"Data Flow Based Automated Integration Testing Framework for Object Oriented Programs Using Evolutionary Approach"

Author

Shahzada Zeeshan Waheed
NUST201362553MCEME35413F

Supervisor

Dr. Usman Qamar

DEPARTMENT OF COMPUTER ENGINEERING
COLLEGE OF ELECTRICAL & MECHANICAL ENGINEERING
NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY ISLAMABAD

SEPTEMBER, 2015

"Data Flow Based Automated Integration Testing Framework for Object Oriented Programs Using Evolutionary Approach"

Author

Shahzada Zeeshan Waheed
NUST201362553MCEME35413F

A thesis submitted in partial fulfillment of the requirements for the degree of

MS (Computer Software Engineeing)

Thesis Supervisor

Dr. Usman Qamar

Thesis Supervisors Signature:_____

DEPARTMENT OF COMPUTER ENGINEERING
COLLEGE OF ELECTRICAL & MECHANICAL ENGINEERING
NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY ISLAMABAD

Author's Declaration

I certify that this research work titled "Data Flow Based Automated Integration Testing Framework for Object Oriented Programs Using Evolutionary Approach" is my own work. The work has not been presented elsewhere for assessment. The material that has been used from other sources it has been properly acknowledged / referred.

Shahzada Zeeshan Waheed
NUST201362553MCEME35413F

Dated:

Language Correctness Certificate

This thesis has been read by an English expert and is free of typing, syntax, semantic, grammatical and spelling mistakes. Thesis is also according to the format given by the university.

_____

Shahzada Zeeshan Waheed
NUST201362553MCEME35413F

_____

Signature of Supervisor

Copyright Statement

Acknowledgements

I would like to thank Almighty Allah for His constant grace showered on me and His increasing gift of knowledge and strength that has relentlessly prevailed my life through the entire thesis work. I express my sincere thanks to my thesis advisor Dr Usman Qamar (Assistant Professor CEME, NUST,Islamabad) for guiding me right from the inception till the successful completion of the project. I sincerely acknowledge him for providing me valuable guidance, support for literature, critical reviews of research and the report and above all the moral support he had provided me in all stages of my work. I would also like to thank Dr. Saad Rehman , Dr. Wasi Haider and Dr. Rehan Hafiz for being on my thesis guidance and evaluation committee.

I am also thankful to my wife and whole family for their support and encouragement. I wish to acknowledge my great debt to all of them whose ideas and contribution influenced me to complete the thesis work.

Dedicated to my exceptional parents, adored siblings and supportive wife whose tremendous support and cooperation led me to this wonderful accomplishment

Abstract

Software is widely used technology of this era. Applications are being developed, solving tasks ranging from simple calculations to complex banking transactions and so on. Use of software in critical systems leaves no space for even simple errors which can lead to sever financial loss or threat to human life. Software must be well tested to verify its functioning. Manual testing is tedious and time consuming job. Object oriented programs use classes and their interactions to perform tasks. Integration testing requires to test the interfaces. Path explosion make it worst to test the integration of classes. Small work done in integrated test case generation of OO programs. Coupling criteria is used for generation of automated test cases in proposed approach. Search space is reduced by selecting only coupling methods that are directly involved in integration. Objects must be in proper state before testing process started. Further def-use analysis helps in achieving the desired object states for proper interfaces testing, representing methods in intermediate tree. Method sequences are generated considering data flow of involved state variables. Test can't be accomplished without proper input data. We propose a fitness function for test data generation considering coupling path coverage. Genetic algorithm is used to optimize the solution based on proposed fitness function. Eleven randomly selected project are used from SF100 (Software Testing Benchmark) to show the strength of proposed approach. Results showed relatively high coverage as compared to random testing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software testing is the process of identifying the defects and ensuring the quality of the software. There are two types of testing, white box testing and black box testing. In black box testing softwares inner details are hidden. Input and expected outputs values are considered in black box testing for testing functionality. But white box testing tests the internal structure of the system. Coverage criteria like branch coverage, statement coverage and decision coverage etc. can be used for white box testing. Testing can be performed in unit, integration or system level. Software testing can only be achieved with the help of proper data. However, doing this manually is a difficult task due to the presence of huge number of predicate nodes in the module. Test data generation in code based testing, is the process of identifying a set of test data, which satisfies the given testing criterion. Up till now most of the automated testing approaches deals with the unit tests. Small work done in the area of automated integration testing. Especially in object oriented program automated test generation for integration testing is difficult task. At integration level, the variables are passed as arguments to other components and variables change their names; also multiple paths are executed from different components to ensure proper functionality. Search based software engineering can be used for testing software oriented problems using evolutionary approach.

1 Motivation

Testing is important to ensure the software quality at different levels of software development. Early identification of bugs saves the cost of redoing. Object oriented programming is most widely used programming approach. It attempts to provide a model for programming based on objects. At integration level, difficulty increase; also multiple paths are executed from different components to ensure proper functionality. There is a need for a framework to test the object oriented programs at integration level. Test cases in this case are method sequences leading to integration. Objects must be in proper state, before testing to be performed. Proper methods sequence calls helps to achieve it. Manual testing is a tedious job so there is a need for efficient framework to test object oriented program at integration level that covers both method sequence and the test data generation.

1.1 Problem statement

Software testing is active field of research. Manual testing is a tedious job.Research focus is moved towards automated testing to save resources and time. In Object oriented integration testing, classes interact with each other leads to path explosion. There is no approach available for automated test case generation of OO programs covering integration criteria. Test cases are accepted from user in approach proposed by Khan and Nadeem (2014). Deciding for suitable coverage criteria is also a challenging task. There is a need for framework to cover object oriented integration testing, generating test cases as well as test data to accomplish the testing process.

1.2 Objective and Contribution

Here is the list of objectives and contribution that we made in area of automated software testing.

- Setting the criteria for integration testing (coverage details)
- Automated test case generation achieving data flow coverage
- Design the fitness function for test data generation in integration testing
- Proposing the model for test data generation using evolutionary approach)
- Case study to show the significance of newly proposed approach
- Developing the prototype tool for object oriented integration testing based on proposed approach.

2 Outline

Chapter 2 discusses the background. In chapter 3, we provide details about software testing including coverage details and level of testing etc. Chapter 4 includes literature review. We present the design and methodology of our proposed approach in chapter 5. Chapter 6 discusses the experimental setup, case study and results followed by detail comparison of our approach with existing techniques. Finally, chapter 7 concludes this thesis and proposes an outlook of possible extensions, modifications, and improvements as future work.

Chapter 2

Background

1 Object Oriented Software

Object oriented is the paradigm which maps the real world entities into program domain. These objects contain data and specified behaviours. Data is normally known as attributes or fields and behaviours are specified using methods or functions. OOSE was developed by Ivar Jacobson in 1992 Jacobson (1992). In OO Development basic unit for building application is known as Class. Each class represents real world entity and serve as blueprint for objects. These objects interact with each other to perform particular task. OO programming has many advantages over procedural language like code reusability, encapsulation, design benefits and software maintenance etc.

Class: Defines the template for the objects.

Object: Is the instance of class.

It contains data values to represent the real world entity Object is the representation of the real world entity. Properties of the objects can be categorized into following three categories state, services and identity Orso (1998). State represents the value of the attribute of the objects. Attributes can be primitive or user defined types. Services are also known as methods. State of the object can only be changed through methods. Constructor is called at the time of object creation. There can be more than one constructor. Arguments determine which constructor to be called. Identity is the property of the object that separate it from other object even if both objects are in same state.

1.1 Code Reusability

It saves time and resources for doing the redundant tasks. Object oriented provides powerful feature to reuse the code. Key idea behind the code reusability is using the code written for specific purpose and reusing it for construction of another program or software. In OO domain code can be reused by simply adding existing resources or reusing it through inheritance.

1.2 Encapsulation

Encapsulation allows object to hide its data members from illegal use. Three type of access modifiers are used for this purpose. With *private* modifier these attributes can only be accessed within the same class. *Protected* modifier extends it to subclasses as well,

protected attributed can further be accessed from sub-classes. Third modifier is the *public* that allows direct access to the attributes. Program can take advantage from *private* and *protected* access modifiers to prevent other programmers from illegal tempering of the data. Additionally object also defines how other entities will interact with it using *public* methods or member functions.

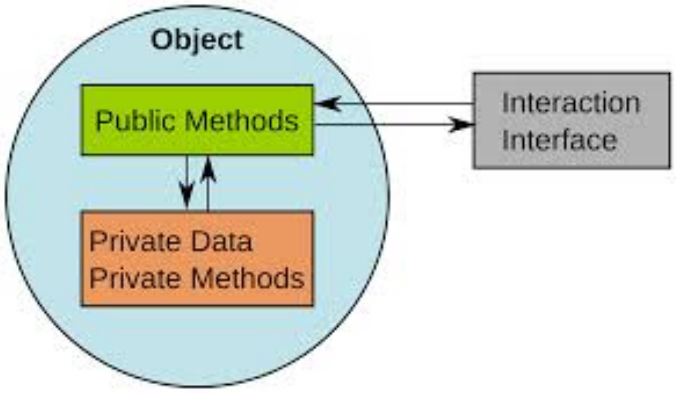


Figure 2.1: Encapsulation

Object contains both data members and methods encapsulated together. Only public interfaces are used for interaction.

1.3 Abstraction

Abstraction and encapsulation are related terms. Abstraction is used to manage the complexity. Use of classes and objects enables to see a system as components reducing the complexity. Programmer came to know about the inner details of the system as development progress. It enable stakeholders to see from particular view of interest and leaving the other details behind.

1.4 Design Benefits

It is very difficult to manage large programs. Thinking software as collection of interacting objects make it easier to manage it when complexity is increased to the certain extend. Planning phase benefits in improving design to cope complexity and flexible designs.

1.5 Software Maintenance

In software development we cannot resist changes. Legacy software must be maintained and accommodate changes to adopt the new environment. We cannot simply throw the software. In OO Design as lot of time is spent in design and development, less time is required to maintain it.

1.6 Properties

In object oriented programming, a class objects may interact with other objects to accomplish required task. In OOP three type of relationships are available inheritance,

composition and aggregation. These relations are defined depending upon the nature of the relation between objects.

1.6.1 Inheritance

Inheritance is a kind of relation in which one class is based on another class. Subclass inherits the properties from the base class. This concept is same as family inheritance, children inherits the properties from parents. This is core of reusability. Subclass can access the attributes and properties of base class depending upon the access modifier used. Inheritance can be single, multiple or multilevel. In single inheritance subclass inherits the features from single base class. In multiple inheritance class can have more than one base or super classes. Third form for inheritance is multilevel, in which inheritance can be seen as chain. Subclass inherits properties from another subclass and so on.

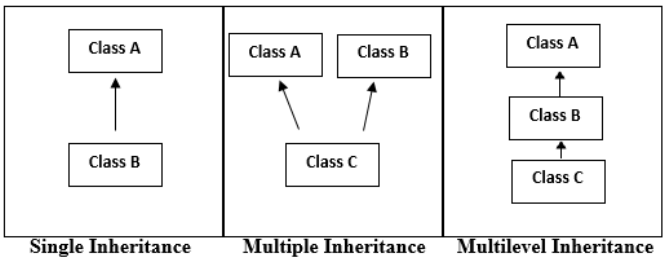


Figure 2.2: Types of Inheritance

1.6.2 Composition Vs Aggregation

Composition is the relationship between classes when one class object is physically included into another class. It is also known as *has a* relation. This is the strong type of relation. Contained object cannot exist outside the container class. Car and door is the pure example of composition. Car object compose of doors, but there is no existence of door outside the car object. In other hand aggregation is also *has a* relation. But difference between composition and aggregation is that, a contained objects has their own identity outside the container class. Company and employees is a classic example of aggregation. Company has employees, but employees can exist even if the company object is destroyed.

2 Class

Object oriented programming allows to define non-primitive data types. Primitive datatypes are defined by programming language like int, char, double, float etc. Non-primitive types are *use defined* types. Class is used in defining such structural elements. Class serves as a template and provides structure for the new data types. Object Oriented programming use classes as building blocks. As abstract view, we can say that class has two parts: Member variable and member functions. Member variable are the data fields and defines the properties of that class. Data variable can be primitive or non-primitive depending upon the requirements. A class can holds the object of another class and it depends upon the nature of the relation between classes. Composition and aggregation concept

implies in this situation. Member functions shows behaviour of the class. Behaviour is the functionality that an object can perform. As class is the mapping of real world entity, behaviours implementation help in actual reflection of real world entity. Access modifiers used to define the access to the resources. Interaction with class is done using public interfaces. Methods declared as public can be directly accessed from outside the class. Here is the example of class showing structure.

```
class Person {
  float height;
  double weight;

  Person() {
    height=0.0f;
    weight=0.0;
  }
  Person(float h, double w) {
    height=h;
    weight=w;
  }

  void setHeight(float h) {
    height=h
  };
  void setWeight(double w) {
    weight=w
  };
  float getHeight() {
    return height
  };
  double getWeight() {
    return weight
  };
}
```

Figure 2.3: Example of Class

Fig 2.3 shows class *Person* having two data member height and width. This class includes one constructor to initialize default values. Class can have more than constructors, and it is called when object is created. Class is by default private and private members are not accessible from outside class. This Person class contains four member functions. Information hiding is the property of object oriented program and to enforce it all data members are kept as private. There are three type of modifiers used for defining different level of visibility: public, private and protected. Private and protected members cant be accessed from main function directly using class object. Protected members are only accessible within the same class and subclasses (in case of inheritance). Public members are directly accessible outside the class using class objects.

```
Person p1=new Person ();
// This is not correct if constructor is private
Float height=p1. height;     // wrong
Float height=p1.getheight (); //Correct
```

2.1 Polymorphism

Word polymorphism means multiple forms. This is the important feature of object oriented programming. In OOP, it is possible to refer one type object to another type at runtime. This is called polymorphic behaviour. This functionality can only be achieved by overriding the specific method. Method signature include method name, parameters and return type Smith (2015). Program decide on runtime, which method to be called depending upon the referred class object. Even it does not require to change any code. Polymorphism allows inherited classes to add methods with same functions but different implementation. Late-binding keep it pending until actual method invocation. In general there are four type of polymorphism runtime, compile time, ad-hoc polymorphism and coercion polymorphism. Runtime polymorphism is well known type polymorphism. It is the case in which base class reference is used to hold derived or sub classes object on run time. As reference of the base class is used, compiler decide at runtime which method to call. Compile time polymorphism is also known as parametric polymorphism. Compile time polymorphism make it possible to execute same piece of code on different data types. Templates in C++ and generic type in java are used to implement it. Ad-hoc polymorphism is implemented through function overloading. Overloading functions has same name but different signatures. For example plus method taking two integer arguments returns sum of both variables whereas another function with same name but receiving two strings will return a concatenated string. Coercion polymorphism is when some type is implicitly converted to another type. Same object oriented concepts are used in each programming language with slight different implementation details. As in java there is no multiple inheritance. Instead you can achieve the same functionality by implementing the interface. In other hand, C++ allows to use multiple inheritance. There is no concept of pointer in java and all the object are reference variables. We are required to assign proper memory before using it. In C++ we explicitly need to define the destructor to release the memory but in java this is done automatically by garbage collector. Object oriented is powerful language and due to its vast features like code reusability, design and maintenance benefits keeping it alive and it is the best choice for every single project now a days.

Chapter 3

Software Testing

Software testing is way of ensuring quality, it is the process of identifying the defects and ensuring the quality of the software. It executes the software with intend to find defects in it. Testing is an important phase of software development life cycle (SDLC). Testing is a procedure that enhances the nature of programming. The objective of testing is to discover more bugs in an orderly and powerful way. Testing can be characterized as the procedure of checking and accepting a product item. Process validation may be performed at the start of an existence cycle to produce the item right. Verification and validation are two independent procedures used for quality assurance of software products.

Validation is the process of assuring the quality as software product meets the requirements. Validation is customer oriented. In other hand verification is process oriented. It ensures that required process and constraints are properly imposed to produce this product. A good quality software meets following requirements:

- Meets the prerequisites that guided its implementation and maintenance
- Meets the stakeholders requirements
- Reacts efficiently to a wide range of inputs
- Performs its capacities within the adequate time
- It is usable
- Keep running in the required constraints and provide expected behaviour

Software tester needs to check whether a given input maps to the correct output. There are different level and approaches used to ensure the software quality. Testing ca be done manually or automated. Manual software testing is tedious job to do Whittaker et al. (2000). Testing invests more than 50% of the total software development cost Kao et al. (1999). Improper testing leads to delivering of the useless product that does not meet the specified requirement. It results in losing the business repute and valued customers. So its worth investing in testing instead of delivering a low quality product. There are two types of testing, white box testing and black box testing. In black box testing softwares inner details are hidden. Input and expected outputs values are considered in black box testing for testing functionality. But white box testing tests the internal structure of the system. Coverage criteria like branch coverage, statement coverage and decision coverage

etc. can be used for white box testing. Testing can be performed in unit, integration or system level. Software testing can only be achieved with the help of proper data. However, doing this manually is a difficult task due to the presence of huge number of predicate nodes in modules under test. Writing manual tests is tedious task specially when working with the large enterprise level applications. Object oriented programming reflects the real world entities in the programming domain. It is preferred paradigm for developing large scale applications. It divides the system in to units that interact with each other to perform required tasks. Reusability is the plus for using OOP, It can be achieved by reusing a class or by inheritance. Encapsulation and information hiding allows to combine properties and behaviours of the object into a single entity hiding its properties and allowing interfaces to interact with other entities. Traditional techniques for testing software includes functional testing, branch testing and statement testing. These techniques are useful in finding defects written in procedural languages. But these techniques cannot be applied to the OO application as it is Kao et al. (1999). Testing a single class is relatively easy task as compared to integration testing of object oriented software. Integration testing involves more than one units and they interact with each other to perform a resulting task.

1 Levels of Testing

Generally there are four level of testing unit, integration, system and acceptance testing. Unit and integration testing is performed by developers while system and acceptance testing is done with coordination of stakeholders or users of the system. Each level

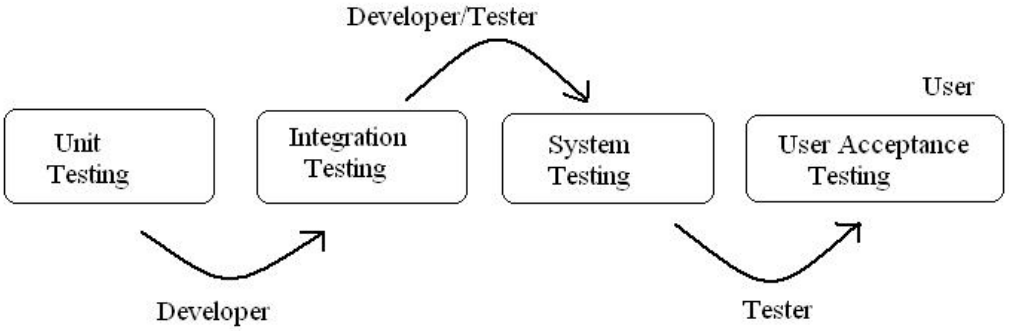


Figure 3.1: Levels of testing

involves different methodologies while assessing the quality. Testing must be performed in each level to ensure quality. Goal of software testing is to identify defects. Role of testing is to reveal the cases when system does not behave as expected. A successful test is more capable of finding defects. There are three main steps involves to perform this tasks: set of input values, determine the expected behaviour and get the actual results. Data used to perform test is known as test data. Exhaustive testing is impossible. Testing can only be performed to some extent. This means that there exists a trade-off between the accuracy of the test and the number of selected data **?**. Selected best input data is more important for efficient testing.

There are three basic form of testing.

- Black Box testing

- White Box testing
- Grey-Box Testing

1.1 Black Box Testing

Black box testing is defined as "Black box testing (also called functional testing) is testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions" Williams (2006). Its main focus is on functionality, testers run the program and check the output on specified input. Black box testing is mostly done with end user.

1.2 White Box Testing

White box testing, tests the inner structure and working of the system and it is done by developer. In this case input data is prepared to meet the specified coverage criteria. It can be performed in each level of software testing. Paths can be tested within unit, integrations are system level. This technique cannot be used to validate the requirements or check unimplemented parts, instead it can only detect internal faults or design of the system. White box test is done by the developer and it is expensive then black box testing.

1.3 Grey-box testing

In grey-box testing, we have limited internal knowledge about the software under test. It takes advantage of both black box and white box testing.

2 Testing techniques

Testing techniques are categorized in following four categories Orso (1998). Each technique is only capable to identify specific class of errors. Selection of testing technique is purely problem oriented.

- Specification Based Testing
- Program Based Testing
- Fault Based Testing
- Model Based Testing

2.1 Specification Based Testing

It is also known as functional testing. Testers test the functionality according to the requirements. It is the form of block box testing. Tester only knows the functionality of the system, instead of internal structure. Input is mapped to the required output. Requirements are used to drive test cases. Its purely a functionality testing and code is not directly used for specification testing Kaner (2006). Specification can be collected from following forms:

- Requirement document
- Use Cases
- Formal Descriptions

Specification of written document are the simple english statements. It is the description of the functionalities of the system. Use cases are the interaction of the actors to the system. It shows the system functionality with respect to the sequence of actions. Use cases organize the functional requirements. There can be more than one scenario of the specific action, i.e. alternate scenario. Formal specification is the mathematical description of the system specifications. It can also be used to provide an evidence that the system is correctly specified expressing syntax and semantics.

Here are some common specification based testing techniques.

Equivalence Partitioning: It is also known as equivalence class partitioning. Instead of single point, testing should be done on each domain of the program. Input values are used to define the partition of the system domain. Selection of test data from each domain is more effective then selecting it using single point. Evaluating only one test case from each class is enough to show the proper working of the system. At-least one test case from each partition must be evaluated. This is effective way of reducing test cases. Partitions must be selected wisely, because efficiency of the test purely depends on the viability of the partitions. Here is example of class partitioning, where input data is considered valid month in range of 1-12.

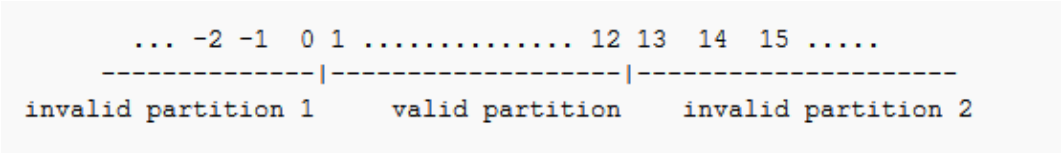


Figure 3.2: Class Partitioning

In above example there are three classes: One valid partition and two invalid partition. Next step is to select test input from each partitioning class. For example -10, 9, 200 is more effective to test the above scenario instead of 4, 7, 100.

Boundary Value Analysis: Boundary value analysis, tests are designed to include values form the boundaries in a specific range. This technique uses equivalence class partition to derive the tests. There is a boundary between each partition, and values on the boundary are considered in this case. Sometimes its very difficult to find boundary values. Consider example in figure 5. There are three partitioning classes and two boundaries between them. Data that lies on the boundaries is selected for testing in boundary value analysis. In this case test would be 0, 1 from first partition and 12, 13 from the second partition.

Cause Effect Graph: This is one of the black box testing technique. It is the graphical representation of the causes linked with the effects. Causes are the input values

that are mapped to the effects (output). Sequence in generating the test may involves examining the causes to find the particular effects and decision table is used to summaries the actions. Here is structure of the cause effect diagram. It is also known as fish bone diagram.

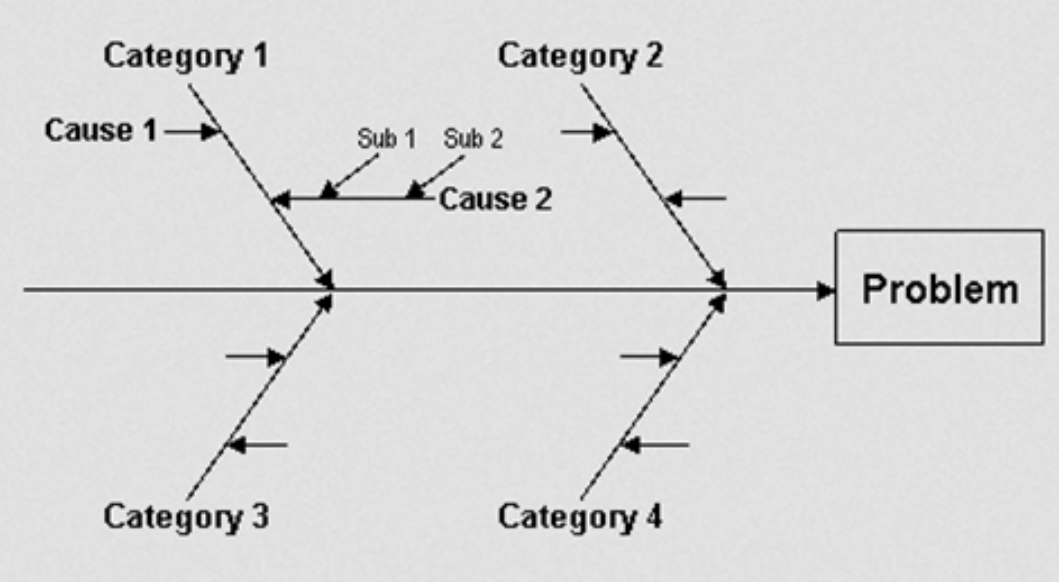


Figure 3.3: Fish Bone Diagram

2.2 Code Based Testing

A structural testing technique that uses code to generate tests. It is the form of white box testing. Code based testing technique can only be considered if tester have access to the program code. It requires the basic programming knowledge, so mostly done by the developers. It is the verification technique, and test data is selected for adequate coverage of internal structure. Level of coverage depends on the requirements. Coverage criteria can be control flow based or data flow based.

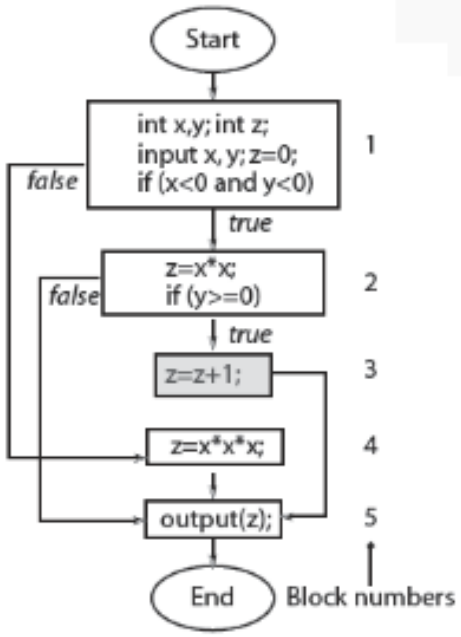
Control flow Based Coverage: Criteria for this coverage is based on the control structure of the software under test. Control flow graph shows the sequence of actions having single starting and exit point. Here are the different coverage criteria based on the control flow coverage:
Functional Coverage: Assures that each function in the code is executed
Statement Coverage: It assures that each statement in software under test is executed at least once
Condition Coverage: It assures that each conditional statement is executed once
Path Coverage: It assures that each path in the program is executed at-least once. Each condition must be evaluated both as true and false for path coverage
Combination of different coverage criteria can also be used. It is the most expensive form of testing, increasing the coverage criteria requires more tests to evaluate it.

Data Flow base Coverage: This is most widely used technique in testing. Data is center of attention for generating test in this form of testing. Analysis help to find

the flow of data and can be graphically represented i.e. Data flow diagram. Test cases are generated by selecting a path according to the operation performed on data. Def-use analysis is helpful in data flow testing. Code contains variables that are used to hold data to solve a particular problem. These variable must be defined before using it.
Consider following equation

$$a = b + c \tag{3.1}$$

In above equation a is defined, while b and c are used. *Computational use* also known as C-Use is the use of variable in equation, function call or output statement. *Predicate use* also known as P-Use is the use of variable in condition. Same as control flow graph, data flow graph is used to show the flow of define use variable in code under test. Here is the example that used three varibale x,y,z and data flow graph shows the flow of data in this code. W. Eric uses an example to illustrate the def-use from the data flow graph shown in figure 3.4.



Node (or Block)	def	c-use	p-use
1	{x, y, z}	{ }	{x, y}
2	{z}	{x}	{y}
3	{z}	{z}	{ }
4	{z}	{x}	{ }
5	{ }	{z}	{ }

Figure 3.4: Data Flow Coverage

Table in figure 3.4. shows the def, c-use and p-use of the variables use in sample code [36].
Def-Use pair shows the line numbers in the pair where a particular variable is define and used. Def-Clear path is the any path in the code starting from the node where x variable is defined and ending where x variable is used, without redefinition.
Here are the formulas for the c-use coverage, p-use coverage and all-use coverage.

$$C\_UseCoverage = \frac{CU_c}{(CU - CU_f)} \tag{3.2}$$

Where CU is total c-uses, CU_c is the total c-uses covered and CU_f is the total c-uses not feasible.

$$P\_UseCoverage = \frac{PU_c}{(PU - PU_f)} \tag{3.3}$$

Where PU is total p-uses, PU_c is the total p-uses covered and PU_f is the total p-uses not feasible.

$$All\_UseCoverage = \frac{(CU_c + PU_c)}{((CU + PU) - (CU_f + PU_f)} \tag{3.4}$$

All-Use coverage is required to be 1 for adequate test.

2.3 Fault Based Testing

Fault base testing techniques are not using code or specification for testing. Instead it focus on the assumptions that errors are present in the code and generate similar faulty programs to locate these error. Fault based testing determines pre specified faults in code under test .Morell (1990). Scope of fault based testing is to generate the alternate programs and distinguish it with others. FBT provides evidence that, code does not contain any specified faults. Mutation testing is on the fault based testing techniques.

Fault Based Symbolic Execution: Symbolic execution analyse the program to find if full coverage is obtained. Instead of actual input values, symbolic values are used for program execution. Program reshaped containing symbols and different constrain solver are used to find the actual values of the symbols. It is very difficult to implement symbolic execution on large program due to path explosion. Some approaches uses heuristics to find optimal path to avoid this problem Ma et al. (2011).

```
x=input();
x=x*7;
if(x >10)
    print("grater");
else
    print("less");
```

In Symbolic execution, instead of providing input values random symbols are used. Consider above code example, in which value of x is read from user. Instead of providing actual number we will assign a symbol s to x . In next line as x is being multiplied by 7, so x become 7*s. Soling constrain depends upon the goal of testing. For example in this case our goal is to print "grater", so 7*s must be greater than 10. Next constraint solver finds the values of the symbols based on path constraint.

Mutation Testing: Involves modifying the program and evaluate the tests. Mutation testing uses mutation analysis to evaluate tests or generate new test cases .Ammann and Offutt (2008). Mutants are the modified programs. Mutation testing accepts the

mutants and test data. This test data is used to execute the newly created mutants. Mutant is said to be killed, if test cases differentiate them from original program. Mutant operators are used for atomically generation of mutants.
Some of the mutant operators are as follows:

- Constant alteration
- Scalar variable alteration
- Arithmetic operator alteration
- Return statement alteration
- Statement deletion
- Relational operator replacement
- Change order of parameter in method call

Different open source tools are available for mutation testing, it is useful on unit testing level.

3 Object Oriented Testing

Testing object oriented program is complex task. Though traditional testing techniques can be used to test OO program but new concerns require additional techniques to test OO software. OOP involves classes and objects. Objects are instances of classes. In object oriented testing, we not only focus on input/out values but the initial and final state of the object is also important. Class contains two parts, member variable and member functions. These member variables cannot accessed directly, functions are used to perform operations. It is very difficult to examine what is happening internally due to encapsulation and information hiding. So the state of the object cannot be directly examined.

3.1 Object Oriented Unit testing

Unit testing ensures that each unit in system works perfectly in isolation before integration began. A class is the unit of object oriented program. As compared to procedural language testing, class testing is complex task. Unit testing of object oriented program is not simple as running the code and providing inputs from console etc. instead it involves writing a piece of code that call the method of class under test and checks it behaviour based on some assumptions. These assumptions can be coverage goal or any other criteria used for testing.
Main issues in unit testing of object oriented program are:

- Difficulty in finding which class object to test
- Deciding where to stop, what is the enough criteria for testing
- No of methods to be tested in specific class or unit

There are many approach available that deals with the unit testing of object oriented program .Fraser and Arcuri (2011b), .Cheon et al. (2005), .Xie et al. (2004), .d'Amorim et al. (2006).

3.2 Object Oriented Integration testing

Goal of integration testing to test if units are properly working together. It is the inter-class testing technique. It ensures that communication through interfaces is done properly. They are three approach used in integration. In Big Bang Approach, all unit are coupled together at a time. In this Aapproach, it is very difficult to maintain the test and its results. Second is the Bottom Up approach, in which lower level components are tested first. This is recursive process and continues until all objects are integrated and tested together. Third is the Top Down Approach. First top level modules are integrated together.
Issues in integration testing are as follows:

- Finding units that are involved in integration
- Method sequence generation for high coverage of inter-class testing
- Generating desired object state for integration testing
- Generation of test data to run these tests

Units working properly in isolation may cause problem in integration. Goal of integration testing to test the interfaces. manual testing is a tedious job. Automated approaches are used instead. Generating test data is the main aim of automated techniques. Only less work done in test generation of object oriented program due to complex nature and method explosion Bashir and Nadeem (2009).

4 Search Based Software Testing

Search based software testing involves automation of testing process using meta-heuristic optimization techniques McMinn (2011). Automated test data is generated using search based techniques. ant colony, evolutionary computation, and genetic algorithms are the common examples of optimization techniques. Roots of search based software testing traces back to 1976 when *Miller and Spooner* first proposed floating point test data generation approach based on cost function Miller and Spooner (1976). Optimization algorithms are categorized as derivative or derivative free algorithms. Hill climbing is the example of derivative based algorithms, is used gradient information for optimization. But derivative free algorithms uses direct value of the fitness function. Optimization algorithm can be deterministic or stochastic. Deterministic algorithms does not use any randomness in finding optimal solution. So if we start at same initial point we will find the exact one solution each time. Hill-climbing and downhill are examples of deterministic optimization algorithms. On the other hand stochastic algorithms adds randomness and using same initial point we will at different points. Genetic algorithm and PSO are the good example of this type of algorithms.
Goal of most of the metaheuristics is as follows:

- These techniques guide the search process
- Explores the search space to find the optimal solution
- These algorithms are not problem specific and mostly based on approximation

- A fitness function is used to evaluate the solution

Fitness function is the objective function, used to assess the solutions. Fitness function is problem oriented and efficiency of algorithm depends on it. Sometimes is not easy to find a fitness function that fits to our situation. Good fitness function leads to good solution.
One of the simplest form of optimization algorithm is random search.

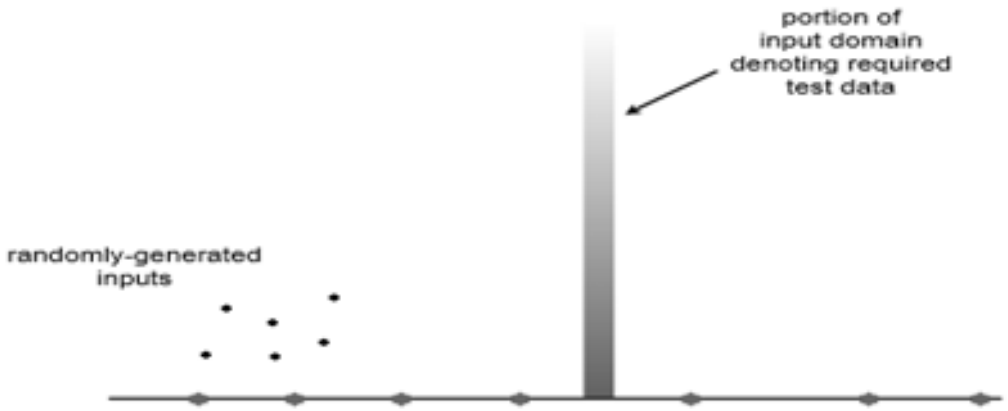


Figure 3.5: Random Search

But is very difficult to reach at required solution with random search. There is a need a little guidance to find the optimal solution. Figure 3.5 shows the random point selection in input domain. There is very less probability that points are selected as required. Solution is to use the fitness function. These fitness functions are reliable in finding the optimal solution and being used in most of optimization algorithms. Genetic Algorithm is found to be more effective in test data generation Ciupa and Leitner (2005), Khan and Nadeem (2013), Bashir and Nadeem (2009), Pachauri and Srivastava (2013), Hermadi et al. (2003).

5 Genetic Algorithm

Genetic algorithms is one the heuristic algorithm with adaptive nature. GA uses the Charles Darwin theory of evaluation to find best solution. Competition in individuals for the resources results in fittest solutions to survive. It is based on natural genetics to solve optimization problem. An optimization problem can be define as finding best solution from different available solutions.
Implementation of genetic algorithm needs:

- Genetic representation of solution
- Fitness function

Representation of genetic solution is problem specific. Each problem requires different solution representation. Fitness function is used to evaluate the individuals. It is also known objective function.
Following steps involves in genetic algorithm:

- Each individuals fitness is calculated
- Individuals with the more fitness value are considered successful
- Successful individuals are then used to produce more offspring for next generation
- Two good parents produces the child solution that better than its parents

Fitness Function: Fitness function is also known as objective function. Efficiency of genetic algorithm purely depends on fitness function design. A good fitness function leads to discovery of fittest solution.
Here are some examples of bench mark fitness function.

$$f(x) = sin(x) \tag{3.5}$$

$$f(y) = cos(y) \tag{3.6}$$

$$f(a + iy) = \sum_{k=1}^{\infty} \frac{1}{k^(k + iy)} \tag{3.7}$$

Individual is a candidate solution, to apply fitness function. Value calculation of fitness function depends on the dynamics and structure of the variable.

$$f(x_1, x_{2,3}) = (2x_1 + 1)^2 + (3x_2 + 4)^2 + (x_3 - 2)^2 \tag{3.8}$$

To evaluate the optimization function in equation 3.8, individual provides the values for *x1*, *x2* and *x3*. Individual can be represented in the form of vector e.g. (2, 3, 4). Individuals are alternatively known as genes.

Population is the collection of individuals. Size of population is fixed throughout the evaluation. Genetic algorithm is recursive process and on each iteration, some operation are performed to produce new population.

Diversity is the property of genetic algorithm, it represents the average distance of individuals in population. High diversity is achieved if average distance between individuals is high. High diversity is required for genetic algorithm to explore the large search space, because low diversity only explores the small region (shown in figure 3.6).

Population contains n solutions. Initial population is generated randomly. Fitness for each individual is calculated to select the best parents for next generation. Genetic algorithm use two operator to find optimal solution. Individuals are replaced with the new solutions hence form new population. New solution on average have more good solutions as compared to predecessor solution.
GA uses following three operators to complete tasks.

- Selection
- Crossover
- Mutation

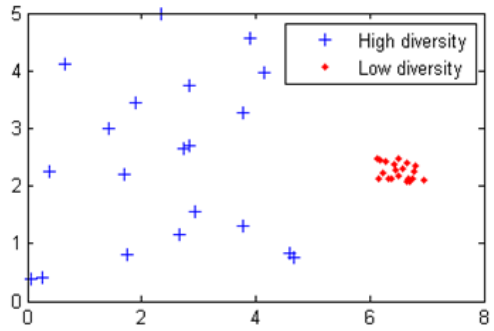


Figure 3.6: Diversity

Selection is the process of selecting best individuals for production of next generation. It is done by calculating the fitness of each individual based on objective function. Common selection techniques includes roulette-wheel selection, tournament selection and truncation selection.
Crossover represents the mating technique for parents. Two individuals are chosen from one of the selection technique. Crossover techniques helps in transferring properties of both individuals to next off-spring. One point cross-over, two pint crossover, cut and splice, uniform and half uniform crossover are commonly used techniques.

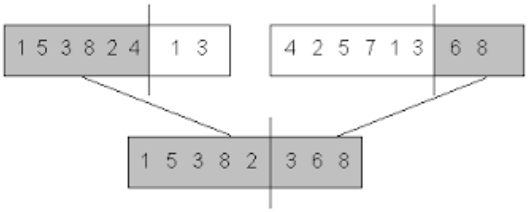


Figure 3.7: Single Point Crossover

Mutation operator is used to maintain the diversity. Mutation change the individual by flipping the bits at the given rate. Mutated individual is entirely different from the original one. Mutation rate is set to low because GA with high mutation rate loses its advantage and turns into random search. Different mutation operators are used depending on the gene types like uniform, non-uniform, boundary or Gaussian etc.

Chapter 4

Literature Review

This chapter provides brief literature review on testing of object oriented programs.

1 Test Case generation

Generating test cases is the tricky part and efficiency of test purely based on the selected test cases. This is the centre of attention for researchers from last few decades. Test case generation techniques can be categorized in four basic approaches .Anand et al. (2013). Model based testing, structural testing using symbolic execution, random testing and search based testing. Symbolic execution was first used in seventies. This technique can be applied to complex real world problems. Dynamic symbolic execution is widely used in static and dynamic code analysis leading to test data generation. Xie et. all proposed the framework for generating object oriented tests using symbolic execution .Kao et al. (1999). Symbolic execution is the expensive type of program analysis. Parameters with unspecified arguments are passed, constraints are built on that parameters and actual inputs are used in solving these constraints.
Seeker is the state of the art approach proposed in 2011 .Ciupa et al. (2006). It uses static and dynamic analysis to generate the test sequence based on user intend. Results showed relatively high branch coverage. Randoop is a random test generation approach. Tests are generated by randomly selecting method calls. Test data is also generated randomly. This data and previously generated sequences are used for primitive and non-primitive arguments, respectively.
The limited computational capability of older generation computers made it impossible to symbolically execute large programs. Tool is proposed that uses the whole test suit generation approach to generate tests for the entire testing criteria Fraser and Arcuri (2011a). It uses mutation based assertion tests to maximize the seeded defects. It helps to identify more defects. It uses population of candidate solution for variable number of test cases. Test case is the sequence of method calls. Most common approach is to select one coverage criteria at a time like statement coverage or branch coverage etc, Tonella (2004a), Harman and McMinn (2010) uses this approach. All coverage criteria are equally important but some are difficult to reach. Whole test suit uses optimization approach to generate test cases for whole criteria rather than the individual goal. Whole test suit generation uses the search based approach to generate test. Crossover and mutation are used for generation of candidate solutions and fitness function is used to find the optimal

solution. Crossover is used to produce a new population based on parents and it uses random selection for crossover position. Harman et al. (2010) propose approach for search based test data generation pointing its significant to the oracle problem, which is the most common problem in evaluating the results. This process continues until required solution is found. In this case candidate solution are the variable number of method calls. Test data is generated based on fitness function evaluation. GA is used for automated test generation in this approach. Firstly random test cases are defined and then crossover and mutation operators are used to optimize the solution. Crossover is used to generate child solution from two or more parents and mutation is to inforce the diversity in the solution by altering one or more gene values in candidate solution. Mutation probability is used as 1/3 in this approach, because in mutation three operations are performed remove, insert and change Fraser and Arcuri (2011b). This technique lacks other coverage criteria, it only focus on branch coverage goal. This technique is used for single class automated testing. However complex object oriented concepts like inheritance, composition are not focused in this paper.
Jin and Offutt proposed system for integration testing Jin and Offutt (1998). This approach was purely for procedural language. Coupling relationships are describes in following categories.

- Parameters coupling
- Shared data coupling
- External device coupling

This approach is extended by Alexender and Offutt for object oriented program Alexander et al. (2000). Four types of coupling are identified for object oriented programs. This technique further used by Shoukat et. all .Khan and Nadeem (2013) for test data generation of object oriented program. Generating the test sequence for object oriented program is challenges task. GA is used for test data generation in this approach but test cases (test sequences) are accepted as input. Random approaches for test sequence generation is RecGen .Maher (2004). RecGen generates test randomly. Object-field-access information is consider for sequence generation of methods in the application.

Denaro et al. (2015) Proposed DynaFlow approach for test case generation of object oriented programs. It concentrate on dynamic properties of object oriented program. This technique generates inter-procedural test case used from single class. Test cases are generated analysing method interactions. Initial test suit is generated and iteratively improves test cases. In each iteration, new test objectives are derived based on execution traces to cover them. results shows that enhanced test cases found to be more effective then classic testing. Failures are detected based on more interesting states and interactions.

2 Test Data generation

Existing test data generation techniques are classified into three categories.

- Path-wise test data generation

- Data specification based
- Random test data generation

In this paper .Ciupa and Leitner (2005) authors proposed GA based approach for the test data generation. Its kind of path testing approach. Manually generation of test data is a challenges task and authors take advantage of GA to automate it.
Basic path testing is one of the structural testing approach McCabe (1976). In this approach, it is ensured that each path is executed at least once. Cyclomatic complexity measurement is used to find the complexity of the algorithm. It tells how many tests are required to test it. It uses the central flow graph to find the covering criteria.
Cyclometric complexity is calculated using following formula:

$$V(G) = e - n + 2 \tag{4.1}$$

V (g) is the minimum number of test cases required to perform testing. GA is used for optimization solution of large complex problem.
Crossover and mutation is used by GA to provide optimal solution. Crossover is used to select the parents that would be used to produce the next generation. It uses random selection point for couple selection .Lin and Yeh (2001). Random generation of crossover point for each genes, exchange first part of each gene and add them for the production of next generation.
Mutation probability is used to state of random selected gene. It prevents GA to generate the local extremes. Authors uses bogdam koral branch distance function to generate fitness function. Koral expressed the branch predicate on the form of relational expression .Korel (1990). They propose a method to represent each predicate in the form of branch function and relation.

Branch Predicate	Branch Function F	relation
$E_1 > E_2$	$E_2 - E_1$	$<$
$E_1 \geq E_2$	$E_2 - E_1$	$\leq$
$E_1 < E_2$	$E_1 - E_2$	$<$
$E_1 \leq E_2$	$E_1 - E_2$	$\leq$
$E_1 = E_2$	$abc\ (E_1 - E_2)$	$=$
$E_1 \neq E_2$	$abc\ E_1 - E_2)$	$\leq$

Table 4.1: Distance Details

This approach evaluate the branch function evaluating the branch expression like a-b, a >b or a <b etc.
Initial population is generated randomly and each chromosome is evaluated based of the fitness function to find the optimal test data.
Fitness function in this approach is designed based on traversal of predicate notes. They uses the predicate evaluation to express the fitness function.

$$f = \frac{1}{((abs(A - B) + 0.5)^2)} \tag{4.2}$$

To avoid the result in infinity a small data values is added in this equation. This technique is evaluating the single branch basic path testing using GA. Results are impressive 38% of the test data has the higher fitness value in the range of 1 to 0.7. Results shows that GA is more effective in test data generation as compared to random data generation. Testing object oriented means, testing sequence of method calls on a particular object. It requires both the target object and the arguments required for the routines. But most important is the value of the arguments to complete the test.

- Autotest generates objects and stores it in object pool
- Decide if we need to create an object if I is not the object pool

Object creation is done by following procedure

- Choose one of the creation procedure
- Choose arguments values
- Call routines on these values

Arguments can be object or primitive types like integer, float, And boolean etc. Author suggests to use random values for the simple types. Using this approach speed up the execution and decrease the baisness. But Korat directly set the filed without using above procedure Boyapati et al. (2002). Providing direct values skip the fundamental concepts of creating objects at this point. Maher (2004) proposed adaptive random testing. In this techniques they improve the random selection criteria by spreading out the selected values over corresponding interval. This approach cannot mapped to user defined classes and only works for primitive data types. To counter this issue and apply this approach to the objects, authors added randomness in object creation and approach is called object distance .Ciupa et al. (2006). It is used in object selection based on how far one object is from another based on object distance.
Distance can be given as weighted sum of the following factors Distance: distance b/w their types, distance between immediate vales, fields involving references to other objects. Test oracles are used to validate the testing procedure. In this approach pre, post and invariant conditions are to validate the results Meyer et al. (2007). This approach is optimized by adding partitioning concept in selecting object states. Objects are partitioned into different disjoint spaces based on the states and allows automated test to pick the objects from different spaces adding diversity in to it. Object states can be identified by using argument-less queries. Argument-less queries return true or false based on the criteria of that query and these are useful in selecting the object states. Autotest uses the forward exploration procedure to identify the new abstract states that are useful in selecting object states. Despite Auto run involves manual inspection of results, this approach showed better result over using simple random testing.
Arcuri and Fraser introduced novel approach to generate test suit satisfying coverage criteria .Fraser and Arcuri (2011b). This technique asserts small set of assertions that summarize the current behaviour and can be used for evaluation. According to authors test cases can be generated automatically but main issue is the test oracle, how to validate the results. This tool uses search based technique with following approaches.

- Hybrid search .Harman and McMinn (2010)
- Dynamic symbolic execution .Whittaker et al. (2000)
- Testability transformation .Ramler and Wolfmaier (2006)

Common approaches for test data generation focus on single coverage criteria. Commonly used techniques are dynamic symbolic execution and search based techniques [5].

2.1 Dynamic symbolic execution

Dynamic symbolic execution is widely used in static and dynamic code analysis leading to test data generation. Instead of actual inputs, symbols are used identify the valid inputs for test data to obtain required coverage. Xie proposed the framework for generating object oriented tests using symbolic execution .Kao et al. (1999). Botella et. all used the dynamic symbolic approach to detect the infeasible path while generating test cases. It saves lot of time and effort invested in the data generation for infeasible paths .Delahaye et al. (2015). Another approach used for redundant state detection using symbolic execution .Bugrara and Engler (2013). This system is evaluated on 66 bench marks and results showed significant improvement in both coverage and fault state detection. .Jamrozik et al. (2012), .Cadar et al. (2011), .Xie et al. (2009), .Saxena et al. (2010) uses the some variations of dynamic symbolic execution and produced better results in software testing domain.

3 Search Based Techniques

Search based software testing is rooted back in 1976. Use of metaheuristic in software testing is commonly known as Search based software testing. SBSE can be applied to both black box and white box optimization problems. This technique is effective where space is too large to find optimal solution. Fitness function guides the search to obtain the best results. SBSE is applied successfully in software testing.
Tool uses the whole test suit generation approach to generate tests for the entire criteria .Fraser and Arcuri (2011a). It uses mutation based assertion tests to maximize the seeded defects. It helps to identify more defects. It uses population of candidate solution for variable number of test cases. Test case is the sequence of method calls. Most common approach is to select one coverage criteria at a time like statement coverage or branch coverage etc. Harman and McMinn (2010), Tonella (2004b) uses this approach. All coverage criteria are equally important but some are difficult to reach. Whole test suit uses optimization approach to generate test cases for whole criterias rather than the individual goal. Whole test suit generation uses the search based approach to generate test. Crossover and mutation are used for generation of candidate solutions and fitness function is used to find the optimal solution. Crossover is used to produce a new population based on parents and it uses random selection for crossover position. Harman .Harman et al. (2010) proposed approach for search based test data generation pointing its significant to the oracle problem, which is the most common problem in evaluating the results. This process continues until required solution is found. In this case candidate

solution are the variable number of method calls. Test data is generated based on fitness function evaluation.

Fitness function: There are different approaches present for fitness function evaluation. Author presented the branch coverage criteria generalized to remaining criteria. Branches are expressed as control structures like if or looping conditions etc. Branch coverage is satisfied if all branches are evaluated to true and false as well. Optimal solution covers all the branches and its statements in given code.

$$fitness(t) = |M| - |M_t| + \sum_{B_k \subset B} d(b_k, t) \tag{4.3}$$

Where M is the method set and M_t is the set of method executed and d represents the branch distance. This fitness is used for selection of parents for off springs. Using branch distance for test data generation is very common approach .McMinn (2004). Branch distance is calculated by subtracting the actual and the required values for a variable to evaluate a branch true or false. Bloat may occurs when very small improvement in fitness value is made with very large and difficult solution.
GA is used for automated test generation in this purpose. Firstly random test cases are defined and then crossover and mutation operators are used to optimize the solution.
Crossover is used to generate child solution from two or more parents and mutation is to enforce the diversity in the solution by altering one or more values in gene values in candidate solution. Authors used mutation probability as 1/3 in this approach because in mutation as there operations are performed in mutation remove, insert and change .Fraser and Arcuri (2011a).
This technique lacks other coverage criteria, it only focus on branch coverage goal. Authors prove this technique for single class automated testing. However complex object oriented concepts like inheritance, composition are not focused in this paper.

Another approach for test data generation is presented by using Boolean queries and contracts .Liu et al. (2007). In object oriented testing, difficulty is to select the interesting object that identifies bugs. If the testing units are classes then Boolean queries are helpful in finding solution. It helps in selecting the object states and identifying test cases. Contracts are the preconditions, post conditions and invariants. In this approach class properties are selected based on Boolean queries and contracts. This approach is based on black box testing, no need for implementation details. Argument-less queries are used for identifying the object states. Object state machine is formed to record all the states and transitions to identify the unexpected behaviour of the class under test. It is also used to evaluate the completeness of the test suit. Acquiring the all reachable states is challenging task. Boolean constrains solver is used in this approach. SCIStus solver is used to extract all the object states. Next step is the pruning of these states. Simplify tool is used for this purpose to neglect the states that does not make sense. With the help of this tool states that are not reachable are neglected from the state set. Boolean query coverage criteria is used for testing, which implies that this test will cover all the reachable states identified. After selection of the test states, authors used the AutoTest .Ciupa and Leitner (2005) approach for testing the object oriented code. Autotest is the tool, in which test cases are generated by calling methods and test oracles are selected from the invariant and post conditions of the contracts. If preconditions are satisfied but post conditions are violated then it is the indication of the bug and marked as buggy

routine and it is the output of test. Main significance of this approach is to identify the buggy routines based on the invariant. Manual states selection is used if some states are not reachable.
Integration testing tests the interfaces of different components. One approach for coupling based integration testing is proposed by Shaukat and Aamer .Khan and Nadeem (2013). GA is used for test data generation in this approach. Unit testing is used prior to the integration testing. When unit works properly their interfaces are tested integration testing.
Evolutionary approach in test data generation is known as evolutionary testing. It includes algorithms like GA, PSO, ant-colony etc. Evolutionary algorithms are simulated evaluation used to evaluate the candidate solutions. Candidates are assigned fitness values based on user defined evaluation criteria and function. Multiple parents are selected to be parent based on fitness. In GA crossover and mutation operators used to create the next level offsprings which are again assigned fitness values using same criteria. It overwrites the weak off springs and continue this process till the user defined criteria.
This technique is based on Jin and offutts approach for integration testing .Jin and Offutt (1998). This approach was purely for procedural language. Coupling relationships are describes in following categories.

- Parameters coupling
- Shared data coupling
- External device coupling

This approach is extended by Alexender and Offutt for object oriented program .Alexander et al. (2000). Four types of coupling are identified for object oriented program. This technique further used by shoukat et all for test data generation of object oriented program.
In integration testing, multiple test paths are identified. Mapping table for actual and formal parameters are used facilitate the def-use analysis. In this approach coupling path is used as input to generate test data for the given coupling path. Antecedent is the method that defines the variable, and consequent is the method that uses this variable .Khan and Nadeem (2013). Here coupling sequence is the path from antecedent node to consequent node.
This approach is divided into phases.

- Accepts the test cases as input
- Generation of test data based on GA

In this approach they uses coupling variables and fitness function for integration testing of object oriented program. Based on the coupling path fitness function is evaluated for optimized test generation. Limitation to this approach is that user needs to provide the test cases manually. These paths are used for automated test data generation. They use fitness function proposed by Tracey et all .Tracey et al. (2000). Fitness values are calculated based on branch predicates. These fitness values are used by GA to generate optimal test data.

As we discussed in evolutionary testing, result are purely based on fitness function. Fitness function pays vital role in effectiveness of the evolutionary testing. Bilal et all. Proposed a fitness function for object oriented program considering object state as well as coverage criteria Bashir and Nadeem (2009). Their work is based on .Baresel et al. (2002) work, fitness function is given as follows.

$$fitness(t) = la\_approx\_level + (1 - la_branch_dist) \tag{4.4}$$

$$approx_level = dependent - executed \tag{4.5}$$

$$la_branch_dist = (\sum_{i=1}^{p} (m\_branch\_dist)_i)/p \tag{4.6}$$

Sate variables are not managed separately in this approach. State problem may arise if state condition is dependent on same state variable. To avoid state problem author proposed a methodology of keeping the objects state separate from coverage fitness. Object state will help in finding if mutation operation is required.
So fitness is described as:

$$fitness(t) = (State_fitness, Coverage_fitness) \tag{4.7}$$

And state distance is *1- la_bracnch distane.*
State fitness is calculated by adding the branch distance of predicate that depends on this state variable and dividing it with total number of branch predicates.

$$coverage = la_(approx_level) + (1 - branch_distance) \tag{4.8}$$

Branch distance of state variables is excluded from coverage fitness. Fitness zero means no mutation is required and non-zero means mutation is required in this part Baresel et al. (2002). Issue may arise when more than one classes are used then state fitness of each object is calculated separately.
This fitness calculation helps to guide the evolutionary algorithm in finding better tests and it also helps to solve object state problem. In evolutionary testing sequence of the method calls to produce objects to be tested is challenging task. In evolutionary testing of OO program the sequence of method calls is generated to produce objects for test. Automated integration testing is still active research topic especially in object oriented domain.
Here is the list of techniques using GA with parameters used Sharma et al. (2014).

Author	Testing Technique	Crossover Rate	Mutation Rate	Crossover method	Selection Method	Mutation Method	No of Generations
D. J Berndt, 2005	High Volume Testing	--	--	--	--	-	500
Doungsa	Gery-box testing	0.4	0.3	Single one point	Random	Random	2
Dr. Velur Rajappa, 2008	System Testing	0.5	0.5	Single point	Tournament selection	--	--
Francisca Emanuelle, 2006	Functional Testing	0.8	0.01	Point Crossover	random	flip	50
Jose Carlos, 2008	Unit testing	0.1, 0.8, 0.33	0.1, 0.8, 0.34	Random point	Tournament selection	> 7.5 average generations (Combination of C.R and M. R and r =[0.1, 0.8, 0.33]	--
Maha Alzabidi, 2009	Path testing	1.0	0.005	Double point	Random and roulette wheel	Flip	50
Mark Last, 2006	Black Box	Adaptive	0.01	One point	random	Flip	200
R. Krishnamoorthy , 2009	Regression testing	for r = [0,1], r < user value	1	Random point	Roulette Wheel	Random	25
Robert M. Patton, 2003	Usage testing	--	--	One point	Fitness proportionate	Random One - point	30
Ruilian Zhao , 2008	Black Box	0.8	0.15	--	Roulette Wheel	--	>500
Stefan Wappler, 2006	Unit Testing	--	--	Point crossover	Tournament Selection	Point mutation & Real Mutation	<10
P. R Srivastava, 2009	Path Testing	For r = [0,1], r < 0.8	For r = [0, 1], r < 0.3	Pairwise	Random	Flip	3

Table 4.2: GA Parameters Used in Testing

Chapter 5

Methodology

Only small work is done in the integration testing of object oriented program. Saukat Khan et all proposed coupling based approach for test data generation and used cost function for fitness evaluation .Khan and Nadeem (2013). But there is still need to input the coupling path to generate the tests.
Proposed technique is divided into two categories. Automated generation of test cases leading to test data generation. This research is conducted in three steps to achieve

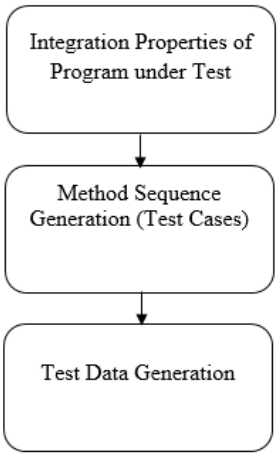


Figure 5.1: Research Focus

the stated goal. First step involves understanding and tailoring integration properties for further use. Second steps involves test case generation and finally research focus is towards test data generation.
Units must interact properly for efficient working of the system. Integration testing, tests the interfaces to ensure the functionality. In object oriented software classes are considered as unit. Path explosion make it difficult to test inter class functionality. This problem is tackled by considering the coupling relation between classes.

Proposed steps for integration testing are as follows:

- Finding units that are involved in integration

- Method sequence generation for high coverage of inter-class testing
- Generating desired object states for integration testing
- Generation of test data to run these tests

1 Automated Test Case Generation

We used coupling based information in method selection for automated test case generation.

There are four types of coupling identified by Jin and Offutt (1998).

- Parameter coupling
- Shared data coupling
- Global coupling
- External device coupling

Parameter coupling: This type of coupling occurs when method of one class passes the object of another class as arguments.

Shared data coupling: This type of coupling relation exists when multiple class objects are working on the same shared data.

Global Coupling: When A and B both classes share the same global reference.

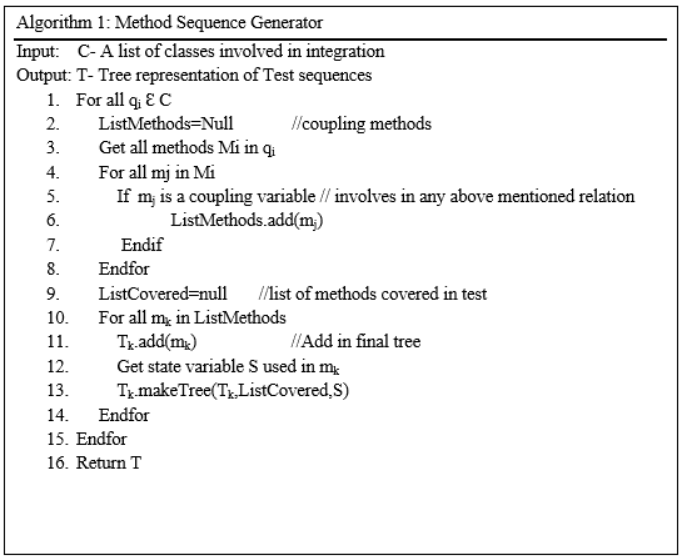
External Device Coupling: When both units or classes are dependent on same external device.

We are neglecting the External Device Coupling at the moment and using other three type of coupling for test sequence generation of object oriented integration testing.

Following situation can arise while looking at the classes in coupling point of view.

Argument based: when another class object is passed as arguments
Composition based: can be local (within function) or class level

Proposed algorithm uses the coupling relations to generate the variable length method sequence to test the integration of different components.



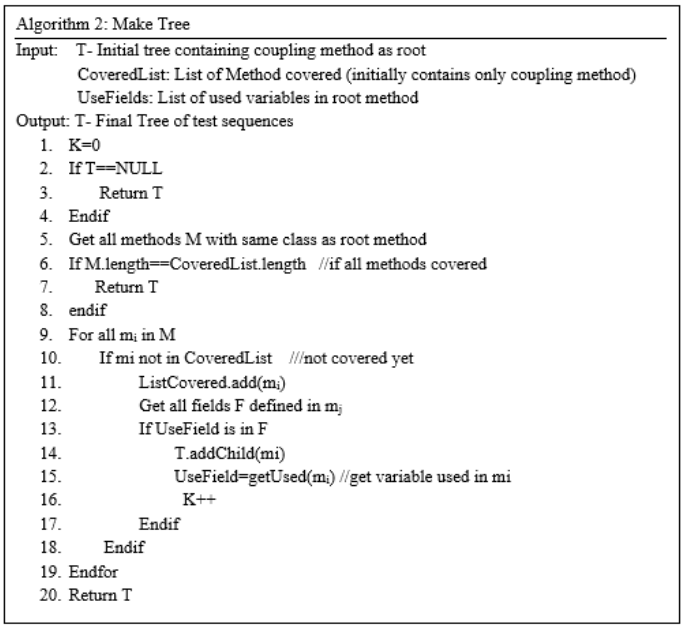
Algorithm 1: Method Sequence Generator

Input: C- A list of classes involved in integration
Output: T- Tree representation of Test sequences

```
1.  For all qi ε C
2.      ListMethods=Null          //coupling methods
3.      Get all methods Mi in qi
4.      For all mj in Mi
5.        If  mj is a coupling variable // involves in any above mentioned relation
6.              ListMethods.add(mj)
7.         Endif
8.      Endfor
9.      ListCovered=null     //list of methods covered in test
10.     For all mk in ListMethods
11.       Tk.add(mk)                //Add in final tree
12.       Get state variable S used in mk
13.       Tk.makeTree(Tk,ListCovered,S)
14.     Endfor
15. Endfor
16. Return T
```

Figure 5.2: Algorithm for Method Sequence Generation

We use data flow coverage criteria. Each DU path is considered in selecting method sequence to generate the desired state for integration testing.



Algorithm 2: Make Tree

Input: T- Initial tree containing coupling method as root
CoveredList: List of Method covered (initially contains only coupling method)
UseFields: List of used variables in root method
Output: T- Final Tree of test sequences

```
1.  K=0
2.  If T==NULL
3.      Return T
4.  Endif
5.  Get all methods M with same class as root method
6.  If M.length==CoveredList.length  //if all methods covered
7.     Return T
8.  endif
9.  For all mi in M
10.     If mi not in CoveredList   ///not covered yet
11.           ListCovered.add(mi)
12.           Get all fields F defined in mj
13.           If UseField is in F
14.                 T.addChild(mi)
15.                 UseField=getUsed(mi) //get variable used in mi
16.                  K++
17.           Endif
18.      Endif
19. Endfor
20. Return T
```

Figure 5.3: Algorithm for Tree Generation

Instead to generating the method sequence for branch coverage we use coupling based data flow coverage for test case generation. It generates a Tree that contains coupling method as root and all the possible def-use paths can be represented by the sub nodes of that tree.
Our algorithm assumes that each unit is already tested and validated. It only requires to test the interfaces, through which units are interacting with each other. Abstract level steps for generating method sequence are as follows (Details are described in algorithm

5.2 and 5.3).

1. Accepts the classes involve in integration
2. Parse the source code and identify the coupling methods if a method is involved in integration (As referred to coupling relation)
3. Main focus is to test these methods. But objects must be in the desired state for proper integration testing
4. Our approach involves DU-analysis to get the predecessor methods to generate the desired states.
5. We add coupling method as root of tree and identify the list of state variables being used in coupling methods
6. Methods using the state variables (that are defined in coupling method) are child nodes in test tree
7. Each child node is processed for DU-analysis to find sub nodes for each child that contains predecessor method (recursively)

This is recursive process and output is the tree, representing method sequences with coupling method as root. integration testing.

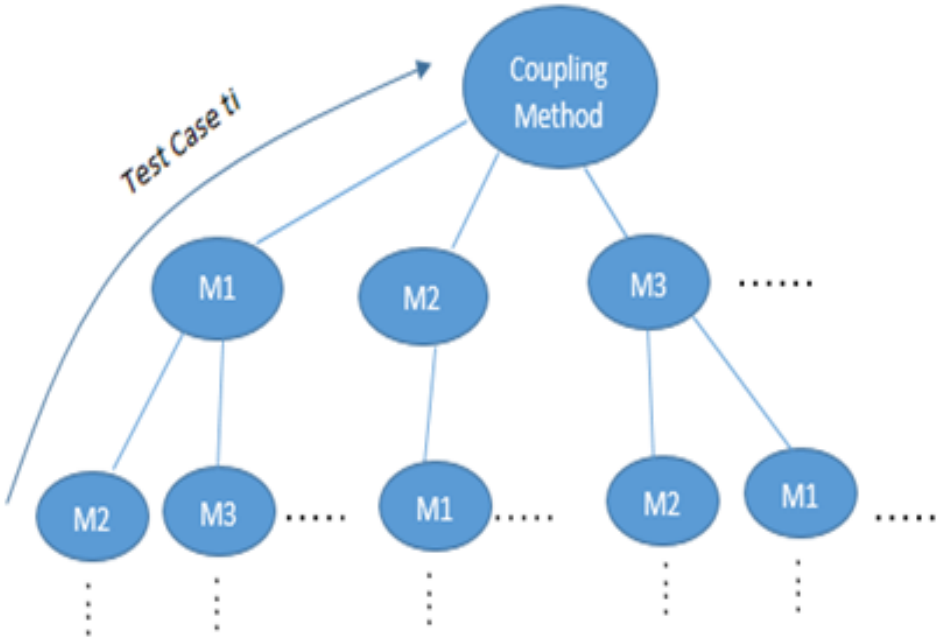


Figure 5.4: Coupling Tree

Tree representation is the output of our test sequence generation algorithm. There are variable number of methods (nodes) involved in each test case. Root node represents the coupling method that involved in integration and child nodes are predecessor methods selected based on DU analysis.
Problem may occurs when SUT contains two classes A and B, suppose method a1 of class A calls method b1 of class B and this b1 method calls the a1 in return. This scenario is referred to as cyclic loop.

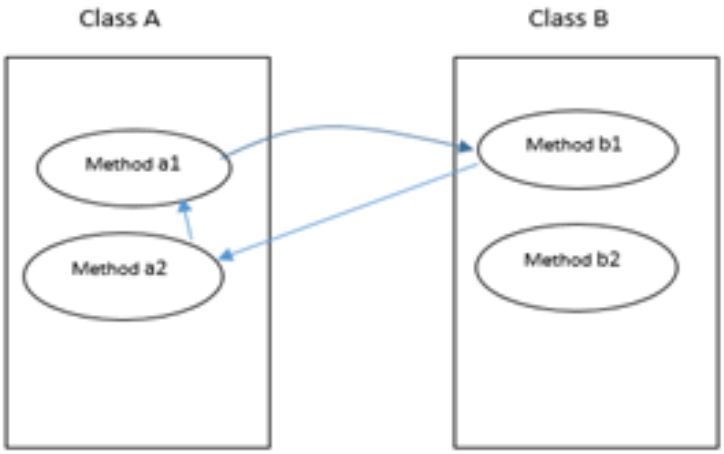


Figure 5.5: Cyclic Loop

To avoid this cyclic method calls we use weighted information with each method call. Algorithm contains *coveredMethod* list which insures that no method is selected twice in any test case. While selecting a method we increase the weight of it, knowing that this method is now selected. Any method with weight more than one is the alarming condition that this method can be a part of cyclic loop. These weights are further analysed to omit cyclic loops from test cases.

To generate the test cases we need to traverse the tree in the reverse order. Following steps involve in printing method sequence from test tree.

i Get all the leave methods (methods on leave nodes)

ii For each leave node

Select leave node

Print the method name on current node

Select parent node

Repeat steps b to c until node == NULL

iii Endfor

This sequence does not include constructor, we need to further analyse and add the object declaration for constructing the required objects to complete the test.

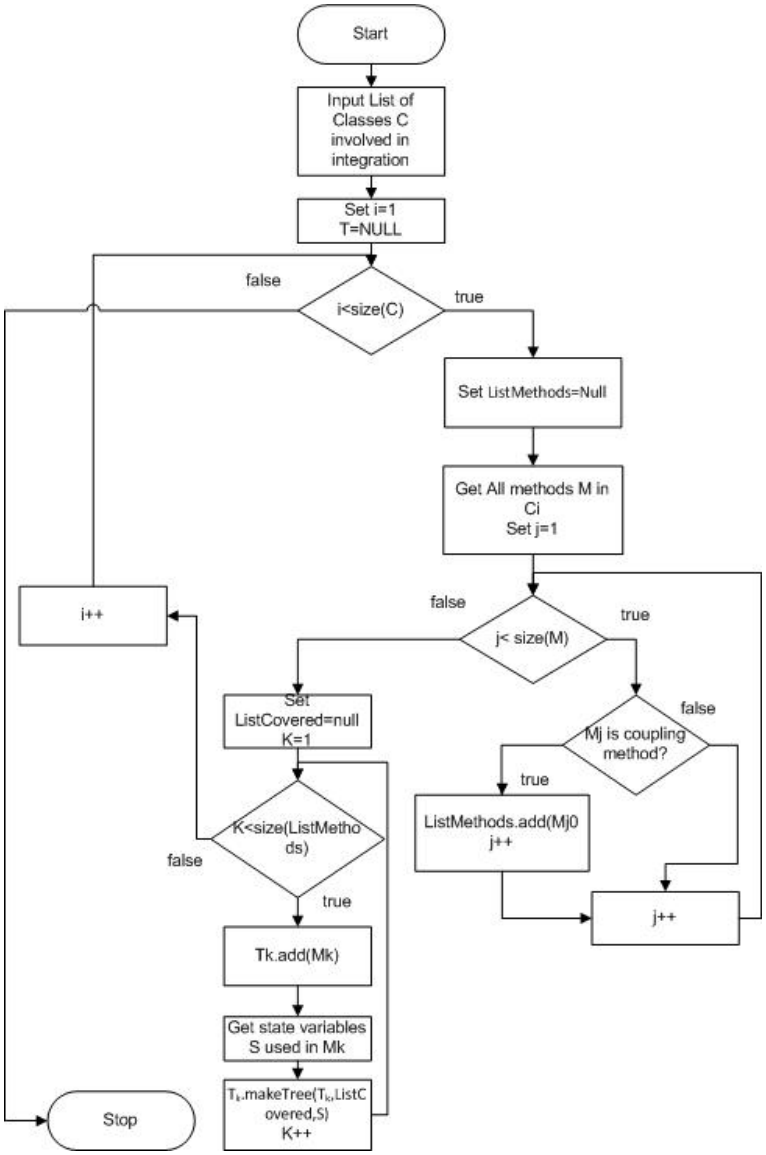


Figure 5.6: Flow Chart (Test Case generation)

Once we have enough integration test cases (method sequences) to test the object oriented program, there is a need for input data to test it.

2 Test Data Generation

Test data generation in program testing, is the process of identifying a set of test data, which satisfies the given testing criterion. Up till now most of the test data generation approaches deals with the unit tests. There is very some work done in generating test data in integration testing. Especially in object oriented program automated test generation for integration testing is difficult task. At integration level, the variables are passed as arguments to other components and variables change their names; also multiple paths are executed from different components to ensure proper functionality. Search based software engineering is used for solving software oriented problems using evolutionary approach. We use Genetic Algorithm for the test data for object oriented integration testing using test cases generated in previous step. GA found to be very effective in test data generation (list of techniques using GA are listed in table 4.2)
One we have test cases, for object oriented integration testing, next step is to execute that tests. Test data must be prepared to successfully evaluate the test. Here we are considering data flow based coupling path as coverage criteria.

2.1 Coupling Path

Coupling path is defined as sequence of method, in which one method define a variable and second method use it .Ammann (2012). In object oriented programming coupling path can be direct or indirect. In coupling path, antecedent method is the one, defining a state variable and consequent method is using that state variable. If both antecedent and consequent methods are same, it is called directly coupling data flow. Figure illustrate the direct and indirect coupling data flow path.

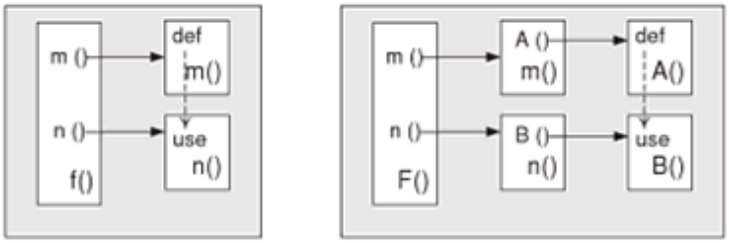


Figure 5.7: Coupling Path in OO Program

This coupling path may exists in single method, if a state variable is being defined and used in same method.
Consider following example to illustrate the coupling path. Using above method sequence generation algorithm, we get the test cases. These test cases are the sequence of method calls and good enough to test the integrated functionality of code.
Tests are in following format:

Object a, a.a1() , object b, b.b1(a)

In above example b1 was the coupling method and that particular test case is designed to transform object in desired state before testing based on define-use analysis of state variable. Coupling path for particular state variable is the sequence of statements or methods in which a variable is defined and finally used. This information is used in defining the fitness function for data generation to execute this test.

2.2 Fitness Function

In search based problems, solution is evaluated on fitness function. Fitness function is the reflection of the required solution. This function is used by evolutionary algorithms to explore the search space. For test data generation of object oriented program we propose fitness function based on coupling path. In unit test, its enough to have statement or branch coverage but for integration test criteria must be redefined.
Lets consider that, following annotations:

t: is the set of test cases
n: state variables of program under test
m: total methods in SUT
C: Is the set of coupling methods
C_v: is the coupling variable
C_p: is the set of coupling paths in coupling set C
b: is the branches involves in a coupling path
k: is the number of coupling path
m: is the number of branches in a specific coupling path

Fitness function is defined as:

$$fitnessfunction = 1 - \frac{|Cp_{cov}|}{|Cp|} + \frac{1}{\sum_{i=1}^{k}\sum_{j=1}^{m} d(b_{ij}, t) + 1} \tag{5.1}$$

$$d\,(b_k\,,t\,) = \begin{cases} 0, & \textit{if path is covered} \\ \textit{Branch Distance}, & \textit{if not covered} \end{cases}$$

d is distance, value is based on the coverage achievement. d is zero if this specific coupling path is covered otherwise branch distance is calculated. Coupling path may involves more than one branches, so d is the sum of all branch distances. Branch distances is calculated from following branch functions.

Branch Predicate	Branch Function F	relation
$E_1 > E_2$	$E_2 - E_1$	$<$
$E_1 \geq E_2$	$E_2 - E_1$	$\leq$
$E_1 < E_2$	$E_1 - E_2$	$<$
$E_1 \leq E_2$	$E_1 - E_2$	$\leq$
$E_1 = E_2$	$abc\ (E_1 - E_2)$	$=$
$E_1 \neq E_2$	$abc\ E_1 - E_2)$	$\leq$

Table 5.1: Branch Distance

We based this fitness function on the function defined for unit testing in Fraser and Arcuri (2011a). Where fitness is purely based on the methods covered in unit code under test. New function is capable of guiding the search considering the coupling criteria.
Genetic algorithm is found to be more accurate in test data generation field. GA can be successfully used in test data generation for integration testing of OO program. Genetic algorithms is one the heuristic algorithm with adaptive nature. GA uses the Charles Darwin theory of evaluation to find best solution. Competition in individuals for the resources results in fittest solutions to survive. It is based on natural genetics to solve optimization problem. An optimization problem can be define as finding best solution from different available solutions.

Implementation of genetic algorithm needs:

- Genetic representation of solution
- Fitness function

Individuals are selected dynamically. Each individual is designed considering the variables being input in the target test case. Individuals are dependent on respected test cases.
Above fitness function guide the GA to find optimal solution for test data generation. Following steps involves in genetic algorithm:

- Individuals are selected based on test case
- Each individuals fitness is calculated
- Individuals with the more fitness value are considered successful
- Successful individuals are then use to produce more offspring for next generation
- Two good parents produces the child solution that better than its parents
- Process continues until optimal solution is found or maximum iteration reached

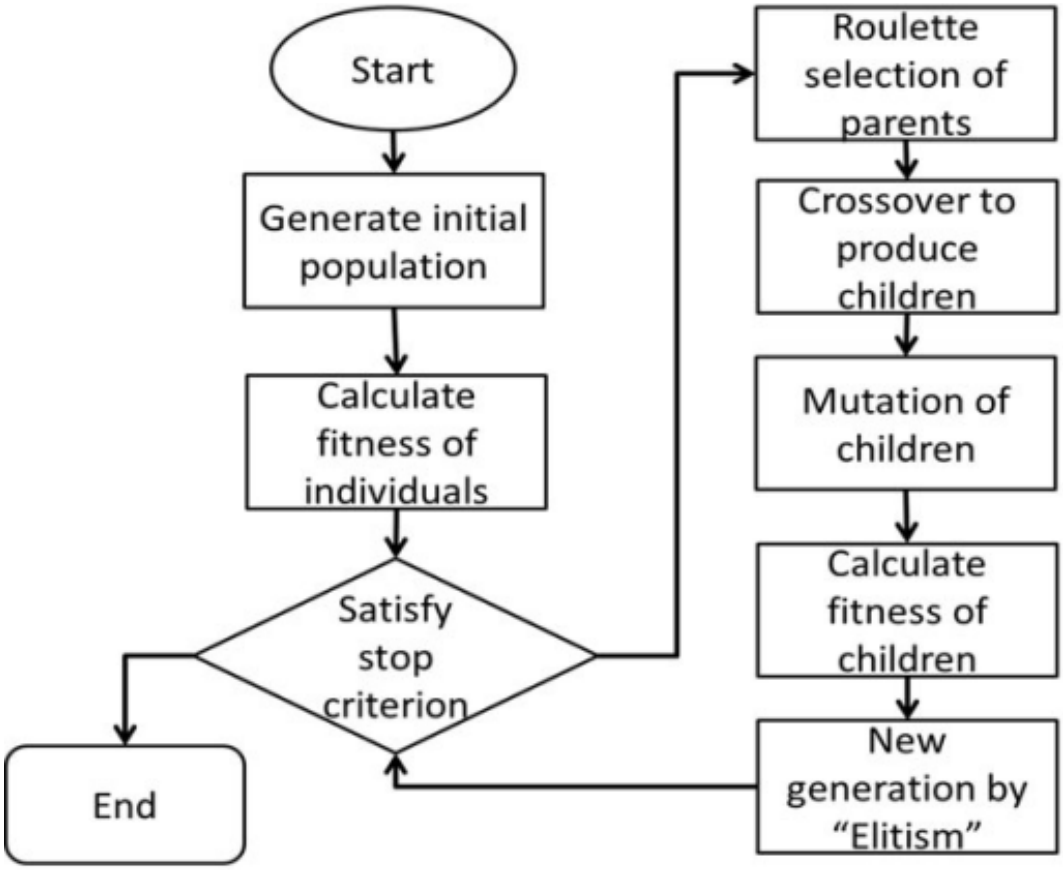


Figure 5.8: GA Flow

Individual is the set of combination of input variables that are involved in specific test case. Individuals are selected dynamically considering the variables involved in performing test. Once individual is selected, initial values are generated randomly. Fitness of each individual is calculated. For fitness calculation, we required to generate the coupling paths.
Table with define-use occurrences of each of state variable helps in selecting coupling path leading to fitness calculation. State variables may contains attributes from different classes. State variables are organized in following format.

Coupling Variable	Method 1	Method 2	Method 3	Method n...
Class.var1	d		u	
Class2.var2		d, u		d
Class2.var	u		d	
Class3.var	d		u	
Class3.var3		d	u	u

Table 5.2: Coupling path (define-use representation)

Table shows the summary of define and use variables with respect to methods. This table is dynamically drawn considering the methods involved in specific test. Coupling paths are directly gathered from given table.

For example:
Here is the simple test case:

Class3 obj1 ->obj1.Method1 () ->obj3.Method3 ()

In the above sequence and given define-use transition table, coupling paths are as follows. In method1 *var1* and *var* are being defined and in Method3 both variables are being used. There are total two coupling paths that must be covered to ensure the coupling testing. This information is being used in fitness function to guide the search space finding optimal input data values to run the test.

Chapter 6

Experimentation and Case Study Analysis

1 Introduction

This chapter introduces to case study and experimental details used for evaluation of proposed approach. Details includes test preparation, parameter setting and achieved results. We develop prototype tool AITT (Automated Integration Testing Tool) to test the significance of the proposed approach. AITT is developed in java.
Basic architecture of the tool is shown in following diagram. There four basic modules analyser, code instrumentor, test case generation, test data generation. Source code is accepted by analyser. Analyser compiles the code for errors and identify coupling classes and coupling methods from each coupling class. We use javassist tool to analyse the source code.
Instrumetor module takes the coupling methods and classes and store them into internal format. Internal format helps in further investigation on these classes and generating test cases accordingly.
Integrated test case generator is one of the important modules in AITT. Def-use analysis is performed prior to the generation of test cases. This analysis helpss in generating the method sequence to achieve the object in the required state. Objects must be in proper state to perform test properly. Def-use analysis is done for each coupling method to generate the test case. Our focus is to generate test cases to each of the coupling method that is involved in any type of integration. Coupling tree generator, generates the coupling sequence based on data flow in the form of tree to generate the test cases. Algorithm 1 and 2 discussed detailed process in previous section.

Once integration test cases is generated, then control transfers to the next module Test Data Generator. Genetic algorithm is used to generate the test data. An optimization problem can be define as finding best solution from different available solutions. GA needs to set parameters like population size, crossover and mutation to find optimal solution.

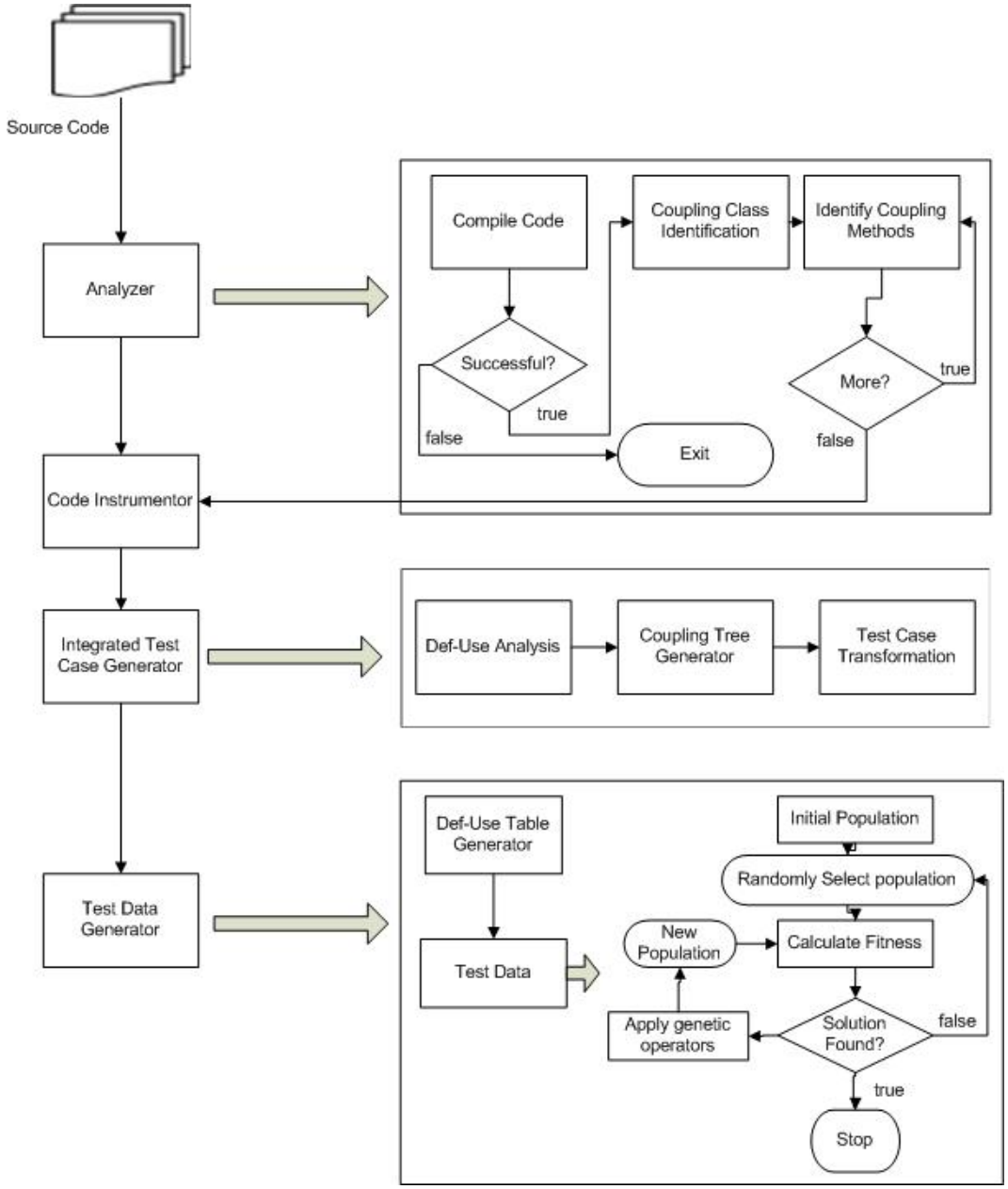


Figure 6.1: Tool Architecture

Here are the parameters used by GA in generating test data for object oriented integration testing.

GA Parameters	Values
Individuals	15
Population Size	20
Selection Technique	Fitness proportionate selection
Crossover	Two-point crossover
Crossover rate	0.80
Mutation rate	0.012
Individual Length	variable

Table 6.1: GA Parameters Used for Test Data Generation

Single population contains multiple individuals. In this problem, each individuals contain the variables involved in the given test case. Individuals compute the fitness using predefined fitness function. Each population contains multiple individuals. There is need to decide the max limit to number of population to avoid infinite problem. Crossover operation is use to generate the chromosomes (individuals) for next generation. It is the process generating new child from two or more parent solutions. Crossover rate defines how many individuals are selected for mating. It leads population to the optimal solution so far (exploitation). Experiments shows 0.80 gives better results in generating good individuals for our specific problem. While mutation maintains diversity in solution and make sure to avoid local optima. Mutation probability allows GA to decide whether mutate a solution or not.
Individual population is selected randomly. We set 100 as maximum population size and 50 individuals in each. GA process continues until optimal solution (test data) is found or maximum iteration is completed.

1.1 Case Study and results

We have selected randomly **11 java projects** from the **SF100** a benchmark for software testing Fraser and Arcuri (2012). We selected JIGL (Java Image and Graphics Library) project to show the detailedS flow of our technique. JIGL is Brigham Young University's Java Image and Graphics Library, originally developed by Bryan Morse's lab at BYU. There are total 43 classes in *JIGL project* that provides graphic support to java program. This project is divided into GUI, image, internal, maths and signal module.
GUI module contains six classes used for interface representation.

CloseableFrame.java: *CloseableFrame allows the user to create a frame that can be closed without exiting the program.*
CloseableMainFrame.java: *CloseableMainFrame allows the user to create a frame that can be closed and exit the program. This is not possible for java.awt.Frame.*
ImageCanvas.java: *ImageCanvas is a class made to facilitate the displaying of a JIGL image. ImageCanvas also easily supports a highlight box when active and a mouse drawn selection box.*
JImageCanvas.java: *JImageCanvas is a SWING compatible class made to facilitate the displaying of a JIGL image.*

SequenceCanvas.java: *Sequence Canvas is a class that handles all the appropriate functionality of a sequence. It includes animation and a control bar for that animation.*

There are 32 classes in image module. Provides classes for basic JIGL images, JIGL histogram and other auxiliary things. The basic types of JIGL images are: BinayImage, GrayImage, ColorImage, RealGrayImage, RealColorImage, ComplexImage. All kinds of image classes directly or indirectly implements.

ColorImage.java: *This class provides functionality on color images range from simple get pixel value, multiple, divide pixels from triplets to complex task like clip image, add images and subtract region of interest from another image etc.*

ComplexImage.java: *A complex image is a set of two RealGrayImage plane: real plane and imaginary plane. ComplexImage implements image.*

Histogram.java: *Histogram keeps track of histogram information for an image. The range of histogram's grayscale starts from min() and ends with max(). Supports only GrayImage and RealGrayImage.*

There 29 more classes in this module like phaseimage ,ROI, Operator, MIPMOP etc. According to the sourceforge Internal module will serves as wrapper and this module is not implemented yet.
Maths module contains nine classes and providing support for complex number, matrix and vector.

Complex.java: *This class provides support to add, subtract perform conjugates etc. on complex numbers*

Matrix.java: *This class allows a programmer to create real matrices with an arbitrary number of rows and columns.*

Vector.java: *This class allows a programmer to create real vectors of an arbitrary dimension.*

ScaleMatrix.java: *Class used to create the a diagonal matrix with diagonal entries from given array*

Other classes includes projection matrix, identity matrix and rotation matrix etc.
Signal module provide classes from for basic JIGL signals and other auxiliary things. Basic JIGL signals are: BinaySignal, DiscreteSignal, RealSignal, ComplexSignal. All kinds of signals directly or indirectly implement interface.

Powersignal.java : *PowerSignal takes a ComplexSignal and computes the value at x by adding the squared real and image plane.*

ROI.java: *The ROI class is the Region of Interest of a JIGL Signal.*

Phase signal: *A phase signal is a signal which takes a ComplexSignal and computes*

each pixel by taking the arc tangent of Imaginary plane and the Real plane.

ComplexSignal: *A complex signal is a set of two RealSignal planes: real plane and imaginary plane. Methods contains different functions to be performed on complex images like arthimetic operation,returns magnitude of region of interest and shallow/deep copy etc.*

Vector: *This class allows a programmer to create real vectors of an arbitrary dimension.*

SimpleOperator: *SimpleOperator is a base class that all levelOps and morph classes derive from. SimpleOperator supports BinarySignal, DiscreteSignal, RealSignal, and ComplexSignal.*
Project contains 43 classes and 11758 reachable line of code. In our propose approach we need to select the classes that are involved in integration for generation of coupling test cases leading to test data generation. We selected following classes that can be integrated considering the coupling concept. Some classes are *phaseSignal*, *complex signal*, *ROI*, *realsignal* used below.
We have selected above four classes involved in integration. Here are the list of coupling methods in coupling classes.

```
void divide(int x, float r, float i, ROI roi)
public PhaseSignal(ComplexSignal s)      //Parameter Coupling
public Signal copy()                     // Composition Coupling
public final void setReal(RealSignal pl) //Parameter Coupling
public final void setImag(RealSignal pl) //Parameter Coupling
public Signal copy(ROI roi)              //Parameter Coupling
public final Complex max(ROI roi)        //Parameter Coupling
```

Our focus is to test these coupling methods. Objects must be in proper state before testing a specific method. Test cases are generated using the data flow analysis. Each method is further used in generating coupling tree based on data flow of state variables. Given a coupling method:

void divide(int, float, float,ROI) complexSignal Class method

This function is used to divide a single pixel by a value in a Region of Interest. To test this method we first need to make objects in desired state. Here is the tree generated considering data flow analysis. Target method is on the root node. Second level nodes contains methods that defines state variables being used in root method. We generalize the approach as each n level nodes contains the methods that defines one or more state variable used in n-1 level.

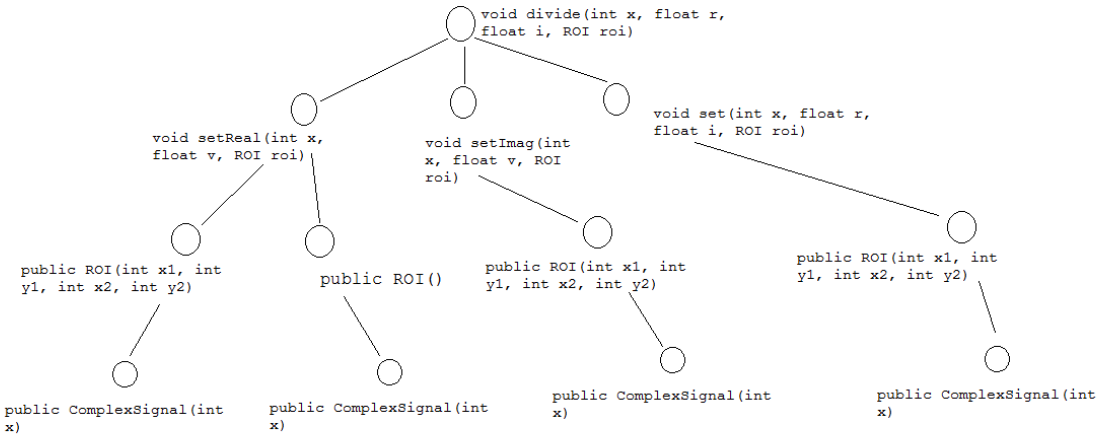


Figure 6.2: Intermediate Tree

Tree in figure 6.2. shows the methods sequence to get desired values of state variables. This tree is directly converted into test cases. Refer to the algorithm 2, describes the generation of test cases from the coupling tree.
Here are the test cases generated considering above tree.

int x;
ComplexSignal $0= new ComplexSignal(x)
int x1,x2,y1,y2;
ROI $2= new ROI();
int x;
float v;
$0.setreal(x,v,$2);
int x;
Float r,i;
$0.divide(x, r, i, $2);

---

int x;
ComplexSignal $0= new ComplexSignal(x);
int x1,x2,y1,y2;
ROI $2= new ROI(x1, x2, y1, y2);
int x;
float v;
$0.setreal(x,v,$2);
int x;
Float r,i;
$0.divide(x, r, i, $2);

```
int x;
ComplexSignal $0= new ComplexSignal(x);
int x1,x2,y1,y2;
ROI $2= new ROI(x1, x2, y1, y2);
int x;
Float r,i;
$0.set(x,r,i,$2);
int x;
$0.divide(x, r, i, $2);
```

```
int x;
ComplexSignal $0= new ComplexSignal(x)
int x1,x2,y1,y2;
ROI $2= new ROI(x1, x2, y1, y2);
int x;
float v;
$0.setImag(x,v,$2);
int x;
Float r,i;
$0.divide(x, r, i, $2);
```

Here is the next coupling method selected for generation of test cases:

public Signal copy() complexSignal Class method

This function makes copy of the signal and returns the mutated object. This functions is selected as a coupling method, because *RealSignal* object is being used in it. To test this method we need to generate the method sequences considering the data flow of state variables. Tree in figure 6.3. shows the methods sequence to get desired values of state

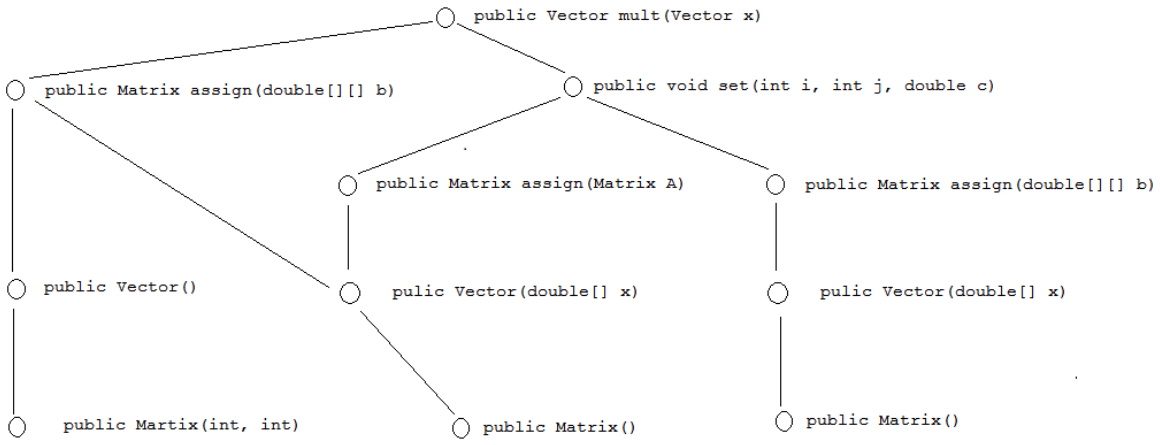


Figure 6.3: Intermediate Tree

variables. This tree is directly converted into test cases.
Here are the test cases generated considering above tree.

Once we have test cases next step is to generate test data using proposed fitness function. This fitness function is used by genetic algorithm to generate test data.

int x,y;
Matrix $0= new Matrix(x,y)
Vector $1=Vector()
double d[x][y];
$0.assign(d);
$0.mult($1);

---

int x,y;
Matrix $0= new Matrix()
double d[x][y];
$0.assign(d);
$0.mult($1);

```
Matrix $0= new Matrix()
int x[];
Vector $1=Vector(x)
Matrix $2;
int i,j;
double c;
$0.assign($2);
$0.set(i,j,c);
$0.mult($1);
```

```
Matrix $0= new Matrix()
int x[];
Vector $1=Vector(x)
double[x][y] b;
int i,j;
double c;
$0.assign(b);
$0.set(i,j,c);
$0.mult($1);
```

1.2 Test Cases Validations

In above example we have selected one coupling method *divide* that was involved in integration. Tree represents the sequence of method calls, considering in each way a method can be called. There is not just one test case for each coupling method. Number of test cases depends on the dynamics of data flow. Tree helps in generation of diverse sequences for each coupling methods.
Test cases cover 100 % coupling method, as diversity in test cases generation helps in achieving full coverage. In integration testing our main focus is to test the interfaces, that are directly involved in integration. Overall overage can be calculated by ratio of coupling method with uncovered methods.

There are total 68 methods in both classes ComplexSignal and ROI. Every method is called at-least once accept 11 methods, during proposed coupling focusing on coupling methods. So overall coverage is:

59/68*100= 76 %. We can conclude that, in this particular example our approach provides 100% coverage for integration testing but overall 76 % coverage is achieved for classes testing.

State Variable	divide(--)	setRel()	ROI(--)	complexSignal(--)
real	u	d		d
imag	u			d
Length				d
m_lbound			d	
m_ubound			d	

Table 6.2: Coupling Paths(divide(–))

Table 6.2 is used in fitness evaluation. Considering the test cases there are two coupling paths that must be executed for 100% coupling path coverage.

State Variable	Matrix()	Vector(--)	assign(--)	mult(--)
mem	d		d	u
m	d		d	u
n	d		d	u
Vector.mem		d		u

Table 6.3: Coupling Paths (mult(–))

SF100 is the benchmark for software testing. We have randomly selected 11 projects for evaluation of our approach.

Project	#Classes	LOC	Coupling Methods
Jiggler	43	11758	17
caloriecount	30	4686	24
gangup	23	3813	15
fixsuite	14	2228	17
a4j	19	1783	9
corina	15	3680	11
openhre	10	1078	16
bpmail	13	854	10
petsoar	6	496	4
freemind	65	15995	49
gfarcegestionfa	13	2195	18

Table 6.4: Test Projects Details

Project	Coupling Methods	Total Test Cases generated
Jiggler	17	42
caloriecount	24	56
gangup	15	22
fixsuite	17	44
a4j	9	25
corina	11	34
openhre	16	39
bpmail	10	22
petsoar	4	16
freemind	49	78
gfarcegestionfa	18	40

Table 6.5: Test Cases

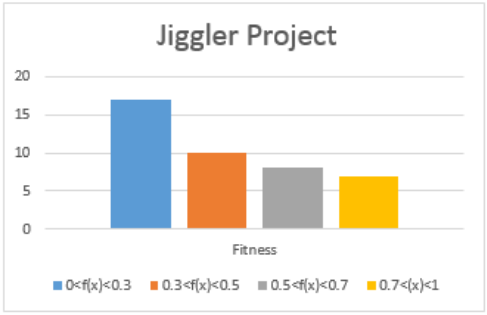
Project	Coupling Methods	Total Test Cases	Coupling paths	Covered	Coverage
Jiggler	17	42	55	50	91%
caloriecount	24	30	62	43	69%
gangup	15	22	47	42	89%
fixsuite	17	44	85	66	78%
a4j	9	25	38	38	100%
corina	11	34	70	61	87%
openhre	16	39	92	65	71%
bpmail	10	22	46	20	43%
petsoar	4	16	20	16	80%
freemind	49	78	114	86	75%
gfarcegestionfa	18	40	67	67	100%

Table 6.6: Test Coverage

Here are the ranges of the fitness function for each of the test project. GA parameters used are defined in table 6.1.

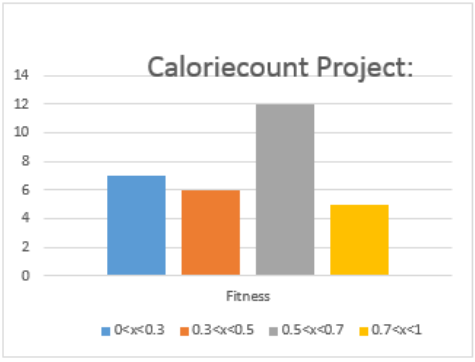
Jiggler Project:

Fitness Value	No. of Test Data
0<x<0.3	17
0.3<x<0.5	10
0.5<x<0.7	8
0.7<x<1	7



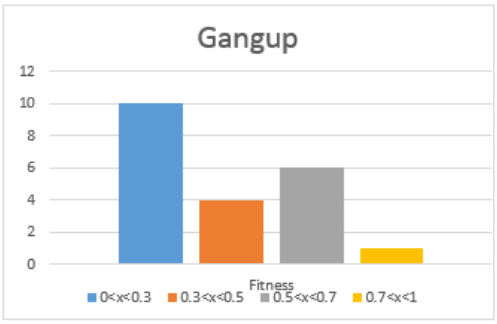
Caloriecount Project:

Fitness Value	No. of Test Data
0<x<0.3	7
0.3<x<0.5	6
0.5<x<0.7	12
0.7<x<1	5



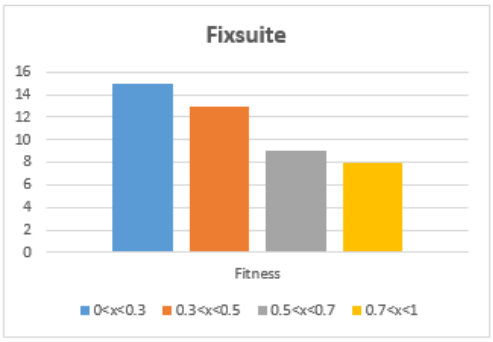
Gangup Project:

Fitness Value	No. of Test Data
0<x<0.3	10
0.3<x<0.5	4
0.5<x<0.7	6
0.7<x<1	1



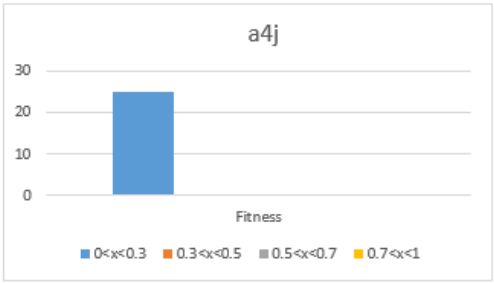
Fixsuite Project:

Fitness Value	No. of Test Data
0<x<0.3	15
0.3<x<0.5	13
0.5<x<0.7	9
0.7<x<1	8



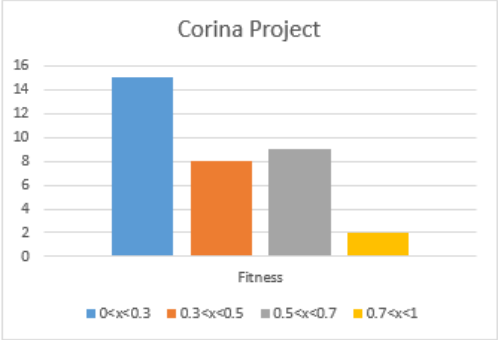
a4j Project:

Fitness Value	No. of Test Data
0<x<0.3	25
0.3<x<0.5	0
0.5<x<0.7	0
0.7<x<1	0



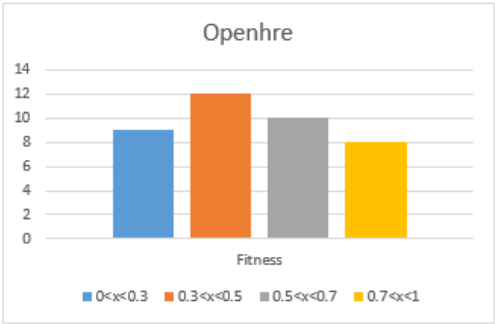
Corina Project:

Fitness Value	No. of Test Data
0<x<0.3	15
0.3<x<0.5	8
0.5<x<0.7	9
0.7<x<1	2



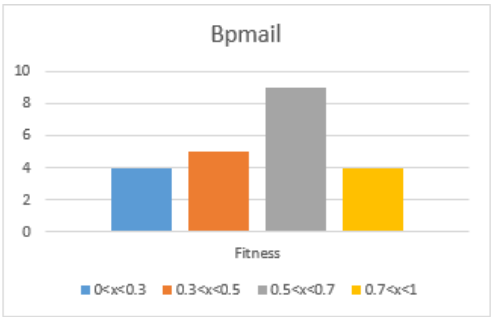
Openhre Project:

Fitness Value	No. of Test Data
0<x<0.3	9
0.3<x<0.5	12
0.5<x<0.7	10
0.7<x<1	8



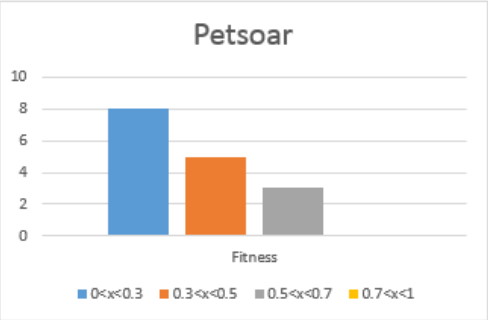
Bpmail Project:

Fitness Value	No. of Test Data
0<x<0.3	4
0.3<x<0.5	5
0.5<x<0.7	9
0.7<x<1	4



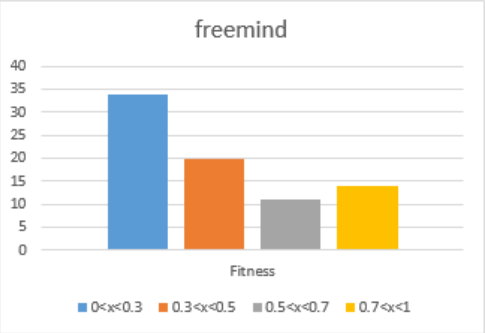
Petsoar Project:

Fitness Value	No. of Test Data
0<x<0.3	8
0.3<x<0.5	5
0.5<x<0.7	3
0.7<x<1	0



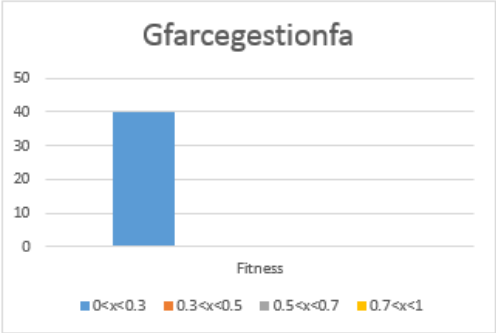
Freemind Project:

Fitness Value	No. of Test Data
0<x<0.3	34
0.3<x<0.5	20
0.5<x<0.7	11
0.7<x<1	14



Gfarcegestionfa Project:

Fitness Value	No. of Test Data
0<x<0.3	40
0.3<x<0.5	0
0.5<x<0.7	0
0.7<x<1	0



2 Results Discussion

Case study shows the step by step working of proposed framework. Jigl project is randomly selected for this purpose. Coupling tree shows the sequences of method calls to generate the test cases based on data flow analysis. This will get the objects in desired states before the further investigation is carried out. Code coverage is used to represent the degree to which a source code of program is tested. Path coverage criteria is used in this approach. Test cases are generated using data flow analysis of state variable considering only coupling methods. Proposed technique is evaluated on 11 randomly selected different project selected from SF 100. SF 100 is the bench mark projects available for software testing. It includes JAVA projects, mostly provides library support for further use. Coverage of each project is calculated using total coupling path identified and path covered by the test data generated using proposed fitness function. Overall more than 75 % coverage is achieved except four projects, having coverage less than 75%. Fitness values for each project is categorised in four different categories. Fitness value zero means that every coupling path is covered and one represents that not even a signal path is covered for specific test case. There is no criteria available for integrated test case generation. Most of the test cases generated randomly are not executable. And test data generated randomly without fitness function leads to low coverage as compared to our approach. Early proposed techniques were based on other coverage criteria i.e. statement, branch coverage etc or targeting unit level test case generation. Results clearly indicates that proposed approach is suitable when testing focus is integration level.

Chapter 7

Conclusion

1 Conclusion and Future Research Plan

In this thesis we presented a framework for integration testing of object oriented program. Our research is divided into two main portions, test case generation and test data generation. Testing can be performed in unit, integration or system level. Coverage based testing of object oriented program is intensive task due to path explosion. Lot of work done in unit testing of object oriented program. Automated integration testing is still active research area these days. We proposed algorithm for integrated test case generation. This algorithm is based on coupling relation of object oriented program. Most of the techniques uses statement coverage or branch coverage in testing of OO program. But this coverage criteria is suitable for unit testing. Once code is properly tested at unit level, there is no need to test each branch at integration level too. Selected coupling variables make it possible to only consider methods involves in integration. Coupling tree is generated for each selected methods considering def-use pairs. Final test cases are directly generated form proposed coupling tree. Test cases are further used to automatically generate input data used to execute the tests. Coupling path is uses as coverage criteria in this approach. Optimization algorithms work on fitness functions to find solution. Genetic algorithm is found more effective in test data generation. Experiments shows that our designed fitness function is capable enough to generate the data. Proposed fitness function is using coupling path and branch distance to guide the search for better coverage. We developed prototype tool in java for evaluation of proposed approach. We use 11 randomly selected projects from SF100 and average coverage is more than 80 percent for selected projects. Jigl project is selected as case study and step by step working of proposed algorithm shown including results. Proposed integration testing approach is only tested on java project, there is still need to evaluate this approach on programs written in other programming languages. Test oracles are used to evaluate the responses of software under test on specific test cases. Test data generation approach lacks the generation of correct strings used as input data. Due to large search space, it is seems difficult to generate the required string. This approach instead uses dummy strings to run the test. Further investigation is required crossover and mutation of strings with variable length. Our work is not including generation of test oracles. Generation of test cases in JUNIT compatible form and test oracles are left as future work.

References

Alexander, R. T. et al. (2000). Criteria for testing polymorphic relationships. In *Software Reliability Engineering, 2000. ISSRE 2000. Proceedings. 11th International Symposium on*, pp. 15–23. IEEE.

Ammann, Paul, J. O. (2012). Introduction to software testing. Cambridge University Press.

Ammann, P. and J. Offutt (2008). *Introduction to software testing*. Cambridge University Press.

Anand, S., E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, et al. (2013). An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software 86*(8), 1978–2001.

Baresel, A., H. Sthamer, and M. Schmidt (2002). Fitness function design to improve evolutionary structural testing. In *GECCO*, Volume 2, pp. 1329–1336.

Bashir, M. B. and A. Nadeem (2009). A state based fitness function for evolutionary testing of object-oriented programs. In *Software Engineering Research, Management and Applications 2009*, pp. 83–94. Springer.

Boyapati, C., S. Khurshid, and D. Marinov (2002). Korat: Automated testing based on java predicates. In *ACM SIGSOFT Software Engineering Notes*, Volume 27, pp. 123–133. ACM.

Bugrara, S. and D. R. Engler (2013). Redundant state detection for dynamic symbolic execution. In *USENIX Annual Technical Conference*, pp. 199–211.

Cadar, C., P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser (2011). Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, pp. 1066–1071. ACM.

Cheon, Y., M. Y. Kim, and A. Perumandla (2005). A complete automation of unit testing for java programs.

Ciupa, I. and A. Leitner (2005). Automatic testing based on design by contract. In *Proceedings of Net. ObjectDays*, Volume 2005, pp. 545–557. Citeseer.

Ciupa, I., A. Leitner, M. Oriol, and B. Meyer (2006). Object distance and its application to adaptive random testing of object-oriented programs. In *Proceedings of the 1st international workshop on Random testing*, pp. 55–63. ACM.

d'Amorim, M., C. Pacheco, T. Xie, D. Marinov, and M. D. Ernst (2006). An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pp. 59–68. IEEE.

Delahaye, M., B. Botella, and A. Gotlieb (2015). Infeasible path generalization in dynamic symbolic execution. *Information and Software Technology 58*, 403–418.

Denaro, G., A. Margara, M. Pezze, and M. Vivanti (2015). Dynamic data flow testing of object oriented systems. In *37th International Conference on Software Engineering, ICSE*, Volume 15.

Fraser, G. and A. Arcuri (2011a). Evolutionary generation of whole test suites. In *Quality Software (QSIC), 2011 11th International Conference on*, pp. 31–40. IEEE.

Fraser, G. and A. Arcuri (2011b). Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 416–419. ACM.

Fraser, G. and A. Arcuri (2012). Sound empirical evidence in software testing. In *Proceedings of the 34th International Conference on Software Engineering*, pp. 178–188. IEEE Press.

Harman, M., S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo (2010). Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pp. 182–191. IEEE.

Harman, M. and P. McMinn (2010). A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *Software Engineering, IEEE Transactions on 36*(2), 226–247.

Hermadi, I., M. Ahmed, et al. (2003). Genetic algorithm based test data generator. In *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, Volume 1, pp. 85–91. IEEE.

Jacobson, I. (1992). Object oriented software engineering: a use case driven approach.

Jamrozik, K., G. Fraser, N. Tillmann, and J. De Halleux (2012). Augmented dynamic symbolic execution. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pp. 254–257. IEEE.

Jin, Z. and A. J. Offutt (1998). Coupling-based criteria for integration testing. *Software Testing Verification and Reliability 8*(3), 133–154.

Kaner, C. (2006). Exploratory testing. In *Florida Institute of Technology, Quality Assurance Institute Worldwide Annual Software Testing Conference, Orlando, FL*.

Kao, G., M. Tang, and M. Chen (1999). Investigating test effectiveness on object oriented software-a case study. *Proceedings of Twelfth Annual International Software Quality Week*.

Khan, S. A. and A. Nadeem (2013). Automated test data generation for coupling based integration testing of object oriented programs using evolutionary approaches. In *Information Technology: New Generations (ITNG), 2013 Tenth International Conference on*, pp. 369–374. IEEE.

Khan, S. A. and A. Nadeem (2014). Automated test data generation for coupling based integration testing of object oriented programs using particle swarm optimization (pso). In *Genetic and Evolutionary Computing*, pp. 115–124. Springer.

Korel, B. (1990). Automated software test data generation. *Software Engineering, IEEE Transactions on 16*(8), 870–879.

Lin, J.-C. and P.-L. Yeh (2001). Automatic test data generation for path testing using gas. *Information Sciences 131*(1), 47–64.

Liu, L. L., B. Meyer, and B. Schoeller (2007). Using contracts and boolean queries to improve the quality of automatic test generation. In *Tests and Proofs*, pp. 114–130. Springer.

Ma, K.-K., K. Y. Phang, J. S. Foster, and M. Hicks (2011). Directed symbolic execution. In *Static Analysis*, pp. 95–111. Springer.

Maher, M. J. (2004). Advances in computer science-asian 2004.

McCabe, T. J. (1976). A complexity measure. *Software Engineering, IEEE Transactions on* (4), 308–320.

McMinn, P. (2004). Search-based software test data generation: a survey. *Software testing, Verification and reliability 14*(2), 105–156.

McMinn, P. (2011). Search-based software testing: Past, present and future. In *Software testing, verification and validation workshops (icstw), 2011 ieee fourth international conference on*, pp. 153–163. IEEE.

Meyer, B., I. Ciupa, A. Leitner, and L. L. Liu (2007). *Automatic testing of object-oriented software*. Springer.

Miller, W. and D. L. Spooner (1976). Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering* (3), 223–226.

Morell, L. J. (1990). A theory of fault-based testing. *Software Engineering, IEEE Transactions on 16*(8), 844–857.

Orso, A. (1998). Integration testing of object-oriented software. *Politecnico di Milano, Milano, Italy*.

Pachauri, A. and G. Srivastava (2013). Automated test data generation for branch testing using genetic algorithm: An improved approach using branch ordering, memory and elitism. *Journal of Systems and Software 86*(5), 1191–1208.

Ramler, R. and K. Wolfmaier (2006). Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. In *Proceedings of the 2006 international workshop on Automation of software test*, pp. 85–91. ACM.

Saxena, P., D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song (2010). A symbolic execution framework for javascript. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pp. 513–528. IEEE.

Sharma, C., S. Sabharwal, and R. Sibal (2014). A survey on software testing techniques using genetic algorithm. *arXiv preprint arXiv:1411.1154*.

Smith, B. (2015). Object-oriented programming. In *Advanced ActionScript 3*, pp. 1–23. Springer.

Tonella, P. (2004a). Evolutionary testing of classes. In *ACM SIGSOFT Software Engineering Notes*, Volume 29, pp. 119–128. ACM.

Tonella, P. (2004b). Evolutionary testing of classes. In *ACM SIGSOFT Software Engineering Notes*, Volume 29, pp. 119–128. ACM.

Tracey, N., J. Clark, K. Mander, and J. McDermid (2000). Automated test-data generation for exception conditions. *Software-Practice and Experience 30*(1), 61–79.

Whittaker, J. et al. (2000). What is software testing? and why is it so hard? *Software, IEEE 17*(1), 70–79.

Williams, L. (2006). Testing overview and black-box testing techniques. *URL: http://agile. csc. ncsu. edu/SEMaterials/BlackBox. pdf (accessed: 05/08/2008)*.

Xie, T., D. Marinov, and D. Notkin (2004). Rostra: A framework for detecting redundant object-oriented unit tests. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pp. 196–205. IEEE Computer Society.

Xie, T., N. Tillmann, J. De Halleux, and W. Schulte (2009). Fitness-guided path exploration in dynamic symbolic execution. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pp. 359–368. IEEE.