and the structure and use of test logs and anomaly or incident reports. There are two popular ways to perform dynamic testing, namely, black box testing and white (glass) box testing. Either of these two methods requires a set of well-developed and well-structured test cases.

Dynamic testing cannot prove the absolute correctness of a software product unless it is performed in an exhaustive manner. An exhaustive test requires a set of test cases that guarantees the following: explicitly exercises every possible, module path and every possible combination of paths with every possible module input and every possible combination of module inputs

1.4.7 Types of Testing

1.4.7.1 Black Box or Functional Testing

Black box testing is the testing of a piece of software without regard to its underlying implementation. Specifically, it dictates that test cases for a piece of software are to be generated based solely on an examination of the specification (external description) for that piece of software. The goal of black box testing is to demonstrate that the software being tested does not adhere to its external specification.

The objective is to search for interface errors, function or process errors, performance shortcomings, start-up/shutdown errors, and errors in local (module) databases by selecting appropriate inputs and external conditions and monitoring outputs.

Black box testing attempts to find errors in the following categories:

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external database access
- Performance errors
- Initialization and termination errors
- Black Box Testing Techniques includes:
- Equivalence Partitioning
- Boundary Value Analysis
- Comparison Testing

1.4.7.2 White Box Testing

White box testing also known as glass box testing .It is a test case design method that uses the control structure of the procedural design to derive test cases. Using white box testing methods, the software engineer can derive test cases that

- Guarantee that all independent paths within a module have been exercised at least once.
- Exercise all logical decisions on their true and false sides.
- Execute all loops at their boundaries and within their operational bounds.
- Exercise internal data structures to assure their validity.

Thus white box testing is the testing of the underlying implementation of a piece of software e.g., source code without regard to the specification or external description for that piece of software. The goal of white box testing of source code is to identify

such items as unintentional infinite loops, paths through the code which should be allowed, but which cannot be executed and dead or unreachable code.

White Box Testing Techniques includes:

Control flow testing: Testing on the basis of the flow of control of a program. It is of the following types:

- Statement*:* each statement executed at least once
- Branch*:* each branch traversed and every entry point taken at least once
- Condition*:* each condition True at least once and False at least once.
- Branch/Condition*:* both Branch and Condition coverage achieved.
- Multiple Conditions*:* Multiple Conditions coverage technique states that test cases must be written such that all possible combinations of conditions in each decision are taken at least once.
- Loop*:* Loop coverage technique states that test cases must be written to test the loop counters.

Data flow testing: Testing on the basis of the flow of control of a program. It is of the following types:

- All Definition-Uses*:* It requires that every definition of every variable to every use of that definition be exercised under the test.
- All Uses*:* In this test set include at least one path segment from every definition of every variable to every use of that definition
- All p-uses/some c-uses*:* In All p-uses/some c-uses, test set include at least one path segment from every definition of every variable to every predicate use of that definition; if there are definitions of variables that are not covered by the above prescription then add computational use test cases are required to cover every definition.
- All c-uses/ some p-uses*:* In All c-uses/some p-uses, test set include at least one path segment from every definition of every variable to every computational use of that definition; if there are definitions of variables that are not covered by the above prescription then add predicate use test cases are required to cover every definition.
- All definitions*:* In this, test set includes every definition of every variable be covered by at least one use of that variable, be that use the computational use or predicate use.

- All p-uses*:* In All p-uses, test set include at least one path segment from every definition of every variable to every predicate use of that definition; if there are definitions of variables that are not covered by the above prescription then leave them.
- All c-uses*:* In All c-uses, test set include at least one path segment from every definition of every variable to every computational use of that definition; if there are definitions of variables that are not covered by the above prescription then, leave them.

1.5 Honeypot class

Honeypot classes are those classes which has high number of class dependent on them. In other words these are the most vulnerable classes which need to be tested first as any change to them will affect other classes as well. In any test sequence number of honeypot classes determines its priority to test.

1.6 Motivation & Objective

Properly generated test suites may not only locate the defects in software systems, but also help in reducing the high cost associated with software testing. It is often desired that test sequences in a test suite can be automatically generated to achieve required test coverage. However, automatic test sequence generation remains a major problem in software testing. As testing is crucial for the development of feasible software, it becomes necessary to explore the new techniques of the testing.

The main ideas of the research are:

- Search of new technique for test sequence generation.
- Generation of optimized test sequences using Ant Colony Optimization (ACO) Algorithm and basic features of class

1.7 Organization of Thesis

The First chapter briefly introduces the motivation behind development of the project and the organization of the whole thesis. It also gives the brief introduction of technologies used in the proposed methodology.

The Second chapter reviews briefly the existing work done in test sequence generation The Third Chapter gives the detailed description of the proposed system and discusses the problem formulation

Experimental results are demonstrated in fourth chapter using a case study

The Fifth Chapter gives the conclusion and the future Scope of the thesis

Chapter 2
Literature Review

Software testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. Since it is crucial to software quality and widely deployed by programmers and testers, software testing still remains an art, due to limited understanding of the principles of software [38]

The difficulty in software testing comes from the complexity of software: we can not completely test a program with moderate complexity. Testing is more than just debugging. The purpose of testing can be quality assurance, verification and validation, or reliability estimation. Testing can be used as a generic metric as well. Correctness testing and reliability testing are two major areas of testing. Software testing is a trade-off between budget, time and quality.

Regardless of the limitations, testing is an integral part in software development. It is broadly deployed in every phase in the software development cycle. Typically, more than 50% percent of the development time is spent in testing.

2.1 State Based Test Sequence Generation using ACO

The proposed methodology uses UML Statechart diagrams and ACO to generate test sequences for state-based software testing. The advantages of the proposed approach are that the UML Statechart diagrams exported by UML tools can be directly used to generate test sequences, and the automatically generated test sequences are always feasible, nonredundant and achieve the required test adequacy criterion.

2.1.1 An ACO Approach

State-based testing is frequently used in software testing. There are two major problems commonly associated with state-based software testing:

- Some of the generated test cases are infeasible;
- Inevitably many redundant test cases have to be generated in order to achieve the proper testing coverage required by test adequacy criteria.

For the first problem, approaches using code execution or model execution techniques have been developed to exclude the infeasible paths. However, to the best knowledge of author, no systematic strategy has been reported to successfully deal with both problems before this paper

The UML Statechart diagrams have been extensively used in state-based software testing. In order to define test adequacy criteria for state-based software testing using the UML Statechart diagrams, the Statechart diagrams have to be flattened to remove all hierarchy and concurrency. It has to be emphasized that the Statechart flattening process is merely used for testing purpose, a flattened Statechart diagram is not necessary a semantic equivalence to the original Statechart diagram

The proposed approach addresses the automatic generation of test sequences from the UML Statechart diagrams for state-based software testing. The all states test coverage is used as test adequacy requirement. Specifically, two requirements have been imposed that the generated test suite has to satisfy:

- All-state coverage
- Feasibility – Each test sequence in the test suite represents a feasible path in the corresponding Statechart diagram

A directed graph is defined as G = (V, E) where V is a set of vertices of the graph and E a set of edges of the graph. A flattened UML Statechart can be viewed as a directed

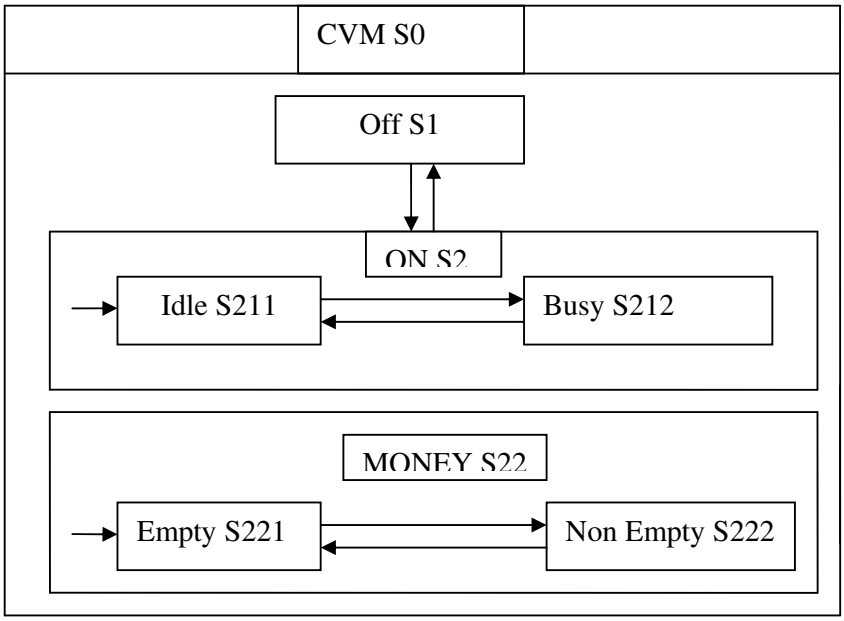


Fig 2.1: Coffee Vendor Machine (CVM)

graph where the vertices are the states of the Statechart diagram, and the edges are the transitions between the states. They have developed a tool to automatically convert a Statechart diagram to a directed graph

The converted graph is shown in figure below.

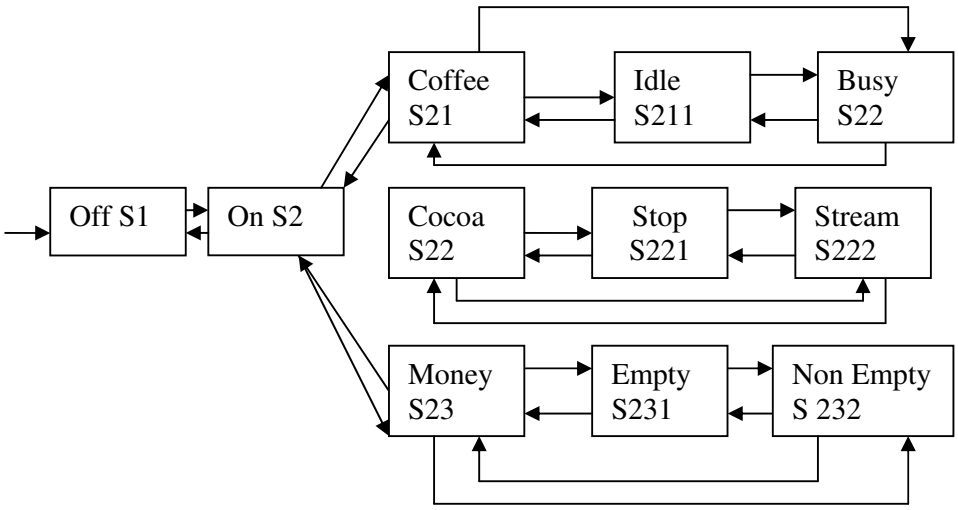


Fig 2.2: Converted Graph (CVM)

The converted graphs are directed, dynamic graphs in which the edges (transitions in Statechart sense) may dynamically appear or disappear based on the evaluation of their guards. Therefore, it is required to consider the problem of sending a group of ants to cooperatively search a directed graph G. It has been observed that the original ACO algorithms in [18], [19] are difficult to be applied to this type of directed and dynamic graphs to generate test data for the corresponding testing problems. An alternative algorithm has to be proposed in order to use ants to search the graphs for test sequence generation.

2.1.1.1 Algorithm

1. Evaluation at vertex α
    - Update the Track - Push the current vertex α into the track set S
    - Evaluate Connections - Evaluate all connections to the current vertex α to determine T. The procedure involves evaluation of all possible transitions from the current states α to other neighboring states, using the state-transition table associated with the UML Statechart diagram

- Sense the Trace - For the non-negative connections in T, the ant senses and gathers the corresponding pheromone levels P at the other ends of the connections

2. Move to next vertex

- Select Destination - The following prioritized rules are used in ant's selection:
  i) Select the vertex Vi with the lowest pheromone level P(Vi) sensed from the current vertex α
  ii)If vertices Vi and Vj shares the same lowest pheromone level P(Vi) P(Vj), but T(Vi) = 0 and T(Vj) = 1, select Vi
  iii)If vertices Vi and Vj shares the same lowest pheromone level P(Vi) = P(Vj) and T(Vi) = T(Vj), randomly select one vertex

Destination β is the vertex selected using the above rules

- Update Pheromone - Update the pheromone level for the current vertex α to

  P (α) = max (P (α), P (β) +1) if T (β) = 1 or

  P (α) = max (P (α), P (β) +1) +TP if T (β) = 0

Where TP is a high pheromone level which decays in one iteration of the steps, namely, TP quickly decays to 0 before ant's next move at the end of Step 2

Move - Move to the destination vertex β, set α: = β, and return to Step 1.

Using the above algorithms the path which are traced are:

Test Sequence 1 = {S1, S2, S21, S211, S21, S2, S22, S221, S222, S22, S2, S21, S211, S212}

Test Sequence 2 = {S1, S2, S2, S22, S221, S222, S22, S2, S21, S211, S212}

Test Sequence 2 is the shortest path for all-states coverage requirement using single ant sequentially. Hence by deploying the ant sequentially we can find the different paths covering all states of the system.

Using the developed algorithm, a group of ants can effectively explore the UML Statechart diagrams and automatically generate test sequences to achieve the test coverage requirement.

2.2 Automatic Mutation Test Input Data Generation

Fault-based testing is often used to overcome limitations of other testing approaches; however it is also recognized as being expensive. On the other hand, evolutionary

algorithms have been proved suitable for reducing the cost of data generation in the context of coverage based testing.

Hence new evolutionary approach based on ant colony optimization for automatic test input data generation in the context of mutation testing is proposed. In new approach the ant colony optimization algorithm is enhanced by a probability density estimation technique.

2.2.1 Mutation Testing

Fault-based testing techniques such as mutation analysis and mutation testing are often advocated to overcome limitations of other testing approaches. Mutation analysis identifies techniques to mutate, i.e., to modify, software artifacts, while mutation testing tests adequacy criteria based on mutation analysis.

In mutation testing the typical testing situation is somehow reversed. It is assumed that a test suite is available together with mutated copies of the original program. A good test case is one that *kills* one or more mutants, for which mutant outputs are different from those of the original program. In this framework, a set of test cases is considered adequate if it distinguishes the original program from all its mutants.

In a real world situation, simple faults are injected into the original program to obtain faulty versions of the program, the mutants. Then test input data are produced to attain the highest possible mutation score, i.e., to kill the highest number of mutants. A set of test cases is more adequate than another if it kills a larger number of mutants. On the other hand, a test suite is preferred over others if it contains fewer test cases and is closer to the adequacy criterion, i.e., has the highest mutation score. Intuitively, mutation testing promotes high quality test suites and has high potential for automation

The main idea which is covered in this discussion is:

- A new evolutionary approach for automatic test input data generations in the context of mutation testing, which naturally reduce the computational cost in such a test strategy.
- Exploitation of a new emergent search technique, ACO, to facilitate input data generation and compare results with Hill Climbing (HC), Genetic Algorithm (GA) and random search (RND) on two programs;

- Incorporation of new ideas based on a probability density estimation process to automatically refine and guide the search in promising search regions;
- A customization of ACO to the problem of generating input test data to kill mutants.

2.2.2 Problem Formulation

Let Pg be a program under test and I = (x1, x2. . . xk) be the vector of its input variables. Each input variable xi takes its values in a domain Di, i = 1. . . K, thus, the domain of the program Pg, without any further knowledge, is the cross product D = D1 × D2 . . . ×Dk. Further assume that R is a set of mutation operators; each mutation operator is a representative of a typical programming error and it produces a single modification in a single program point giving rise to a mutated version of Pg.

By applying mutation operator r ЄR to Pg, N mutated copies M1, M2…. MN of Pg are obtained. In other words, Mi = ri (Pg) with ri 2 R, ri the ith, i = 1, . . . , N, a selected mutation operator that mutates Pg by injecting a simple fault at a statement sm, called the Pg *mutated statement*.

The problem of test data generation in the context of mutation testing consists of finding a set of test input values that maximizes the number of killed mutants. The essential problem is to find assignments of values to input variables (x1, x2, . . . , xk), called *test cases*, such that when the test suite is executed over the set of mutantsM1, M2, . . . ,MN it kills the highest possible number of mutants.

Each mutant Mj, j = 1, . . . , N, is killed if the three conditions are satisfied.

- The first condition (the reachability condition) states that mutated statement sm in the mutant Mj must be reached.
- The second condition requires the value of the mutated expression, once executed, in the statement sm to differ from its value before mutation. In other words, at the mutated statement sm, the state of the mutant is different from the original program's one.
- The third condition (the sufficiency condition) requires the mutated value, i.e., the mutated state at sm, to propagate to the mutant output. In this paper, we will collectively refer to these conditions as *the killing conditions*.

If an input test case t *kills* a mutant Mj this latter is said to be killed or killed by t; otherwise Mj is said to be still alive. Therefore, if T is the set of test cases killing d mutants, the adequacy of T is assessed by its mutation score MScore (T) given by the following formula:

MScore (T) = 100 (d /N − eq)

where eq is the number of equivalent mutants i.e., mutants that cannot be distinguished from Pg. Input variables x1, x2, . . . , xk taking values in D1 × D2 . . . × Dk are assumed to be either integer or real values.

2.2.3 ACO to Generate Test Cases

ACO has been customized according to the mutation testing problem. Ants' foraging behavior is exploited to generate test input data, test cases for killing as many mutants as possible. Since each test case is made up by realizations of input parameters x1, x2, . . . , xk, values chosen in parameter domains, the ants task is to *select good* assignments of parameters values

ACO application can be summarized as follows.

Ants start searching for a test case that kills a given mutant by initially randomly choosing test cases. For each test case chosen by an ant, the mutant is executed and the quality of the test case is evaluated. Quality is quantified as closeness to satisfy the killing conditions. Then the ant deposits pheromone trail, i.e., marks with pheromone the values forming the test case. The quantity of deposed pheromone is proportional to the test case quality. As in nature, artificial ants tend to follow pheromone trails. This indirect communication between ants progressively promotes parameter assignments, i.e., test cases, closer to satisfying the killing conditions and eventually killing the mutant.

2.3.4 Test Input Data Case Construction Mechanism

The test case generation problem was modeled as a directed graph Gr (V, E); each vertex in V, the set of vertices, represents an input parameter xi. Parameter domains are assumed to be finite, numerable and quantized with suitable quantization steps *a* priori known; thus, domain Di, i = 1. . . k is quantized into a set of values QDi containing |QDi| values.

Nodes are considered ordered and circular. They are ordered by the relative position in the program parameter list; thus, x2 follows x1 and precedes x3. They are circular in that the node x1 follows node xk. For any given vertex xi outgoing edges (incoming to xi+1) represent all possible assignments to parameter xi from the quantized domain; thus, there are as many edges between two consecutive nodes as there are values in the quantized domain Di.

Figure 1 reports an example of such a graph traversed by ants to construct test cases.

At each iteration, all ants start their trails from the vertex representing the parameter x1, complete one tour visiting all vertices, and then return back to x1. When an ant on vertex xi moves to the next node xi+1, it chooses an edge (i, j) representing the jth value, vij |QDi|.

At the beginning, an arbitrary order is used by ants to move from one vertex to the following one. In the following iteration, the choice of the edge to be traversed depends on the amount of pheromone accumulated on that edge. The higher the amount, the higher will be the probability of choosing that edge. When all ants complete one tour, each one deposits a pheromone amount on the edges of the traversed path. Each chosen path represents one candidate test case generated in the current tour. The process is iterated, and at each tour more and more adequate test cases are constructed until a stopping criterion is met.

Chapter 3

Proposed Methodology and Problem Formulation

Testing of software remains the most challenging and laborious work in the software development. Various methodologies have been tried to test the software, but finding the 100% bugs or defects in software remains an illusive target. The core fact about the software testing is that it cannot be exhaustive as it is too tedious and time consuming. The automation of the testing is possible but it does not cover all the test cases which are required for exhaustive testing. A suitable automation technique for the optimized test sequence generation is useful as it saves the time and efforts used for testing. The optimized test sequence leads to adequate test case generation

3.1 Gaps in Existing Approaches

There are various testing techniques which are used for the test sequence generation. Some relevant works in the test sequence and test case generation were discussed in the previous chapter. None of the techniques suggests the use of classes in test sequence generation. The brief comparison of the proposed methodology with the existing work is given below.

3.1.1 Test Sequence Generation for State based Software Testing:

- This approach requires the conversion of system in states of UML State Chart Diagram and this graph is need to be converted into another graph known as converted graphs. The problem with states is that what is happening behind it is completely hidden. The proposed methodologies do not require such conversion. Instead of using states it uses the class which gives the complete picture of an entity or system
- This approach generates the test sequence which focuses the complete code coverage, while proposed methodology focuses on the most optimized test sequence based on different parameters

3.1.2 Automatic Mutation Test Input Data Generation:

- In this approach, first it is required to generate a number of Mutants and then need to find out the test cases which could kill the maximum number of

Mutants. In the proposed approach no such modification is required to the source code. It only needs to parse the source code, extract the all necessary classes used in it and finally generates the test sequence.

- The mutation testing is a complicated process, requires a lots of iteration before required test cases can be generated which could kill the maximum number of Mutants. While in the proposed iteration done only on the basis of four parameters namely dependency, cyclomatic complexity, methods and attributes. Thus requires less time to generate the test sequence.

3.2 Problem Statement

The proposed methodology works on the three important concepts software testing, ant colony algorithm and basic class concepts to produced the optimized test sequence. The class contains useful information like number of methods used, number of attributes used, dependency relation with other class. We exploit can all these important details and utilize them to generate the optimized test sequence.

3.3 Justification

The proposed approach makes extensive use of class and its basic features to generate the test sequence. The use of class in test sequence generation is quiet useful as it gives the complete modularity to system developed. To best of our knowledge there is no such approach which directly makes use of classes to generate the test sequence. Hence the proposed approach provide a new way of test sequence generation, which could revolutionize the testing of object oriented languages such as java, c++ , VB .Net etc.

3.4 Proposed Methodology

Proposed work describes a test tools which could generate the test sequence by directly using the program source code. The source code used here is java source code. The generated test sequence is based on the classes used in java source code. Developer can easily analyze the defect in the particular class and correct it. In the proposed approach, test sequence is generated on the basis of basic features of class like

- Dependency used to find honeypot class

- Attribute
- Methods
- Cyclomatic complexity of methods used in class.

The test sequence is optimized on the basis of these four parameters

3.4.1 System Overview

The proposed methodology focuses on using the classes and its basic features like dependency, methods, attributes and cyclomatic complexity of methods to generate the optimized test sequence. The method is based on the step by step calculation on the basis of above mentioned four parameters. Each parameter is used as a base for the comparison. The test sequence is generated on the basis of these four parameters. The best and optimized path to test is the one who's all four values:-

- Total number of Honeypot classes
- Total number of cyclomatic complexity of each methods in a class
- Total number of methods
- Total number of attributes

are maximum in order of appearance.

Various steps involved in proposed methodology can be briefly defined as follows:

- The source code browser searches the required source code to be tested on the system.
- The source code is parsed by the source code parser. The source code parser reads the code from the starting till the end. Every possible string of the source code is stored by the parser to gather the required information. The output of the source code parser is the plain text
- Then the class extractor finds the possible number of classes in the source code. The java source code serves as an input to the class extractor. These classes are then used to generate the test sequence
- The Parameter locator finds all the four parameters namely dependencies, attributes, methods and cyclomatic complexity of each class. It analyzes the each class separately to find out all the parameters required for the test sequence generation and comparison. All four parameters are assigned to each class automatically.

- The graph generator generates the graph on the basis of four parameters namely dependency, attributes, methods, and cyclomatic complexity. Four different types of graph are generated on the basis of these parameters. Each graph has its own importance; dependency graph is used to find the possible number of honeypot classes i.e. most vulnerable class by using the ant colony algorithm, attribute graph is used to find the test sequence on the basis of attribute value of each sequence, and so on.
- Finally test sequences are generated on the basis of graphs mentioned above for all four parameters and their total weight is calculated at the end of each path. The optimized path is chosen on the criterion of having maximum values of all the four parameters. The first priority is given to the maximum number of honeypot in the test sequence, second priority is given to the cyclomatic complexity , third to the methods and the last to the attribute values

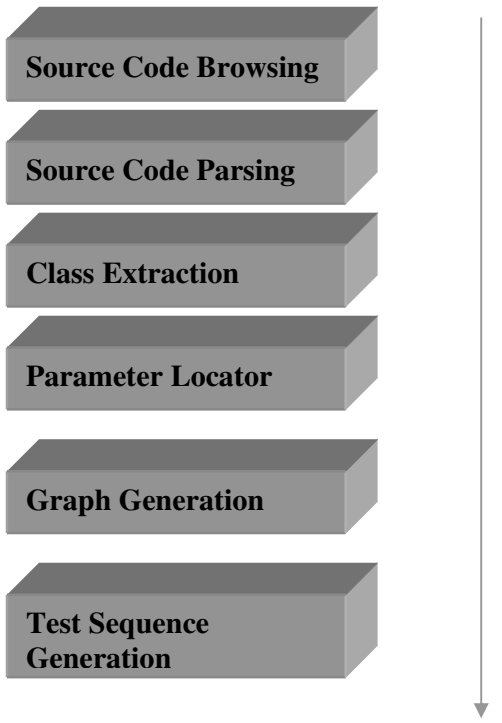


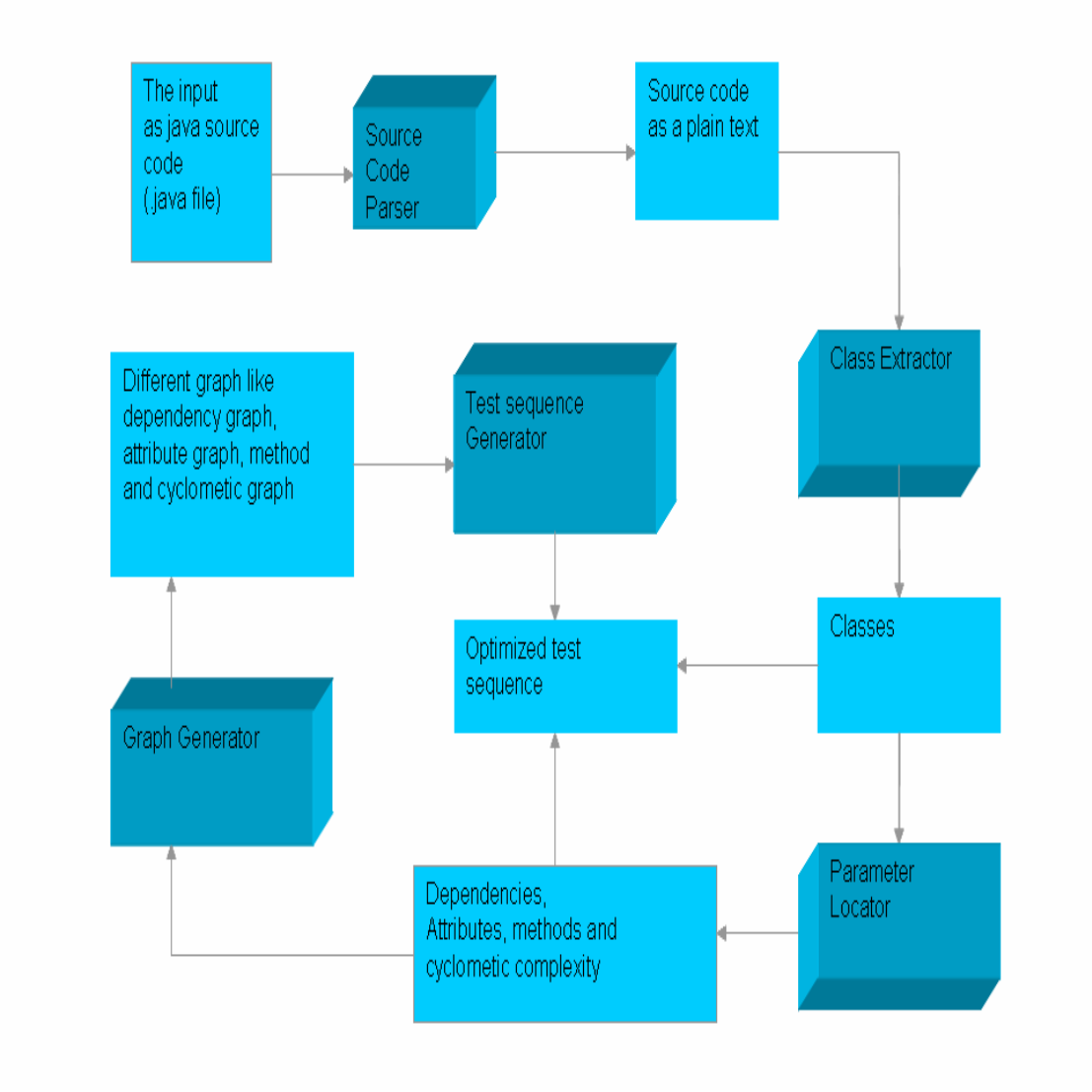
Figure 3.1: Step by step approach for test sequence generation

3.4.2 Working of the System

The sequential structure of the process is given in the figure 3.2, which shows all the steps involved in the process**.**

- **Input**

  The java code serves as an input to the tool proposed in this methodology. The tool requires the .java file of any java source code as an input. The .java file consists of various classes which are used to generate the test sequences. The generated test sequence is entirely consists of classes.



3.2: Detailed approach for test sequence generation

3.4.2.1 Source Code Parser

The work of source code parser is to read the source code from starting of the program till the end. The main purpose is to read the every string of the source code. After parsing, the source code is saved as a collection of string. Each class is saved with all the required information separately. After reading the source code it can be viewed as shown below.

```
Class one
{
}
Class two extends one
{
}
Class three extends one
{
}
Class four extends two
{
```

Figure 3.3: Source Code

3.4.2.2 Class Extractor

The work of class extractor is to find out the possible number of classes present in the program source code. The classes are found in fixed order, according to the their appearance in the source code

```
one
two
three
four
five
six
seven
.
.
.
```

Figure 3.4: Total no of Classes

3.4.2.3 Parameter Locator

Parameter Locator is very important part of the proposed system. The main purpose of parameter locator is to find out all the four parameter namely dependency, attributes, methods and cyclomatic complexity for every class. It plays the major role as every

parameter's value is crucial in determining the test sequence. All the information regarding each class is stored separately with all the values of parameters for each class

```
Class name= one
Dependency—one-->0
Attribute=a
Methods =m
Cyclomatic complexity=c
Class name= two
Dependency—one-->two
Attribute=b
Methods =n
Cyclomatic complexity=d
```

Figure 3.5: Parameter's value for each class

3.4.2.4 Graph Generator

Graph generator plays a vital role in the proposed system. It generates the four types of the graph

- **Dependency graph**

  The dependency graph is generated according to the dependency property of the class. The classes are arranged in the proper order according their dependency relation with the other class. This can be better understood by the diagram shown below.

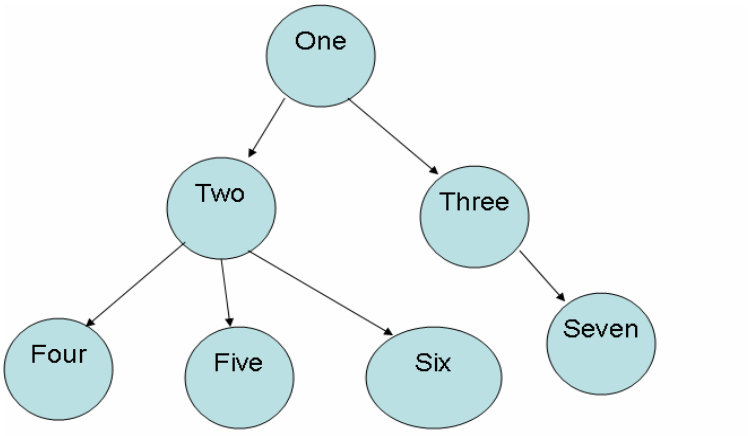


Figure 3.6: Dependency graph showing dependency b/w classes

- **Identification of Honey pot classes using ACO Algorithm and dependency graph**

  An alternative ACO based method is proposed in order to use ants to search the dependency graph for the identification of honeypot classes. In order to apply ACO algorithm to solve the problem of finding the Honeypot classes, a number of issues need to be addressed:

- Transformation of the testing problem into a graph. It is done by the graph generator.
- A mechanism for creating possible solutions efficiently and a suitable criterion to stop solution generation. Parameter locator is used to find the dependency among classes and the suitable criterion to stop the solution is the total class coverage.
- A suitable method for updating the pheromone. An algorithm is used to update the pheromone, discussed later
- A transition rule for determining the probability of an ant traversing from one class to another. Class with large amount of pheromone is chosen, in case of tie random choice is made.

Two requirements the generated test sequence has to satisfy:

- All-class coverage
- Feasibility – Each test sequence in the test suite represents a feasible path in the corresponding dependency class diagram

The following algorithm is proposed for an ant to explore the dependency graph:

- **Algorithm**

- Initial pheromone level of each class in graph=1; (Pheromone initialization)
- Move to the next class from the root class with equal pheromone with random choice.
- Move from one class to another class until leaf class found

  (a) If class is leaf class then move towards the parent class until any child class found is untraced

  (b) If any parent class in the backtracking has the child class untraced then increment the pheromone level of parent class by one and decrement the pheromone level of traced child by one.

(Pheromone updation and evaporation)

(c) Go to the step 2 until all class are covered

After covering the whole dependency graph and using the Ant algorithm the class which has highest number of pheromone level is termed as the *Honeypot class*. Test path is generated each time ant reaches the leaf class. Each test path is marked with total number of honeypot class in their sequence

For example considering the figure 3.6 and collecting the data in the tabular form after using the proposed algorithm

Table 3.1: Pheromone updation

Serial no.	Class name	Initial (ph value)	Final (ph value)	Honeypot classes
1	One	1	2	One
2	Two	1	2	Two
3	Three	1	1	
4	Four	1	1	
5	Five	1	1	
6	Six	1	1	
7	Seven	1	1	

The honeypot classes using the proposed ACO algorithm which has highest number. of pheromone level are *One and Two*

- **Attribute Graph**

  The attribute graph is generated on the basis of attribute values of each class. A test sequence is generated using the weight of each class as attribute. A simple binary search can be used to find the test sequence. After generating the test sequence, it is marked with the sum total of attribute value of each class

  For example consider the following graph; each class is marked with their attribute values

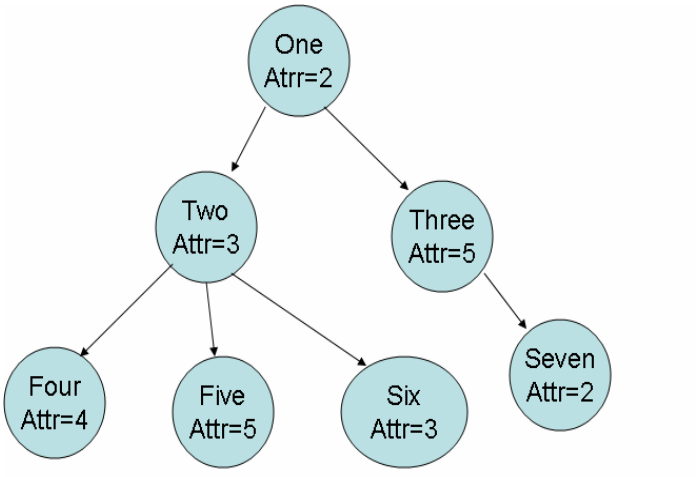


Figure 3.7: attribute graph

- **Method Graph**

  The method graph is generated on the basis of total number of methods used in each class. A test sequence is generated using the weight of each class as number of methods. A simple binary search can be used to find the test sequence. After generating the test sequence, it is marked with the sum total of no of methods used in each class

  For example consider the following graph; each class is marked with their no of methods used

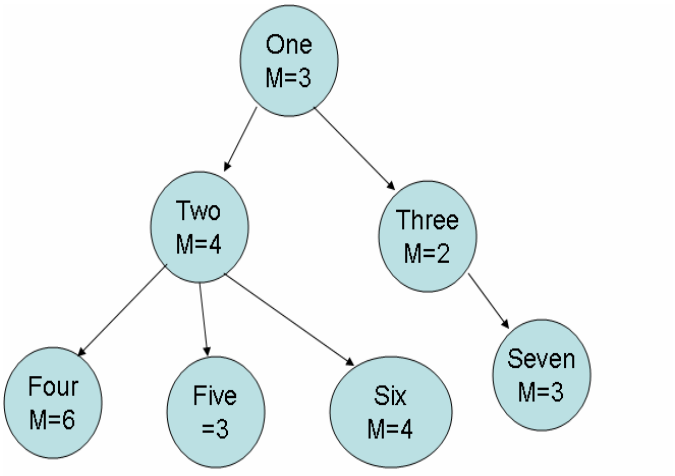


Figure 3.8: method graph

- **Cyclomatic complexity graph**

  The Cyclomatic complexity graph is generated on the basis of total number of cyclomatic complexity of each used methods in each class. A test sequence is generated using the weight of each class as considering cyclomatic complexity of each method. A simple binary search can be used to find the test sequence. After generating the test sequence, it is marked with the sum total of cyclomatic complexity of each method used in each class

  For example consider the following graph; each class is marked with the cyclomatic complexity in each class.

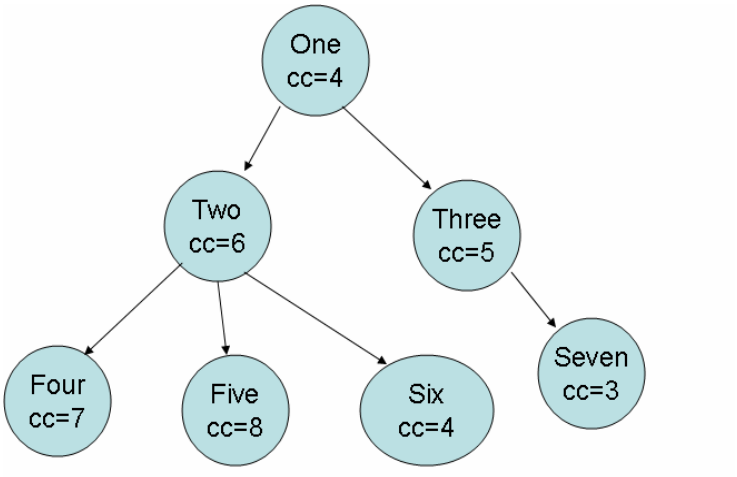


Figure 3.9: cyclomatic complexity graph

3.4.2.5 Test Sequence Generator

Test sequence generator generates the different types test sequence on the basis of all four parameters. Input to the sequence generator is a graph and output is the test sequence. It also sums up the weight of each class appears in test sequence at end of the each path

For example passing all different graphs mentioned above, the generated test sequences are:-

Test sequences for dependency graph with honeypot classes in bracket

(1) One—Two—Four (2)

(2) One—Two—Five (2)

(3) One—Two—Six (2)

(4) One—Three—Seven (1)

The test paths generated for attribute graph are:-

(1) One—Two—Four (sum of attributes of classes=9)

(2) One—Two—Five (10)

(3) One—Two—Six (8)

(4) One—Three—Seven (9)

The test paths generated for methods graph are:-

(1) One—Two—Four (sum of no. of methods of classes=13)

(2) One—Two—Five (10)

(3) One—Two—Six (11)

(4) One—Three—Seven (8)

The test paths generated for cyclomatic graph are:-

(1) One—Two—Four (sum of cyclomatic complexity of each methods used in each classes=17)

(2) One—Two—Five (18)

(3) One—Two—Six (14)

(4) One—Three—Seven (12)

- **Output**

  After comparing all the test paths with their appropriate weight of each four parameters dependency, attribute, no. of methods and cyclomatic complexity of each method in a class, optimized path can be identified.

  Considering all the above mentioned paths in tabular form

Table 3.2 Comparison table

Serial no.	Test sequences	Quality sets (H, C,M,A)
1	One—Two—Four	2, 17, 13, 9
2	One—Two—Five	2, 18, 10,10
3	One—Two—Six	2, 14, 11,8
4	One—Three—Seven	1, 12, 8,9

where h= total no of honeypot

c= total no of complexity

a= total no of attribute

m=total no of methods

After comparing the values of each parameter, the critical path is one whose all values is maximum i.e. One –Two—Five.

3.5 Discussion

The number of honeypot classes in a path is indicator of its complexity. If number of honeypot classes is more in a particular path, it means it is more important and needs more testing attention. Test paths generated on the basis of attributes value, methods value and cyclomatic complexity. Also helps the tester to identify more critical paths. Finally on the basis of combination of all these four factors, heavy weight means more critical test paths can be found.

Chapter 4
Experimental Results

As testing is one of the necessary tasks in the software development, efforts are continuously made in order to find the best and optimized test sequence to produce the software with minimum errors. There are several methods of test sequence generation like state based test sequence generation, mutation based test generation etc. There has not been any effort to use concepts of class in test sequence generation. The proposed methodology uses the various properties of class to generate the optimized test sequence. The advantage of using the class in test sequence is that we can immediately find out the erroneous class occurring in the source, hence can easily correct the defects. To support our work we present a case study.

4.1 Case Study

Consider an arbitrary java code of a virtual system for explaining the proper working of the system. The code consists of different classes, using dependency relation for communication between themselves. The outputs of different step carried out in the process have been supported by the diagrams.

- **Input**

  Input is java code, after parsing the java code using the source code parser, source code can be viewed as simple text as shown in figure 4.1

```
class one
{
}
class two extends one
{
}
class three extends one
{
}
class four extends two
{
}
class five extends three
{
}
class six extends two
{
}
class seven extends three
{
}
class eight extends six
{
```

Figure 4.1: Source code

- **Class extraction**

  Using the class extractor all class of source code is extracted as shown in figure 4.2

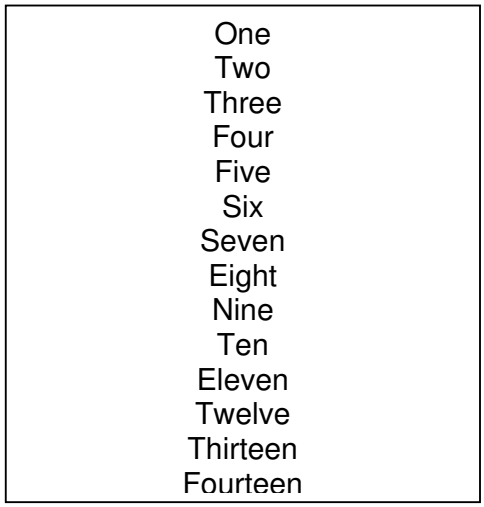


Figure 4.2: Name of class extracted

- **Parameter detection**

  Parameter locator is responsible for finding all the values of four parameters of each class.

  In tabular form all the values of parameters are shown in table 5.1

Table 4.1 Value of four parameter for each class

Serial no	Class name	Dependency of class	No. of attributes	No. of methods	No of cyclomatic complexity
1	One	Root class	5	5	10
2	Two	One----->Two	6	4	8
3	Three	One----->Three	4	6	12
4	Four	Two----->Four	3	3	9
5	Five	Three---->Five	5	7	14
6	Six	Two------>Six	4	6	12
7	Seven	Three---->Seven	5	4	10
8	Eight	Six------>Eight	5	2	6
9	Nine	Six------>Nine	3	4	8
10	Ten	Four----->Ten	2	4	7
11	Eleven	Five------>Eleven	6	5	14
12	Twelve	Eleven---->Twelve	5	7	16
13	Thirteen	Eleven---->Thirteen	5	6	13
14	Fourteen	Twelve----- >Fourteen	2	1	3

- **Graph generation and test sequence generation**

  Using the graph generator all the four the four types of graphs is generated.

- **Dependency graph**

  The dependency graph generated on the basis of dependency relationship between classes

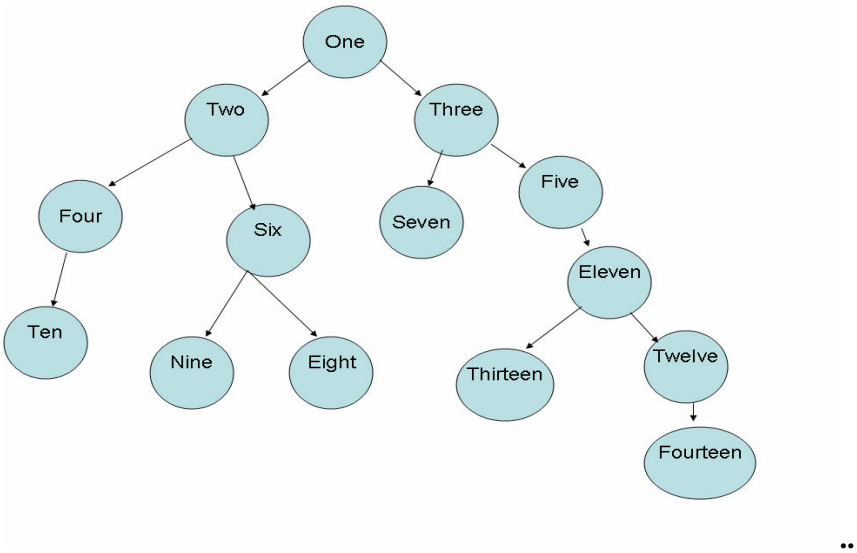


Figure 4.3: Dependency graph

- **Honeypot class identification using ACO algorithm**

  Using the proposed ant colony algorithm a table of pheromone variation is generated, using this table we find out the number of honeypot classes

Table 4.2: Pheromone updation

Serial no	Class name	Initial ph.	Final ph.	Honeypot class
1	One	1	2	One
2	Two	1	2	Two
3	Three	1	2	Three
4	Four	1	1	
5	Five	1	1	
6	Six	1	2	Six
7	Seven	1	1	
8	Eight	1	1	
9	Nine	1	1	
10	Ten	1	1	
11	Eleven	1	2	Eleven
12	Twelve	1	1	
13	Thirteen	1	1	
14	Fourteen	1	1	

With the help of pheromone table, the honeypot classes are

(1) One

(2) Two

(3) Three

(4) Six

(5) Eleven

Test sequence with the number of honeypot class in bracket

(1) One—Two—Four—Ten (2)

(2) One—Two—Six—Nine (3)

(3) One—Two—Six—Eight (3)

(4) One—Three—Seven (2)

(5) One—Three—Five—Eleven—Thirteen (3)

(6) One—Three—Five—Eleven—Twelve—Fourteen (3)

(1) Attribute graph

The attribute graph generated on the basis of attribute values of each class

Test sequence with total number of attributes in bracket

(1) One—Two—Four—Ten (16)

(2) One—Two—Six—Nine (18)

(3) One—Two—Six—Eight (20)

(4) One—Three—Seven (14)

(5) One—Three—Five—Eleven—Thirteen (25)

(6) One—Three—Five—Eleven—Twelve—Fourteen (27)

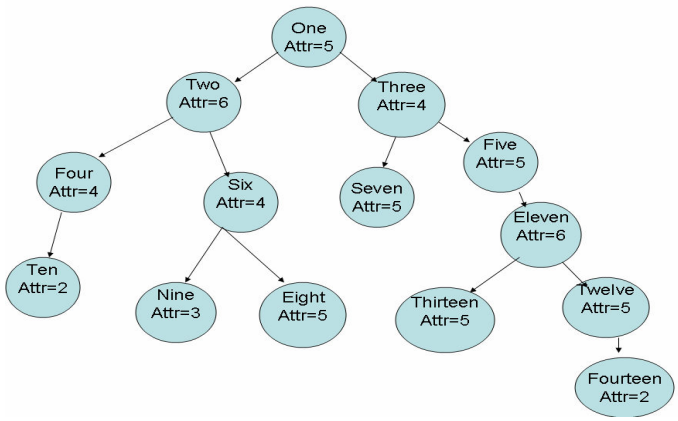


Figure 4.4: Attribute graph

(2) Methods graph

The methods graph generated on the basis of no of methods used in each class

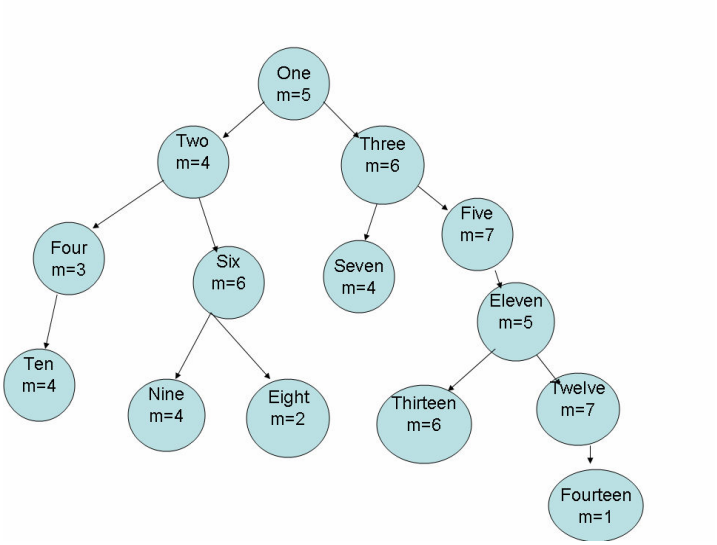


Figure 4.5: Method graph

Test sequence with total number of methods in bracket

(1) One—Two—Four—Ten (16)

(2) One—Two—Six—Nine (19)

(3) One—Two—Six—Eight (17)

(4) One—Three—Seven (15)

(5) One—Three—Five—Eleven—Thirteen (29)

(6) One—Three—Five—Eleven—Twelve—Fourteen (31)

(4) Cyclomatic complexity graph

The cyclomatic complexity graph generated on the basis of number of cyclomatic complexity of each methods used in each class

Test sequence with total number of cyclomatic complexity of each methods in bracket

(1) One—Two—Four—Ten (34)

(2) One—Two—Six—Nine (38)

(3) One—Two—Six—Eight (36)

(4) One—Three—Seven (32)

(5) One—Three—Five—Eleven—Thirteen (53)

(6) One—Three—Five—Eleven—Twelve—Fourteen (69)

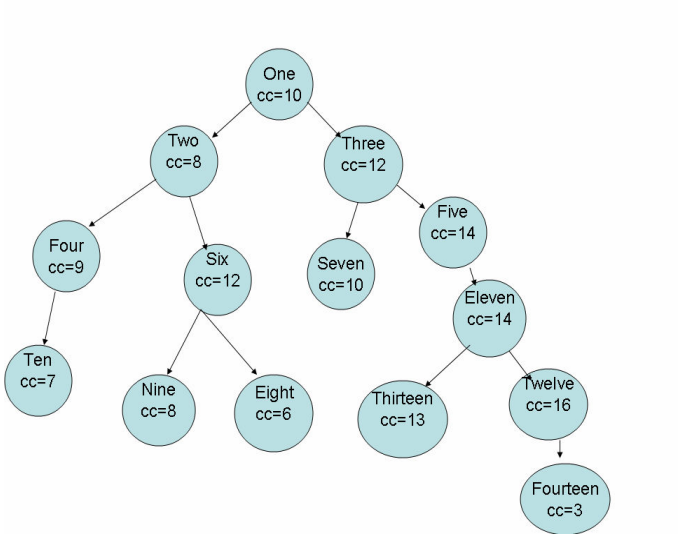


Figure 4.6: Cyclomatic complexity graph

(5) Comparison of graphs

Results of all graph is summarized in table for comparison. In comparison, most optimized path is one whose all four parameter's value is maximum in order of

a) Total no of honeypot classes
b) Total no of cyclomatic complexity of each methods used in each class
c) Total no methods
d) Total no of attributes

The values is shown in table below

Comparison table:

Table 4.3: Comparison of test sequence

Serial no	Test sequence	Quality set (H,C,M,A)
1	One—Two—Four—Ten	2, 34, 16, 16
2	One—Two—Six—Nine	3, 38, 18, 19
3	One—Two—Six—Eight	3, 36, 20, 17
4	One—Three—Seven	2, 32, 14, 15
5	One—Three—Five—Eleven—Thirteen	3, 53, 25, 29
6	One—Three—Five—Eleven—Twelve—Fourteen	3, 69, 27, 31

5.2 Experimental Results

- **Objective:** To find the optimized test sequence for any java source
- **Input:** The input to the system is .java file of any system
- **Output:** the output is the optimized test sequence

  By analyzing the comparison table the test sequence with highest value of all four parameters in order of preference is

  One—Three—Five—Eleven—Twelve—Fourteen

  Hence it is most optimized path to be tested first in given java source code
- **Efficiency and Limitations:** The proposed methodology efficiently determines the most critical path. But it works only for java source code; it may be extended for other object oriented language also like c++.

The analysis based on the above case study supports the proposed methodology. The above case study shows that the results are as expected. It supports the proposed view of using java code as object-oriented language to find the optimized test sequence.

Chapter 5

Conclusion and Future Scope

Software Testing remains the challenging task in software development. The automation of testing helps in faster result of the test cases generation but its complete automation remains an illusion. There are various methods used for the automatic test sequence generation, but there still remains the lack of completeness and adequacy in test cases. There are various techniques used for the test sequence generation but none of them provide the complete satisfactory results. There is always a search for better and most optimized way to generate the test sequence so as to save the bulk amount of time spending on testing. The use of classes for the test sequence generation has not been explored; it can be very useful simple for the object oriented codes.

The proposed work is an effort to search a new technique "*Test Sequence Generation and Optimization*" which uses the class and its basic features to generate the test sequence; it can be very useful and handy for the testing of object oriented systems.

5.1 Conclusions

The proposed system used the basic class features to generate the optimized test sequence. The proposed methodology uses the .java file from java source code as an input and after extracting all classes with its basic features generates the optimized test sequence

The major contribution of this work is summarized:-

- **Appropriate Covering of Defects: -** As the test sequence entirely consists of classes, the defects can be more specifically identified for the appropriate classes where it occurs.
- **Reduced Testing Time:-** In this approach test sequence is generated by directly using the code without any modification, thus can be done at any time during code generation

- **Simple and Robust: -** The proposed methodology presents a new approach of using the classes in test sequence generation which is simple to use and predict test sequences based on four parameters.

5.2 Future Scope

The work done can be enhanced further in following ways:

- The proposed system is limited to use the .java file as an input. It can be extended to use the other object oriented language such as C++, Ruby, VB .Net etc
- Optimization parameter**:** The proposed methodology uses only four parameter dependency, cyclomatic complexity, methods and attributes. Further work can be extended to include more parameters such as association, generalization and aggregation etc.

References

[1] B.Beizer. “Software Testing Techniques”, Van Nostrand Reinhold, 2nd edition, 1990.

[2] Colorni A., M. Dorigo and V. Maniezzo. Distributed Optimization by Ant Colonies. Proceedings of ECAL91 - European Conference on Artificial Life, Paris, France, F.Varela and P.Bourgine (Eds.), Elsevier Publishing, 1991, pp.134--142.

[3] C. Hurkens and S. Tiourine, Upper and lower bounding techniques for frequency assignment problems, Technical Report, T.U. Eindhoven 1995. pp. 95-34

[4] Doerner, K., Gutjahr, W. J., “Extracting Test Sequences from a Markov Software Usage Model by ACO”, LNCS, Vol. 2724, Springer Verlag, 2003. pp. 2465-2476

[5] F. Glover, Tabu search, ORSA Journal on Computing 1 1989, pp. 190--206.

[6] Gonzalez, E. A. Ant colony-based solutions for sub-optimal control systems. M.Sc. Thesis, Graduate School of Engineering, De La Salle University – Manila, Philippines. 2006.

[7] Handl, J., Knowles, J. and Dorigo, M. (2006) Ant-based clustering and topographic mapping. Artificial Life *12(1).*

[8] J.H. Holland, Adaptation in natural and artificial systems, University of Michigan Press, 1975.

[9] L.M Gambardella and M. Dorigo M, Solving Symmetric and Asymmetric TSPs by Ant Colonies , Proceedings of the IEEE Conference on Evolutionary Computation ICEC96, Nagoya, Japan, May 20-22, 1996, pp. 622-627.

[10] M. Dorigo and L.M. Gambardella, Ant colony system: a cooperative learning approach to the traveling salesman problem, IEEE Transaction on Evolutionary Computation 1, 1997, pp. 53--66.

[11] M. Dorigo, V. Maniezzo, and A. Colorni, The ant system: an autocatalytic optimizing process, Technical Report TR, Politecnico di Milano 1991, pp. 91-016

[12] M.Dorigo, Ant colony optimization web page, http://iridia.ulb.ac.be/mdorigo/ACO/ACO.html

[13] M.Dorigo, G. Di Caro & L.M. Gambardella Ant Algorithms for Discrete Optimization. Artificial Life, 5(2), 1999. pp. 137-172.

[14] M. Dorigo, Optimization, learning and natural algorithms, Ph.D. Thesis, Politecnico di Milano, Milano, 1992.

[15] M. Dorigo, T. Stützle. The ant colony optimization metaheuristic: Algorithms, applications and advances. In F. Glover and G. Kochenberger, editors, Handbook of Metaheuristics. Kluwer Academic Publishers, to appear in 2002.

[16] M. Dorigo and L.M. Gambardella, Ant colonies for the traveling salesman problem. Biosystems 43, 1997, pp. 73-81.

[17,] M. Dorigo, M. Birattari, and T. Stützle Ant colony optimization: Artificial ants as a computational intelligence technique. IEEE Computational Intelligence Magazine, vol. 1, no. 4, 2006, pp. 28-39.

[18] M. Dorigo., Maniezzo, V., Colorni, A., **"**Positive Feedback as a Search Strategy", Politecnico di Milano, Italy, 1991, pp. 91-016

[19] M. Dorigo., Maniezzo, V., Colorni, A**., "**The Ant System: Optimization by a Colony of Cooperating Agents", IEEE Transactions on Systems, Man, and Cybernetics-Part *B,* Vol. 26, No.1, 1996, pp. 29-41

[20] McMinn, P., Holcombe, M., "The State Problem for Evolutionary Testing", Proc. GECCO 2003, LNCS Vol. 2724, Springer Verlag, 2003. pp. 2488-2500

[21] M. Birattari, P. Pellegrini, and M. Dorigo On the invariance of ant colony optimization. *IEEE Transactions on Evolutionary Computation,* vol. 11, no. 6, 2007, pp. 732-742.

[22] M. Manfrin, M. Birattari, T. Stützle, and M. Dorigo Parallel ant colony optimization for the traveling salesman problem. P.Y. Schobbens, W. Vanhoof, and G. Schwanen (Eds.) *BNAIC 2006: 18th Belgium - Netherlands Conference on Artificial Intelligence,* 2006, pp. 409-410,

[23] R. S. Pressman. "Software Engineering: A Practitioner's Approach", 3rd Edition, McGraw Hill, New York, 1992, p. 559.

[24] S. Kirkpatrik, C.D. Gelatt, and M.P. Vecchi, Optimization by simulated annealing, Science 220 1983, pp. 671--680.

[25] S. Lin and B.W. Kernighan, "An effective heuristic algorithm for the traveling salesman problem," Operations Research, vol. 21, 1973, pp. 498–516

[26] T.A. Feo and M.G.C. Resende, Greedy randomized adaptive search procedures, Journal of Global Optimization 6 1995, pp. 109--133.

[27] T. Back and H.-P. Schwefel, An overview of evolutionary algorithms for parameter optimization, Evolutionary Computation 1(1), 1993, pp. 1-23.

[28] Thomas St¨utzle and Holger Hoos. Improvements on the Ant System: Introducing max-min Ant System. In Proceedings of the International Conference on Artificial Neural Networks and Genetic Algorithms (ICANNGA'97), Springer Verlag, Wien, 1997, pp. 245–249,

[29] V. Maniezzo, A. Mingozzi, and R. Baldacci, A bionomic approach to the capacitated p-median problem, Journal of Heuristics 4(3) 1998, pp. 263--280.

[30] V. Maniezzo, Exact and approximate nondeterministic tree-search procedures for the quadratic assignment problem, INFORMS Journal of Computing 11(4) 1999, pp. 358--369.

Web References

[31] Class, http://en.wikipedia.org/wiki/Class_(computer_science)

[32] Class diagram, http://en.wikipedia.org/wiki/Class_diagram

[33] Class, http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Class.html

[34]Cyclomaticcomplexity,http://www.sei.cmu.edu/str/descriptions/cyclomatic_body

[35] Cyclomatic complexity, http://en.wikipedia.org/wiki/Cyclomatic_complexity

[36] Fields,http://en.wikipedia.org/wiki/Field_%28computer_science%29

[37] Methods, http://en.wikipedia.org/wiki/Method_%28computer_science%29

[38] Object oriented programming, http://en.wikipedia.org/wiki/Object-oriented_programming

[39]SoftwareTesting,http://www.ece.cmu.edu/koopman/des_s99/sw_testing/referenc

[40] The Open Group, http://tetworks.opengroup.org/.