CHAPTER 2

Literature Survey of the Techniques for OO Testing and Testability Assessment

2.1 Introduction

Software quality can be assured through two ways: 1) by testing the software, and 2) through testability improvement. Testing is a very important activity of the software development life cycle. The area of object oriented testing has emerged with the increasing interest in using the object oriented paradigm for the software development. Testing object oriented systems is more important than testing the procedural software, as it promotes reuse. There are a number of issues associated with OO software which are not relevant in the procedural software like problems due to inheritance, problems due to polymorphism etc. [4]. Many researchers claim that not every procedural testing technique is applicable for testing OO software and there needs to be a different set of techniques for testing the object oriented software [17, 47, 48, and 61]. This is because these two approaches of software development have different concepts. Hence, the techniques for testing both the paradigms should be different. Object oriented software focus on the concepts like class, object, encapsulation, information hiding, inheritance, polymorphism, and dynamic binding etc. Each of these concepts adds a different set of problems for testing the object oriented software. Also, the levels of testing procedural software and the object oriented software are different due to the difference in both the paradigms. A lot of
research has been done in the field of object oriented testing and various techniques have been developed for testing the object oriented programs.

Binder [22] claims that testing can be made more effective if the software systems are designed in such a way that their testability is high. Software testability is an external software attribute which measures the testing effort required to test the software. Hence, there is a need of OO testing techniques which emphasize on reducing the testing effort and improving the testability of the software. Hence, new testing techniques should be developed which reduce the testing cost. To develop such OO testing techniques: 1) there is a need to have an understanding of software testability, and 2) the ways in which the testability of the OO software can be improved. A number of approaches to testability assessment have been proposed in the literature [22, 52, 57, 81, and 83].

This chapter explains different levels of testing the object oriented software, the problems in testing the object oriented software and a survey of the various object oriented testing techniques which exist in the literature. This chapter also presents various testability assessment techniques for procedural as well as OO software.

2.2 Levels of Testing Object Oriented Software

The testing for procedural software is carried out at three levels: unit testing, integration testing, and system testing. Traditional unit and integration levels defined for procedural programs do not fit well in case of object oriented systems.

The main difference between the testing levels of object oriented software and procedural software comes from the fact that the smallest unit of testing in object
oriented software is different from the one in the procedural software. The smallest unit of testing in object oriented software is class whereas in procedural software, it is a procedure or a function. Various researchers [3, 100, and 11] have given different levels of testing for object oriented software, these are described below.

Smith and Robson [100] have given the following classification of the testing levels for the object oriented software.

- **The algorithm level**: This is the method level.

- **The class level**: In this level the interaction between the attributes and methods of a class is tested.

- **The cluster level**: In this level the interactions amongst a group of cooperating classes is tested.

- **The system level**: In this level the testing of the entire system is done

Of these four levels, the class and the cluster levels are specific to the object oriented paradigm, while the algorithm and system levels resemble the traditional unit and system levels. There is yet another classification of levels for object oriented testing given by Jorgenson [111]. He provides three levels of object oriented testing: unit level (class level), integration level (cluster level), and the system level.

### 2.3 Object Oriented Testing Problems

Object oriented software has different concepts like object, class, inheritance, encapsulation, polymorphism, dynamic binding etc. and they introduce different kinds of problems for testing these systems [3, 41, 99, 109, and 123]. Testing object oriented methods is easier than testing the procedural methods while integration
testing is difficult in case of object oriented systems [117]. In the literature, a number of researchers have mentioned the OO testing problems caused by different object oriented concepts. Some of these problems are mentioned below.

**The problems due to Inheritance are as follows:**

The problems posed by various authors due to the property of inheritance are as follows:

**The problems of inheritance by Perry and Kaiser [41] are as follows:**

1. The inherited methods must be tested again in the context of the derived class.
2. The child class should be retested for the overridden methods.
3. The child class must be retested if the specification order of the parent classes is changed.

**The problems of inheritance by Smith and Robson [99] are as follows:**

Smith and Robson have classified inheritance into restrictive, non-restrictive, and repeated inheritance; multiple and simple inheritance. Simple inheritance means inheriting from only one parent class while multiple inheritance occurs when a child class inherits from two or more parent classes. Strict inheritance occurs when the child class takes all the features from the parent class. Nonstrict inheritance occurs when some of the features from the parent class are not present or are renamed in the derived class. Repeated inheritance occurs when a child class derives from the same parent class more than once. This occurs when a class inherits from two other classes that inherit from a common parent class. The problems of testing due to the different kinds of inheritances are as given below.
**Problems due to the strict multiple inheritance:**

1. A problem occurs if two members in two base classes have the same name.
2. If the parent class is changed then the child class needs to be retested.

**Problems due to the nonstrict inheritance:**

If the redefined feature is invoked in the context of the parent class for an object of the child class then it causes runtime error.

**Problems due to the repeated inheritance:**

It is not clear whether the test cases of the parent class can be reused to test the child class or not. A different set of problems given by Orso and Silva [3] due to the inheritance property are as follows:

- Testing the multiple inheritance and repeated inheritance is difficult due to the same name problem.
- Whether the base class test cases can be used for testing the derived class.
- There should be a proper method to test whether the inheritance relation is truly representing the IS-A relationship or not.
- Testing an abstract class is difficult as it can not be instantiated.

**Problems due to message passing and concurrency:**

Smith and Robson [99] state that the control flow in a conventional program and an OO program are different. Flow of control in OO programs is like the message passing from one object to another, the receiving object performs some operation in response to that message. Such programs are tested by testing the state of the object after passing every message.
2.4.1 Unit testing techniques

Unit testing is a class level testing technique as class is the smallest unit to be tested in OO systems. All the following techniques are class level testing techniques.

Harrold [107] has given a class testing method that shows the reduced effort of testing child classes by reusing the testing information of the parent class. The method consists of the following steps: 1) initially, all the member functions of the base classes are tested individually and the interaction among the member functions is tested, 2) a testing history is maintained for each base class and each subclass derives its testing history from the testing history of its parent class, 3) a subclass's testing history indicates which test cases must be run to test the subclass, 4) the inherited attributes are retested in the context of the subclass by identifying and testing their interactions with the newly defined attributes of the subclass, 5) the test cases in the parent class's test suite that can be reused to validate the subclass and attributes of the subclass which require new test cases are also be identified.

Parrish [1] has developed a technique for testing a class in which test cases are generated from the implementation of a class. This is helpful in cases where there is no formal specification of the class. This technique also differs from the traditional methods in that its generation of test cases is not random but systematic. The technique involves making every message a node, and a directed edge between nodes represents the possibility that one routine might be invoked followed by the second routine. The authors discuss two specific formal specification methods that can be applied to this technique: algebraic (axioms) and model-based (require and ensure clauses). They acknowledge that there is a problem in generating the infeasible test
cases, and the generated test cases may not be better than the manual testing. They suggest supplementing the generated cases with manual cases, especially when testing the critical classes.

Doong and Frankl [115] gave a specification based approach for unit testing of object oriented programs. Authors emphasize that the natural unit to test is a class. This approach focuses on interaction of operations. Each test case consists of a tuple of sequences of messages along with the tags indicating whether these sequences should put the objects of the class under test into equivalence states or return objects that are in equivalent states. In this technique the classes are represented in the form of algebraic specification. The concept of inheritance is also taken into consideration for testing the class.

Harrold [60] proposed a code based, data flow testing technique for testing a class. The conventional data flow testing can be applied both to the individual methods in a class and to the methods in a class that interact through messages, but this technique can not be applied to test the data flow interactions that arise on the invocation of a sequence of methods in an arbitrary order. This technique determines which sequence of methods should be executed to test a class even without a specification. The class testing is divided into three levels. 1) Intra-method testing which tests the member functions individually. 2) Inter-method testing which tests a member function together with other member functions that it calls. 3) Intra-class testing which tests the interaction of the member functions when they are called in various sequences. A class control flow graph that connects all the methods in a class is constructed from the class’s code to compute the data flow information. The test
cases are generated using the inter-procedural data flow testing technique. If this technique is used along with the black box testing then it also helps in reducing the testing of unnecessary sequences of methods, as it provides the information about the sequences in which the methods do not interact.

According to Hong [76], the important characteristic of a class is the interaction between data members and the member functions, and this is represented as the definitions and the uses of the data members in the member functions. This is a specification based technique which specifies the behaviour of a class in terms of Class State Machine (CSM). Then, the test cases are generated based on the data flows in CSM. The data flows are identified by converting the CSM into the class flow graph (class flow graph represents both the data flow and the control flow of a class.) and then the conventional data flow testing is applied to generate the test cases using the data flow testing criteria. The test cases are able to find if, data flows in CSM are correctly implemented in the class’s code or not.

Kim [62] states that the class is the basic unit of testing. Class testing measures the interaction between the operations of a class and data bindings are useful for measuring the interaction between the operations of a class. The method of testing a class involves three steps: 1) each method of the class is tested individually using the conventional structured and functional methods of testing; 2) actual data bindings are found which give the measure of interaction between two member functions. An actual data binding set is defined as an ordered tripple <ml, d, m2> where ml and m2 ∈ Methods, d ∈ Data and ml assigns a value to d and m2 references d. A sequence is defined as a non-empty, ordered set of methods of a class for
specific data member of a class. A simple MM-Path is a sequence of a pair of methods represented by the actual data binding, and 3) test the sequence of methods. Testing order of the member functions of a class is defined as a define use relationship of the data members. A class is divided into small pieces associated with each data member and each piece is called a slice. A flow graph is made for each slice and a test sequence for each slice flow graph is made and tested. The combination of test cases for each slice flow graph gives a test suit for the class.

Wang [37] gives an approach that describes an Inheritance level Technique (ILT) as a guide to detect the software errors of the inheritance hierarchy and measures the software complexity of the inheritance hierarchy. The measurement of inheritance metrics and some testing criteria are formed based on the proposed mechanism. Lex and Yacc are used to construct a windowing tool which is used in conjunction with a conventional C++ programming environment to assist a programmer to analyse, test, and measure the C++ programs.

Alkadi [6] has developed a testing assistant, Object Oriented Testing Assistant (OOTA) that facilitates the testing of object oriented code by incorporating the procedures to support object level testing and inheritance testing. OOTA provides a framework that helps to ensure that appropriate components and interactions are tested by generating the code segments that drive the testing process. OOTA was developed and tested using the object oriented paradigm.

A class testing technique given by Chen [103] is based on the object flow based testing. It takes into account the dynamic behaviour of the object oriented programs. Two new coverage criteria have been proposed which are: all bindings and
all def-use pairs. All bindings are determined using the inheritance hierarchy. The def-use pairs are of two types: intra class and inter class object def-use pairs. Constructing an object control flow graph identifies intra class def-use pairs. Inter class object def-use pairs are identified by making use of interprocedural data flow testing methodology proposed by Harrold [60].

Tsai [8] has given a technique for testing the object oriented classes using the data flow analysis. This technique is based on the fact that the data members of a class, which do not participate in the determination of the state of an object, can also give rise to an error and must be tested. Data flow testing is used to test these data members. Firstly, du paths are calculated and then data flow test cases are generated. After that, infeasible and ambiguous test cases are found. Feasible test cases are obtained using the state transition diagram. Then, data flow anomalies are found within the sequences of messages and are removed.

Bayeda [119] has given a class testing technique based on a graph representation of a class, which combines the specification and the implementation of a graph. Two types of views of a method: the specification view and the implementation view are represented by two control graphs. The representation is called class specification implementation graph (CSIG). The CSIG representation covers both the specification and the implementation test cases. In this technique, the class is specified using the class state machine [76]. CSIG is constructed from this specification and then the test cases are generated using the integrated white and black box testing. The construction of CSIG involves the following steps: 1) Generate method implementation graph. 2) Generate a prototype for each method. 3) Generate
method specification graph. 4) Generate a CCFG (Class Control Flow Graph Frame) frame. 5) Insert method graph into a CCFG frame. 6) Connect method graphs with each other.

Sun and Chen [149] gave a method for testing a class based on the UML model in which two objects are tested for equivalence. Few concepts are defined such as; an observer of a class is an operation that returns attribute values of an object of class without affecting any observable attributes. A creator of a class is an operation that returns initial objects of a class. A constructor or transformer of a class is an operation that acts on an object and changes at least one attribute value of the object. An observable context on C is a valid sequence of operations in a class that starts with a constructor or transformer but ends with an observer. As observable contexts are infinite hence only relevant observable contexts are selected using the Data Member Relevance Graph (DRG). Given a canonical specification and its implementation in a class C, two objects O1 and O2 are said to be observationally equivalent if and only if, for any observable context oc on C, O1.oc and O2.oc produce either identical results or are observationally equivalent objects in the output class of oc.

Wappler [128] uses evolutionary algorithms for the automatic generation of the test cases for the white box testing of object oriented software. The test cases are test programs to manipulate the objects to achieve a certain test goal. Strategies for the encoding of test cases to evolvable data structures as well as ideas about how the objective functions could allow for a sophisticated evaluation are proposed. The approach has been implemented by a prototype for the empirical validation. In
experiments with this prototype, evolutionary testing outperformed the random testing.

2.4.2 Integration Testing Techniques

In object oriented paradigm, a cluster is a collection of classes. Hence, cluster level in object oriented paradigm is same as the integration level in the procedural paradigm. This section presents the integration/cluster level testing techniques for the OO software.

Jorgensen [111] has proposed a behaviour-based integration testing technique for the object oriented software. The author says that the functional decomposition emphasizes the levels of testing in the traditional software and the integration order. But there is not a defined integration structure in object oriented systems. Certain errors are revealed only on integration of objects and unit testing cannot reveal the integration level errors. A few new constructs for integration testing have been proposed like: method, message quiescence, event quiescence, thread testing, and thread interaction testing. A method message path (MM path) is a sequence of method executions linked by messages. An MM path starts with a method and ends when it reaches a method, which does not issue any messages, this point is called message quiescence. Event quiescence is an output event generated by an input event. An atomic system function is an input event, followed by a set of MM-paths, and terminated by an output event. Atomic system function is a point where integration and system testing coincide, thus forming a seamless flow between these two types of testing.
Kung [43] solves the test order problem of classes and gives a test strategy for object oriented programs based on the object relation diagram (ORD). ORD is a reverse engineering based model which is generated from analysing the C++ source code. An ORD is a directed graph where vertices represent the object classes and edges represent the relationships among the object classes. This strategy is used for unit testing and integration testing of object oriented programs. The algorithm used for testing uses the topological sorting and clusters of strongly connected subgraphs of ORD. This strategy finds an order to test the classes so that the effort required to construct the test stubs is minimum. The concept of topological sorting is used to find the test order.

Traon [92] presents a model, a strategy, and a methodology for planning integration and regression testing for testing the object oriented software. It shows how to produce a model of structural system test dependencies, which evolves with the refinement process of the object oriented design. The model (test dependency graph) serves as a basis for ordering the classes and the methods to be tested for regression and integration purposes (minimization of test stubs).

Labiche [90] addresses the problem of test order and emphasizes on the problem of reducing the number of stubs. Ordering of testing levels is based on the class diagrams. Polymorphism and dynamic binding features of object oriented systems are considered. Abstract classes are also taken into consideration. Class dependency analysis is done to find the testing levels through the following steps: 1) Calculate the set of classes on which a given class statically depends. 2) Calculate the set of classes on which a given class depends statically or dynamically or both. 3)
Find a boolean function to indicate whether or not a class dynamically depends on at least one class of set obtained in step 2. A test order graph is used to add the information to the testing levels.

Badri [94] addresses the problem of reducing the number of test stubs and determines an effective class integration order. The class integration strategy based on the new class dependency model (CDM) is developed that takes into account the interaction between classes. The objective of the class integration testing is to find the errors related to the interactions between the classes. CDM is a directed graph where the vertices represent the classes and the directed edges represent the dependencies. Inheritance and use are the two relationships that are identified by CDM. A class A uses class B if at least one of class A’s method calls at least one of class B’s methods. The integration strategy has following four steps: 1) Major level no. affectation, 2) Breaking dependency cycles, 3) Minor level affectation, and 4) Testing order generation. In step 1 the classes which are not inherited from any other class are given level 1 and the inherited classes from these classes are given level 2. In step 2, within each major level the dependency cycles are considered. The weight of each class is determined as the sum of its leaving links participating in one or more dependency cycles. The class having highest weight is used as a stub. Its entering links are removed from the diagram. The weight of each class is recalculated until there are no more dependency cycles in each of the major levels. In step 3, each class is given a minor level number taking into account its dependency relationship with other classes. If a class B depends upon class A then B will receive the minor level no. of class A augmented of 1. This process is done at each major level. In step 4, the
integration order is based on the couples (major level, minor level) assigned in the previous steps. The classes having lower minor level are integrated before the classes having higher minor level.

Chen [34] gives a unified methodology for object oriented software testing at the class and the cluster levels. The classes are specified by type signature specifications, including algebraic specifications and the clusters are specified by the contract specifications. Fundamental pairs of equivalent ground terms are used as the class level test cases and the observational equivalence of objects is determined by a relevant observable context technique. Nonequivalent ground terms are used as further class level test cases. The cluster level test suites are made using the sequences of message-passing expressions and postconditions.

Binder [23, and 24] has presented a Flattened Regular Expression (FREE) approach for testing the object oriented software. In this approach, the classes and the clusters of classes are represented as the state machines. The class hierarchy is flattened so that every class in the hierarchy, to be tested, is self contained. The test cases are the sequences of the messages which will cover all the states and the transitions. A class is tested by constructing a graph which connects all the methods within a class, connecting all the intra-method and inter-method data flow paths. Each transition in the state machine is replaced by the corresponding flow graph of the method. The state transition edge is connected to the entry node of the method and the exit node of the method is connected to the resulting state from the transition. The graph thus obtained contains all the possible paths which can exist among the methods within a class.
2.4.3 State Based Testing

Object state testing is a very important aspect of testing object oriented software. It is different from the conventional white box and black box testing. Object state testing focuses on testing the state dependent behaviour of an object. Some of the state based testing techniques are discussed below.

Turner [33] states that the most important thing to test is the interaction between the features and the state of an object. A new technique has been devised using finite state automata to test this. This technique takes into account the random order in which the features can be invoked. The features of an object are the methods of the objects. The state of an object is the combination of the values of all the data members of the class. Hence, sub states are defined as the value of a particular data member at a specific point in time. The substate values are defined to be of two types: specific substate values, and the general substate values. In this technique the following steps take place: 1) current state of an object is found, 2) a feature is invoked, and 3) the resulting state of the object is validated. Similarly, this technique is applied to the integration testing where the interaction of objects of more than one class takes place. If an object of a class is passed as a parameter to another class then both the classes should be unit tested before and then the integration testing of both the classes should take place.

Kung [42] states that certain errors in the implementation of the object state behaviour can not be readily detected by the conventional testing methods. i.e. structural testing and functional testing. The author describes an object state model and a reverse engineering method for extracting the object state behaviour from the
C++ source code. The object state test model is a hierarchical, concurrent, and communicating state machine. In the reverse engineering method the symbolic execution is used to extract the states and the effects of member functions, and the results are used to produce the state machine. The state of a simple object in C++ is captured by the ranges of the values of a subset of the member data of the object. For example, in library system if there is a class with data members: 'book title' and 'number of copies' then 'book title' does not participate in determining the object state, as it never changes. Only the other data member is determining the state.

Martena [134] has developed an interclass testing technique which tests the interaction among the classes. The test case specifications are derived using the data flow analysis. This technique is based on the idea that the execution of a method is affected by the instance variables used by the method and the methods that determine their values. The pairs of methods that define and use the same instance variables are selected. A feasible sequence of method invocations that contains identified definition in correct order is selected. The identified sequence represents the test cases for the class.

A lot of OO testing techniques have been developed to solve various problems of object oriented testing by using various concepts like finite state machine, scenarios, and UML etc. Still, there is a need to develop the testing techniques which may improve the software testability and reduce the testing effort. In a view to develop such testing techniques there is a need to have a complete understanding of software testability and how it can be improved.
Hence, a literature survey on software testability is performed for the conventional as well as the object oriented software.

<table>
<thead>
<tr>
<th>Level Testing</th>
<th>Developed By</th>
<th>Concepts used</th>
<th>Features tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit Testing</td>
<td>Sun and Chen [149]</td>
<td>UML Model</td>
<td>Equivalence of two objects</td>
</tr>
<tr>
<td></td>
<td>Doong and Frankl [115]</td>
<td>Algebraic Specification</td>
<td>Equivalence of two objects, Inheritance</td>
</tr>
<tr>
<td></td>
<td>Wappler [128]</td>
<td>Evolutionary Algorithms</td>
<td>Unit Testing</td>
</tr>
<tr>
<td></td>
<td>Kim [62]</td>
<td>Data Bindings</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Harrold[60]</td>
<td>Data Flow Testing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Harrold[107]</td>
<td></td>
<td>Inheritance</td>
</tr>
<tr>
<td></td>
<td>Alka[6]</td>
<td></td>
<td>Inheritance</td>
</tr>
<tr>
<td></td>
<td>Hong [76]</td>
<td>Data Flow Testing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bayeda [119]</td>
<td>Class specification implementation graph</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Chen [103]</td>
<td>object flow based testing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wang [37]</td>
<td>Lex and Yacc</td>
<td>Inheritance</td>
</tr>
<tr>
<td>Integration Testing</td>
<td>Chen[34]</td>
<td>Algebraic Specification, Contract Specification</td>
<td>Equivalence of two objects</td>
</tr>
<tr>
<td></td>
<td>Binder[23][24]</td>
<td>State machine, data flow graph</td>
<td>Inheritance</td>
</tr>
<tr>
<td></td>
<td>Badri [94]</td>
<td>Class Dependency Model</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Traon[92]</td>
<td>Test Dependency Graph</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Labiche [90]</td>
<td>Class Diagram</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Kung [43]</td>
<td>Object Relation Diagram</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Summarization of OO testing techniques
The following sections describe the research in the area of software testability and the ways to improve the software testability.

2.5 Testability Measurement Techniques

Software testability is an important characteristic of the software and is defined in literature with different viewpoints by various researchers. The testability has been measured at different levels of the software development life cycle: analysis/design level or code level. Software testability is described in terms of the software development process and as a characteristic of the architecture, design, and implementation of the software [88]. Software testability is important because it can be used in the estimation of other software attributes like dependability [19], reliability [147], and reusability of previous verification [143] etc. Researchers have discussed testability with respect to the procedural software, object oriented software, component based software, and the product line software. A number of tools have also been developed to measure the software testability. Below we discuss different approaches for assessing and improving the software testability for procedural as well as the object oriented software.

The IEEE [50] standard glossary defines testability as the degree to which a system or component facilitates the establishment of test criteria and performance of tests to determine whether those criteria have been met. ISO [77] defines it as: ‘attributes of software that bear on the effort needed to validate the software product’.

Bache and Mullerburg [9] measure the testability in terms of the effort needed for testing the software. The testing effort is measured as the minimum number of test
cases required to obtain the full coverage of a given coverage criterion. They make
use of the Fenton–Whitty theory and provide testability measures from the control
flow graphs based on the control-flow based coverage criteria. These measures can be
easily defined and can be calculated from any given procedural language program (for
example C, FORTRAN etc.). The testability measures are only limited to the control
flow based testing methods and are also dependent upon a given testing coverage
criterion. However, the authors claim that the testability measurement should also
take into account the data flow in addition to the control flow. A tool named
QUALMS is used to calculate all the testability metrics based on the control flow.
These testability measures are calculated at the unit level.

Voas and Miller [136] define software testability as the “probability that a
piece of software will fail on its next execution during testing if the software contains
a fault”. They use sensitivity analysis to make an assessment of the software
testability in which the original program and its mutant versions are repeatedly
executed and the probability of detecting the mutants is estimated. This estimation
enables the tester to identify the locations where more amount of testing effort is
required. The testability analysis has been done using the black box testing. This
technique assumes that there can be a single fault in a program. The main purpose of
sensitivity analysis is to provide information that tells how small the program’s
smallest faults are. Through this information one can use the statistical methods to
find how much testing is required to detect the faults of this size, which helps in
determining when to stop the testing. The outcome of the sensitivity analysis is the
probability of failure which occurs if a particular location contains a fault. Voas has
given a method to find the probability of failure. For a particular location in the program, sensitivity analysis provides an estimate of the probability of failure that would occur by a single fault. For the occurrence of failure and its observation, three conditions must hold: 1) the fault must be executed, 2) the data state should be infected, and 3) the incorrect state should be propagated to the output. Thus, sensitivity analysis divides failure into three kinds of processes: execution, infection, and propagation; and estimates the probability of each event through the analysis algorithms. Sensitivity analysis results in the estimated probability of failure that will occur if a particular location in the program contains a fault. Voas [137] provides a dynamic failure based technique that estimates these three kinds of probabilities that help in assessing the testability of a program. A tool named PISCES is developed in C++ language which implements these techniques to estimate the testability of a program [138]. The success of this technique heavily depends upon the testing technique used to derive the test cases to execute on the original program and the mutant version of the program. Voas [141] has performed an empirical study to compare the sensitivity analysis, a dynamic testability prediction technique and the cyclomatic complexity, a static testability technique, using an application which is a CASE version of B-737 autoland system. For this particular application six static measures are used which are: Software Science Length, Estimated Software Science Length, Software Science Volume, Software Science Effort, Cyclomatic Complexity, and Extended Cyclomatic Complexity.

Voas and Miller [139] have developed another approach to determine the fault sensitivity of a given program which focuses on the analysis of semantic content
contained in the program specification and the design documents. The high value of fault sensitivity indicates high testability and low value of fault sensitivity indicates low testability. The fault sensitivity of a program is given by the information loss occurring within the program. Information loss is of two types: implicit and explicit information loss. Implicit information loss occurs when two or more different input parameters produce the same result when presented to a user-defined function or a built-in operator. Implicit information loss can be obtained from the program's specification through a specification metric, the domain-to-range ratio. Explicit information loss occurs when the variables can not be validated during the execution of a program or at the end of the execution. The main cause of explicit information loss is information hiding. Information hiding is a property due to which a module is not allowed to reveal its information since it can be misused. Although this is an essential programming practice but it leads to lowering the testability at the system level, because the value of the local variables is not accessible due to this property. Explicit information loss is difficult to obtain in the early phases of the development and can be obtained through the static code inspection, and the design documents. Voas [145] also claims that software testability can be predicted throughout the development process using the formal specifications, design documents, and the code of the software. Voas [144] mentions that the design heuristics should be to reduce the implicit and explicit information loss. Voas [142] emphasizes the importance of the design characteristics of the software in improving the software testability. Voas [140] claims that software testability is important software characteristic to determine how to best apply verification techniques and to develop quality assurance plans.
Voas uses the testability measures for the intelligent assertion placement as assertions are known to improve the testability.

Freedman [49] emphasizes the testability of the components as important to the software engineering process. He defines domain testability based on the concept of controllability and observability of the software. Controllability means the ease with which a specified output can be produced from a specified input and observability is the ease of determining if a given input has affected the output. A domain testable program is observable and controllable as it does not have any test inconsistencies of input and output. Testing metrics are defined to assess the effort required to make a program domain testable through the modification. The author explains how the testability can be assessed from the specification of a program and experimentally shows that a program developed from a domain-testable specification takes less time to build and test than a program developed from a non domain-testable specification.

According to Binder [25] software testability can be defined as the relative ease and expense of revealing the software faults. A more testable system reduces the time and cost required to achieve the reliability goals. He mentions that software testability depends upon six primary factors: 1) characteristics of the representation, 2) characteristics of the implementation, 3) built-in test capabilities, 4) the test suite, 5) the test support environment, and 6) the software process in which the testing is performed. A reusable test suite and the assertions (preconditions, postconditions, and invariants) increase the testability whereas the user interface bindings reduce the testability. The testability can be assessed through a number of metrics, some of
which are: LCOM (lack of cohesion in methods), PAP (percentage public and protected), PAD (public access to data members), NOR (number of root classes), FIN (fan in), NOC (number of children), DIT (depth of inheritance tree), OVR (percentage of non-overloaded calls), and DYN (percentage of dynamic calls) etc. A high value of the metric indicates the lower testability and the lower value indicates the higher testability. Observability and controllability are two main key components of testability. One must be able to control the input and observe the output of a component to test it. Hence, the design for testability aims at removing the difficulties in the controllable input and the observable output. Because, if we cannot control the input, we cannot be sure what has produced a given output and if we cannot observe the output, we cannot be sure how a particular input has been processed to produce the output. Software testability is the result of good software engineering practice and better software process.

Khoshgoftaar [85, and 86] claims that static software measures take less computation than the dynamic software measures and has performed a case study of real time software to predict the testability of a module from the static measurements of the source code. He uses the definition given by Voas and Miller [136], where testability is a combination of the probability that 1) a particular location is executed, 2) the probability of a fault at a location, and 3) the probability that corrupted results propagate to the observable outputs. The neural networks are used to build the predictive model to measure the testability from the static software metrics. Testability is a dynamic quality attribute and it is difficult to measure it directly.
Neural networks are used as they are better technique to model the nonlinear relationships.

McGregor et al. [102] focuses on estimating the effort needed to test a class as early as possible in the development of an object oriented system. They have introduced the concept of visibility component of a method and define testability of a method as a function of its visibility component (VC). The testability of the class is calculated from the testability of its methods. The calculation of the testability requires complete and accurate specification documents. This measure of testability can be used to effectively estimate and schedule the resources during testing. Visibility Component (VC) is defined as the ratio of the number of inputs to the number of outputs, where the number of inputs is equal to the sum of the number of explicit and implicit input parameters, and the number of outputs is equal to the sum of the number of explicit and implicit output parameters.

Le Traon and Robach [132, and 133] have proposed a testability model to analyze the testability of data-flow designs based on the information theory. The testability model is a graph called the Information Transfer Graph (ITG) which can be applied to model the data-flow designs. ITG is used to analyse the data flow among the software components. An ITG is developed by analyzing a data model. The data flow metrics are derived from the Information quantity (IQ). Two data flow metrics: controllability and observability metrics are derived from the ITGs through the use of IQs. The controllability measures the IQ available on the input of a component from the inputs of the specification. The observability measures the IQ available to the
outputs of the specification from the outputs of a component. The controllability
metric for a given module, in the ITG is defined as:

\[
\text{Controllability} = \frac{\text{Maximum IQ received from flow on component inputs}}{\text{Total IQ that can be received by the component}}
\]

The observability metric for a given module is defined as:

\[
\text{Observability} = \frac{\text{Maximum IQ received on component outputs}}{\text{Total IQ the component can produce on its outputs}}
\]

Lin and Lin [148] define testability measures based on the data flow in the
program. The testability measures are defined in terms of the data flow metrics which
establish a relationship between the variables in the data flow graph of a program. A
number of data flow metrics are used which are: number of nodes (n), number of
inges (e), number of definition-use (def-use) pairs in the program etc. The testability
metrics are derived corresponding to all these data flow metrics as the inverse of the
data flow metrics, for example, the testability metric corresponding to the number of
nodes is:

\[
\text{Testability} \ (n) = \frac{1}{n}
\]

where n represents the number of nodes in the program data flow graph. The higher
the testability value, the easier it is to test a program. The authors claim that if these
data flow analysis based testability metrics are applied at the start of the software
testing phase then the testing resource allocation can be made much more effective.
The above metrics exploit the properties of the program structure which affect the
software testability. These metrics do not provide any relation between these
measurements and the number of faults, but can be used to identify the effort required for testing the software modules. These metrics are easy to compute as they require only a static analysis of the given program.

Jungmayr [127] provides a measure of the testability through the static analysis of the source code of software. He provides the testability metrics, which provide an idea of the effect of dependencies on the testability of the software. One of the testability metric is the average component dependency (ACD). ACD measures the average number of components that a given component depends upon directly or indirectly. The number of such components needs to be tested in case of a change in a given component. However, ACD is an example of a metric that only deals with the effect of dependencies with relation to a particular component.

Robach et al. [130] focuses on the testability analysis of the source code which is generated from the data flow designs. Testability is based on the controllability and the observability of the module for every data flow of the software. The controllability measures the information quantity available on the inputs of a module from the inputs of the software from the data flow. The observability measures the information quantity available on the outputs of the software from the outputs of a module. The analysis method transforms the source code, generated from the data-flow designs into the Static Single Assignment (SSA) form and then an algorithm is used to automatically translate the SSA form into a testability model, which is then used to calculate the testability measures for the source code. SATAN tool is used to analyse the testability of the generated code. Authors find that the analysis at the code level is more complex as compared to the design level, although
it provides more detailed measurement of the testability. This testability model can help to detect the parts of the design/code which cause a lack of testability and to compare the design testability with the code testability so that the coding process can be evaluated.

Baudry et al [12] suggests that only class diagrams and statecharts of UML are the main models that should be analysed for testability. In their study those parts of the software design are identified where complex interactions occur which cause problems in testing the software. They suggest that certain types of object interactions like when one object changes the state of another object through dependencies, make testing difficult. These object interactions are called testability anti patterns which are identified by using a class dependency graph (CDG). These object interactions are the places where the testability can be improved. Figure 2.1 shows the example of a problematic class interaction in which an object of class A can potentially interact with the object of class B either directly through the class association or indirectly through the inheritance hierarchy of the classes D and E. The number and complexity of the undesirable class interactions is identified from the class dependency graph.

Figure 2.1: Problematic Class Interactions
The measurement of the degree of complexity of these class interactions gives an estimate of the testability of the software.

Mouchawrab et al. [108] has presented a measurement framework for the testability of the object oriented software to facilitate the empirical research on testability. The framework helps in assessing the testability at the design level, particularly for Unified Modeling language (UML). A number of design attributes are identified which have an impact on the testability. Also, a number of measures are provided for each of these attributes. The cause-effect relationship between the design attributes and software testability is also described.

Gupta [59] has proposed a fuzzy model for an integrated software testability measure called testability index for object oriented software. The testability measure takes into account four design metrics: ACC (Average Cyclomatic Complexity), DIT (Depth of Inheritance), RFC (Response for a Class), and CBO (Coupling between Objects). These four metrics are the inputs for the fuzzy model which produces a testability index for the class. This model is validated using the Java classes. The testing time of java classes is computed and a correlation with testability is found. A high correlation between testing time and the testability measure is found.

Bruntink and van Deursen [32] predict the testability of a class in an object oriented system by statically analyzing the source code. Five case studies are used to perform an empirical study, which are tested using the JUnit framework. Object oriented metrics like number of children, number of attributes, and number of methods etc. are used to perform the analysis. The JUnit based test suite metrics used are: the number of test cases, and number of lines of code of the JUnit test class. A
correlation is found between the object oriented metrics and the test suite metrics, which provides an indication of the impact of the object oriented metrics on the number and complexity of the test cases.

Zhao [153] follows the definition of testability given by IEEE and uses the beta distribution to assess the software testability. It is theoretically proved that the beta distribution can be used to assess the value of the testing effort and test value also. Siemens programs are used to conduct the experiment and validate the results. The authors introduce the testing effectiveness information into the testability model and get prior distribution estimation for each (software, test criterion) pair. The validation of the beta distribution for the specific program and testing criterion pair is performed as below: an appropriate testing criterion's effectiveness measure is defined. Testing criterion's effectiveness information is introduced into the model. Then, it is proved that after incorporating effectiveness information, the distribution should provide reasonable estimate of the quantity and quality of testing. The effectiveness measure used is called fault detection probability (fdp) and is measured as the ratio of the number of suites that can detect Fault F to the total number of test suites satisfying the testing criterion.

Kansomkea et al. [129] states that testability is a quality attribute which, is used to assess the amount and difficulty of testing and also indicates the difficulty with which the faults can be revealed. They focus on the data flow analysis and follow the definition of testability which is measured as the probability that during testing the faults would be revealed. The component testability is measured by analyzing the execution probability and the propagation probability from the bytecode. The
execution probability is measured as the percentage of the faulty locations executed and the propagation probability is the percentage of faulty locations for which a given input causes an incorrect output. The definitions and uses of the class-component are used to find the testability. They provide a quantitative analysis of the testability of a software component with the help of the bytecode without requiring the source code of the component. This testability of a component is assessed through the testability of every location of the program. The testability of a given location is computed by calculating the probability that: 1) the location would be executed, and 2) the execution will reveal the fault if the location contains a fault. The importance of the testability analysis is that the developers can determine whether the testability of the component should be increased for reusing it.

Goel et al. [57] provides a testability framework for testing the object oriented software. They follow the testability definition based on the two concepts: observability and controllability. A controllability and observability testing tool (COTT) is provided which creates testable object oriented software. The COTT environment of testing is based upon the probe based observability and controllability mechanism. The user program is instrumented with the probes and the control commands. The COTT tool is composed of two main parts: instrumentation subsystem and testing subsystem. The instrumentation subsystem inserts the probes and the control commands, and preprocesses user program to gather the information about the inserted probes and the control commands. The preprocessed user program is fed to the testing subsystem. The testing subsystem executes, monitors, and analyzes the test.
Khan et al. [83] has proposed and validated a metrics based model (MTMOOD) for object oriented design testability through the empirical validation on industrial projects. MTMOOD is based upon the generic quality model of Dromey and involves the following three steps: 1) identify the product properties which influence the testability of the object oriented software, 2) identify the object oriented design metrics, and 3) find a way to link them. The MTMOOD model incorporates four product properties which influence testability. These are: encapsulation, inheritance, coupling, and cohesion. Four new metrics have been defined which are: encapsulation metrics (ENM), reuse metrics (REM), coupling metrics (CPM), and cohesion metrics (CHM). All of these metrics can be calculated from the design information.

Sohn [125] has proposed a new measure of software testability based upon the concept of entropy. The testability is defined as the probability that the tests will detect a fault, provided a fault exists in the program. The fault tree analysis is used to capture the presence of a fault, its propagation, and its effect on the output failure. The testability is a combination of two components, one is the output failure probability which is calculated from the fault tree with an assumption that the basic statement fault probabilities are same and the other is the entropy of the importance of the basic statement faults. The testability measure has been applied to the safety modules and is verified through the branch coverage testing and random testing. Authors claim that the entropy based testability measure can be used to select the output variables which can lead to higher fault detection and can also be used for finding the modules which are more prone to the hidden faults. This measure takes into account the internal complexity of the program and does not require the
execution of the program. This measure can be easily estimated in the cases where software safety analysis is required using the fault tree analysis method.

Gao [52, and 54] mentions the issues related to the testability of the components in the component based software engineering paradigm. The component testability is defined using five factors: controllability, observability, understandability, traceability, and the test support capability. The design and definition of a component's interfaces (such as incoming and outgoing interfaces) as well as the component traceability affect the observability of a component. The component traceability is defined as the extent to which its built-in capability can track the status of its attributes and behaviour. It has two parts: 1) behaviour traceability, and b) trace controllability. Behaviour traceability is defined as the degree to which a component helps tracing of its internal and external behaviours and trace controllability is the extent of the control capability in a component to facilitate the customization of its tracking functions. Component controllability has three different aspects: a) behaviour control, b) feature customization, and c) installation and deployment. Component understandability depends upon the level of component information provided and the way it is presented. Gao [53] describes the component testability through a component testability analysis model.

Bruce [31] claims that the software testability is one of the ways of fault detection. He has provided various factors of software testability and a testability model for object oriented software based on the fault failure model of software testing. The definition proposed by Voas and Miller [81] is followed, which defines software testability as the probability that it will fail on its next execution during
testing if the software includes a fault. The factors that affect the software testability are categorized into three types which are: structural factors, communication factors, and inheritance factors. The structural factors focus on the coding structure of the class and involve three sub factors which are: internal testability of a method, number of methods in a class, and the cohesion among methods in a class. The communication factors focus on the coupling between the classes. Three inheritance factors which can affect the testability of an object oriented program are depth of inheritance tree, the number of children, and the number of disjoint trees.

Alvi [7] has performed an empirical study to determine the testability of a package using Eclipse as the source system. The package testability is determined in terms of the testing effort of the package which is assessed through the JUnit test classes. The source java package and corresponding test packages of Eclipse are used for the analysis. The package level metrics which are used as the independent variables are: Concrete class count (CC), Abstract class count (AC), Number of classes and interfaces (NOCl), Efferent Couplings (Ce), Afferent Couplings (Ca), Abstractness (A), Instability (I), Distance from the Main Sequence (D), Single Line of Code (SLOC), and Number of Packages (NOP). The dependent variables used are Single Lines of Code for Package (SLOCP), Number of test cases/Classes in a Package (NOCP), and the number of sub packages in a test package (NOPt). The value of package level metrics and the test metrics are calculated using various metrics tools which can be used as Eclipse plugins. Spearman’s rank order correlation coefficient is used to predict the testing effort from the package level metrics. The
package level metrics Ce, I, and NOP are found to have a significant correlation with test level metric NOPt.

Gupta and Sinha [56] define software to be testable if it can be tested easily, systematically, and externally at the interface level (i.e. without modifying the code). They claim that a software has neither the controllability property nor the observability property and these measures can be built systematically as an integral part of the software during the earlier phases of software development (i.e. design, detailed design, and coding phases, also called as testability incorporation phases). Test plan is used to incorporate the testability measures in the software. The test plan for each of the pre-testing phases of the software development is an important input for the next phase to make the software testable. Verification test plans of design, detailed design, and coding phases are checked for testability and may undergo changes to incorporate the testability measures. The activity of building testable software has a significant effect on the software development model which leads to a new model called 'Testable Software Development Model'.

Lin and Huang [96] claim that testing polymorphism in object oriented systems is most difficult part of testing as it easily hides faults from testing. A polymorphism RATO (Reference attachment to object) model is provided to accurately understand the polymorphic behaviour and estimating the testability at the design stage. This model also helps in improving the testability by considering the redesign.

Singh and Saha [124] have performed an empirical study to determine the relationship between the object oriented design metrics and the test metrics and to ultimately find the effect of object oriented metrics on the testing effort and on the
testability. The relationship between java classes and their JUnit test classes in four large Java systems is analyzed. A significant correlation is found between most of the object oriented metrics and the test metrics. The results of correlation show that out of four size metrics, three metrics (LOC, NOM, and WMC) are correlated to the testing effort. It is shown that as the size of the software increases, the testability decreases because the testing effort increases. Out of four coupling metrics, three coupling metrics (RFC, CBO, and MPC) are highly correlated to the test metrics. Two cohesion metrics: ICH and LCOM are found to be correlated to the test metrics. Out of 14 metrics, 8 metrics are found to be highly correlated to the testing effort and hence, to the software testability.

2.6 Testability Improvement techniques

In this section we consider various techniques which are used to improve the software testability.

Baudry [13] has proposed a technique to improve the testability by augmenting the designs with the constraint information so that the relationships between classes can be implemented clearly. The UML stereotypes help in including the constraint information in the design which helps in avoiding any ambiguity in the design.

Briand et al. [29] has given an approach to increase the testability in which the contracts (operation pre and post conditions, and class invariants) are instrumented. Authors claim that the instrumented contracts increase the observability of the system by increasing the probability that a fault would be detected upon the execution of the
<table>
<thead>
<tr>
<th>Author</th>
<th>Development stage</th>
<th>Procedural/Object oriented software</th>
<th>Tools</th>
<th>Concepts used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Woas and Miller [81]</td>
<td>Specification and design</td>
<td>Procedural Software</td>
<td></td>
<td>Information loss</td>
</tr>
<tr>
<td>Bache and Mullerburg [9]</td>
<td>Source code, unit level</td>
<td>Procedural Software</td>
<td>QUALMS</td>
<td>Fenton–Whitty theory, control flow graphs</td>
</tr>
<tr>
<td>Jungmyr [127]</td>
<td>Source code</td>
<td>Software components</td>
<td></td>
<td>Dependencies among Components</td>
</tr>
<tr>
<td>Brauntink [32]</td>
<td>Source code</td>
<td>Object oriented Software</td>
<td></td>
<td>Empirical study, JUnit Test metrics</td>
</tr>
<tr>
<td>McGregor et. Al. [102]</td>
<td>Design level</td>
<td>Object oriented software (Class level)</td>
<td></td>
<td>Visibility Component</td>
</tr>
<tr>
<td>Woas and Miller [138]</td>
<td>Source code</td>
<td>Procedural software</td>
<td>PISCES</td>
<td>Mutation and sensitivity analysis, black box testing,</td>
</tr>
<tr>
<td>Khan [83]</td>
<td>Design level</td>
<td>Object oriented software (class level)</td>
<td></td>
<td>Design metrics based model</td>
</tr>
<tr>
<td>Freedman [49]</td>
<td>Specification level</td>
<td>Software Components</td>
<td></td>
<td>Observability and controllability</td>
</tr>
<tr>
<td>Binder [25]</td>
<td>Design Level</td>
<td>Object oriented software</td>
<td></td>
<td>Observability and controllability</td>
</tr>
<tr>
<td>Baudry et al. [12]</td>
<td>Design Level</td>
<td>Object oriented software</td>
<td></td>
<td>Class diagrams and statecharts of UML</td>
</tr>
<tr>
<td>Le Traon and Robach [133]</td>
<td>Design level</td>
<td>Software components</td>
<td></td>
<td>Information Theory, Data flow designs</td>
</tr>
<tr>
<td>Lin and Lin [148]</td>
<td>Source Code</td>
<td>Procedural software</td>
<td></td>
<td>Data flow graph</td>
</tr>
<tr>
<td>Muchawarab et al.</td>
<td>Design level</td>
<td>Object oriented</td>
<td></td>
<td>UML Models</td>
</tr>
</tbody>
</table>
test cases. A case study is used to demonstrate that the contracts detect a greater number of failures than a system in which there are no instrumented contracts.

Nancy [46] considers the testability of architecture and proposes that architectural decisions should consider the test strategies chosen, which means that, the testability of architecture depends upon the architectural pattern and the testing technique chosen. Some testing strategies are better for testing certain architectural patterns than others.

Lamoreaux [91] has developed a software development process to increase the testability of the software products. Through this process the testability can be built into the product during the product development cycle which leads to a maintainable software product. This process consists of simple eight steps which are: 1) determine what is testable, 2) determine which objects or classes can be identified by the automation tool, 3) determine what can be scripted manually, 4) document the non testable components, 5) determine if a reusable method can be created to test a specific control that is used many times in the software application, 6) ensure that the solution will not interfere with the functionality of the software, 7) develop the solution, and 8) validate the solution.

A number of approaches based on the testability assessment are described above. The Table 2.2 shows the summarization of these approaches. The Table 2.2 shows that different approaches are applicable at different levels of the development stage: specification level, design level, and implementation (code) level. Some approaches are defined for the procedural software and some are defined for the object oriented software. A number of tools have also been developed to assess the
software testability. There are a number of empirical studies for the assessment of testability. The empirical studies mainly focus on the code written in C and Java languages. There are different studies which focus on the code level testability and the design level testability. Also, various studies have emphasized the importance of software testability throughout the software development life cycle. A few empirical studies exist which necessitate performing other empirical studies covering the software developed using different languages and different platforms.

2.7 Discussions

This chapter presents OO testing techniques developed in the literature. A lot of issues and problems of OO testing have been solved through these techniques. An important issue is to improve the testability of the OO software. Hence, an extensive literature survey of software testability is also presented. It is seen that software contracts improve the testability of the software. Hence, the testing techniques developed using software contracts will improve the testability of the software. Also to improve the testability of the software it is necessary to assess the testability of the software. In the literature, a number of metrics (measures) have been developed to quantify the software testability. The software testability can be improved with the help of the measurements provided by various researchers by maximising the metric value. Most of the researchers advocate the consideration of software testability throughout the development of the software, i.e., from the requirement specification to the implementation. Since testability helps in planning the testing activities and the
number of empirical studies are very few. Hence, there is a need to perform more empirical studies to assess the testability of the object oriented software.