Chapter 2

Basic Concepts

By using object-oriented paradigm, software reusability, extendibility and reliability will be increased, provided that the software systems are tested properly [12]. In this chapter, we discuss some preliminary concepts and general definitions that form the basic and background knowledge of our research. We also discuss some basic definitions of software testing and the different levels of testing object-oriented software. In the preliminary part, we focus on generating the test cases from the behavioral UML 2.0 models. Before going to put the first step, we discuss some other aspects those are directly relevant to our research work. Our research primarily focuses on generating test cases from UML behavioral design models. Therefore, we give an overview of relevant UML models. Further, our work in this thesis also encompasses the concept of test case generation using program slices. We therefore, present a brief discussion on slicing. For the sake of conciseness, we do not aim to present a detailed description of the background theory. Instead, we provide a brief introduction aimed at highlighting the basic concepts and definitions.

This chapter has been organized as follows. In Section 2.1, we discuss the concepts about faults, failures and errors. Then, in Section 2.2, we discuss about basic concepts and definitions of testing. Section 2.3, provides the different testing procedures. An introduction to object-oriented software testing is presented in Section 2.4. In Section 2.5, we present the methods and levels of testing. In Section 2.6, we discuss the overview of UML diagrams, which are relevant to our research work. In Section 2.7, we highlight the different test adequacy criteria. We present the different fault models in object-oriented system in Section 2.8. In Section 2.9, we discuss the fundamental concepts of program slicing. We describe the basics of test
case prioritization in Section 2.10. Finally, Section 2.11 summarizes the chapter.

### 2.1 Fault, Failure & Error

In this section, we briefly explain the concepts of fault, failure and error, in the programs.

The execution of tests on a system under test (SUT) can result an unexpected behavior. According to Hopper [46], the first such unexpected behavior was caused by a moth in a relay. That is why it is common to speak of bugs. This term, however, does not describe the different stages of fault, error propagation, and failure detection appropriately. In the following, we introduce the fault/failure model as presented in [77, 78].

**Fault:** A static defect in a system is called a fault. A static defect is, i.e., a wrong expression in the source code, misinterpreting a requirement, forgetting a condition, or simply mistyping which is caused by human being. It does not make any effect as long as the fault is just exists in the system without being executed. When the instruction is executed, then the fault is activated. This activated fault can result an error.

**Error:** An error is a wrong internal state of a running system. A wrong internal state of a system can be, e.g., an erroneous program counter or a faulty attribute value. If such wrong values influence the observable behavior of the SUT, the error is said to be propagated to the outside. An error that is propagated to the outside can result in a failure.

**Failure:** A failure is an observable deviation of the actual from the expected behavior of a system.

Failures can be detected directly by test cases. Figure 2.1 shows one possible way from a fault to a failure with fault activation and error propagation. What scenarios

![Fault Activation Error Propagation Failure](image)

**Figure 2.1:** Relationship between fault, failure & error
2.2 Basic concepts and Definitions of Testing

In this section, we present few basic terminologies needed to discuss the research addressed in this thesis.

**Test Specification:** Test specification gives the aspects of what scenarios will be tested, how they will be tested and how often they will be tested or specific types of faults to be detected in a software product. A test specification is therefore a specification of the kind of testing to be applied to a system rather specifying the exact test data for carrying out the test and their exact sequence of input or the expected output data. Test specifications are often associated with coverage criteria known as test adequacy criteria [39, 74].

**Test Scenario:** Test scenarios are test cases or test scripts, and the sequence in which they are to be executed. Test scenarios are test cases that ensure that all flows are tested from start to end. Test scenarios are independent tests, or a series of tests that follow each other, where each of them is dependent on the output of the previous one. Test scenarios are prepared by reviewing functional requirements, and preparing logical groups of functions that can be further broken into test procedures. The terms “test scenario” and “test case” are often used synonymously.

**Test case:** A test case is a triplet \([I, S, O]\), where \(I\) is the input to the system, \(S\) is the state of the system, at which the data is input, and \(O\) is the expected output of the system [73].

**Test suite:** A test suite is a collection of all generated test cases from the system under test (SUT) with which a given software product is to be tested. The execution of a test suite is considered as a test run.

**Minimal test suite:** It is a carefully designed set of test cases such that each test case detects different types of errors [73].
2.3 Testing Procedures

Keeping the facts of fault, failure and errors in mind the test engineer proceeds to test a program out of several interrelated programs of a software product which involves the following activities.

**Object Identification:** The main purpose of this activity is to identify the object which involves in this particular process. Here the objective is to design a set of test cases which ensures that the software supports the object.

**Input data Identification:** The intent of this is to identify the data which are to be tested, which meets the requirements of the specification. Here the objective is to prepare a group of data set for all possible test cases.

**Expected output preparation:** The logic behind this is to prepare all expected outputs which will come from the program for each designed test case. Here the objective is to prepare a list of outcomes which meet the requirements.

**Actual output Generation:** The main idea is to execute the program, to pass the prepared list of inputs to the program and to record the actual output generated. Here the objective is to record the actual outcome from the program.

**Test result Analysis:** The primary goal is to compare the test results of each actual output with the expected output. Here the objective is to declare a test verdict for the program, whether it is passed, fail or inconsistent.

2.4 Object-oriented Software Testing

Object-oriented software testing (OOST) [22] is an important software quality assurance activity to ensure that the benefits object-oriented programmings (OOPs) will be realized. Here, we discuss the different levels of testing associated with OOPs.

1. Intra-method testing: Equivalent to unit testing for programs. Tests are designed for individual methods.

2. Inter-method testing: Tests are designed to test interactions of the methods.
3. Intra-class testing: Tests are constructed for a single entire class, usually as sequences of calls to methods within the class.

4. Inter-class testing: Tests are equivalent to integration testing. But, it is meant to test a number of integrated classes at the same time.

The first three variations are of the unit and module testing type, whereas inter-class testing is a type of integration testing. The overall strategy for object-oriented software testing is identical to the one applied to conventional software testing, but differs in the approach it uses. We begin testing in small and work towards testing in the large. As classes are integrated into an object-oriented architecture, the system as a whole is tested to ensure that errors in the requirements are uncovered.

2.5 Methods and levels of testing

The test can be classified into two broad categories or methods, i.e. black box testing and white box testing. The importance of these approaches is discussed in the following subsections.

2.5.1 Black-Box Testing

Black-box testing technique is one of the major techniques in dynamic testing for designing effective test cases. It is also known as functional testing technique. This technique is considered only the functional requirements of the software or module. Here, the structure or the logic of the software is not taken into consideration and the program is treated as a black-box. A test engineer is concerned only with the functionality and the features found in the program’s specification [91]. Black-box testing is also called as specification-based testing or behavioral testing. In the black-box approach, test cases are designed using the functional specification of the software, i.e. without any knowledge of the internal structure or implementation detail of the software [29, 73, 91]. All test cases are derived from the specification. The tester treats the software under test as a black box; only the inputs, outputs and specification of the program are assumed to be visible. The functionality of the software is tested by observing the outputs corresponding to inputs. There are various methods for the testing of the software using black-box techniques. The two main approaches are boundary value analysis (BVA) [54, 88] and equivalence class partitioning (ECP) [54].
2.5.2 White-Box Testing

White-box testing techniques are also known as structural or glass-box testing techniques. In this type of testing the entire structure, design, and code of the software product has to be examined. By using this testing, one primarily examines the source-code with a focus on control-flow and data-flow. Control flow refers to the flow of control from one instruction to another in a number of ways [24]. White box or structural testing requires that inputs be based solely on the structure of the source code or its data structures. In this testing, the structure and flow of the software under test are visible to the tester. White box testing is also known as code-based testing. Designing white-box test cases requires knowledge of the internal structure of the software under test. Testing plans are made according to the details of the software implementation, such as programming languages, logic, and styles. Test cases are derived from the program structure [12]. Structural testing considers the code of the program, and the test cases are created on the basis of the logic of the program so that every element in the logic will be covered. Here the primary goal is to test the entire logic of the system. Some of the basic forms of white-box testing are statement coverage based testing, branch coverage based testing, path coverage based testing.

Each testing technique has its own advantages and disadvantages. All the bugs from a software product cannot be removed by using only black box or white box technique. For all possible inputs, a structural testing technique cannot detect all faults if there are some missing paths in a program. Similarly, without the knowledge of structural details, many faults will go undetected. Hence, a combination of both structural and functional testing techniques must be used in testing. Black-box testing determines whether the program meets its functional and non-functional requirements without regard to its implementation. Functional test cases are constructed based on the specification of the unit. Coverage methods are used to determine the percentage of the class specification that is covered during testing. Black-box testing tests whether the unit works according to the specification, but does not assure whether testing of all parts of the code has been exercised. White-box testing tests whether the code that exists works properly, but does not test for forgotten code or functionality. As white-box and black-box testing cannot detect maximum number of faults, a mixed approach known as gray-box testing is used to test object-oriented software [1].

**MC/DC Testing:** MC/DC stands for Modified Condition / Decision Coverage
testing. It is a white box testing coverage criterion. It is a kind of predicate coverage where a condition is a leaf level Boolean expression and a decision controls the program flow. The main idea is that each condition must be shown to independently affect the outcome of a decision, i.e. the outcome of a decision changes as a result of changing a single condition [45]. MC/DC is a stronger criterion than statement and branch coverage. Condition testing is stronger testing than branch testing and branch testing is stronger than statement coverage testing. Let us illustrate it with an example.

By Considering three Boolean conditions A, B, C with the expression If \((A \text{ and } (B \text{ or } C))\), we prepare a truth table for the expression. Using the Table 2.1, we have seen that in TC 1, where all the values of A, B, C are True(T) and the Result is also True(T). By considering this statement if we change the value of A to False(F) in column 1 then result is automatically changed to False (F) which is given in TC 5 i.e. (FTTF). Similarly, in TC 2, the values of A,B,C and result are T,T,F and T respectively. By changing the value of A to False(F) it will give rise to another test case given in TC 6. So, we can pair TC 1 with TC 5 and TC 2 with TC 6. Similarly other MC/DC adequate can be calculated and other pairing of test cases can be made. All pairing of test cases are shown in Table 2.1.

Table 2.1: Truth table for the expression \((A \text{ and } (B \text{ or } C))\)

<table>
<thead>
<tr>
<th>TC#</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Result</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>7</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To determine the minimal set of test cases it should be converted to three necessary criteria [45]. The minimal test cases are

- 2,3,4,6
- 2,3,4,7
2.5.3 Gray Box Testing

By combining some features of white-box and black-box techniques, a new approach is designed known as gray-box method [79]. Gray-box testing is based on low level design models representing the expected structure and behavior of the software. It extends the logical coverage criteria of white box method such as path coverage [79]. The design specifications preserve the essential information from the requirement, and are the basis of the code implementation, which provides structural information to a certain extent. UML based testing methods can be considered as gray box testing methods.

2.5.4 Different levels of testing

Object-oriented programs can be tested at four levels, namely algorithmic level, class level, cluster level and system level [21]. To avoid confusion, algorithmic testing, unit testing, integration testing, system testing is same as compared to method level, class/intra-class level, cluster/inter-class level and system level testing respectively.

**Algorithmic Level:** At the algorithmic level, individual methods are tested separately, so that conventional testing techniques can be applied without much problem [54]. The algorithmic level involves the churning of the routine (method) on some data.

**Class Level:** In object-oriented software testing, class is the basic unit of testing [25]. A class is composed of attributes and methods. Though a class is a complete unit, so it can be tested independently. Therefore, it is also called unit testing. Testing of the class involves testing of the methods defined in the class and their impact on the state of the class/objects. The class level testing is concerned with getting the interaction between the attributes and methods of a particular class. Here state of the object must be considered. At the class level, the objective is to verify the integrity of a class by testing it as an individual entity [25]. This focuses on testing methods and their interaction within a single class based on class diagrams and state charts. The type of method (including constructor, modifier, selector, iterator methods) and the type of class (base class or derived class) are used beside other class characteristics to select the appropriate test approach [17].
2.6 Relevant UML Diagrams

**Cluster Level:** The cluster level is related to the integration of classes [12]. Testing at cluster level considers interactions among a group of cooperating classes. Cooperating classes of objects make up a cluster. As the functionality of individual classes has already been verified, inter-class method invocations are tested. This focuses on the interaction between closely related classes. Especially, during cluster level testing, we deal with existing aggregation and association relationships as well as with inheritance, polymorphism and exception handling. They include state-based testing, event-based testing, fault-based testing, deterministic and reachability techniques and formal and semi-formal techniques [29]. The test case design is based on sequence diagrams, collaboration diagrams and activity diagrams [120]. The basic motivation of cluster testing is to validate that the classes in a subsystem are collectively operable, i.e. the clusters of classes provide the intended functionality when made to interact with each other.

**System Level:** Testing at system level involves testing of an entire system in aggregate. System testing is concerned with testing an entire system based on its specifications, and involves several activities such as functional testing (testing from behavioral descriptions of the system) and performance testing (response time and resource utilization) [21].

In other words, the implementation under test is compared to its intended specification. System testing tests all the subsystems (or components) together, seen as a single system to identify faults with respect to the requirements and design goals identified in the analysis and design specifications, respectively. System testing ensures that the complete system complies with the functional and nonfunctional requirements. System testing of object-oriented software is same as conventional software system testing [106]. At the system level, interactions among clusters are tested. System-level testing is concerned with the external inputs and outputs visible to the users of a system.

2.6 Relevant UML Diagrams

In this section, we briefly describe about Unified Modeling Language (UML) and the relevant diagrams used in our work.

The UML is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a stan-
standard way to write a system’s blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schema, and reusable software components. UML has become a de facto standard for object-oriented modeling and design [16, 18]. The Unified Modeling Language (UML) merges three different object-oriented development methods: Object-Oriented Analysis and Design with Application (OOADP) by Grady Booch [18], the Object Modeling Technique OMT by James Rumbaugh [17] and Object-oriented Software Engineering (OOSE) by Ivar Jacobson [16].

The Object Management Group (OMG) issued a request for a standard approach to object-oriented modeling in 1996. Booch, Jacobson and Rumbaugh produced a proposal and introduced the Unified Modeling Language to meet the requirements. The approach was adopted by the OMG and became a standard in 1997. Later, International Organization for Standardization (ISO/IEC 19501:2005) has also adopted UML version 1.4.2 as a standard [16, 18]. In order to meet the needs of software development industry, UML notation was designed to go beyond programming languages, operating systems, application domains, and software life-cycle phases [73].

Since its adoption, UML has widely been embraced by the software engineering community. A large number of organizations across the globe could productively use UML in innovative ways to advance their design and development [73]. UML has been widely accepted also for documenting design models. Of late, researchers are focusing their attention on UML models as a source of information for test case generation, which, if satisfactorily exploited, can go a long way in reducing testing cost and effort and improves software quality [73]. UML-based automatic test generation is a practically important and theoretically challenging topic and receiving increasing attention from researchers.

### 2.6.1 Views of UML

The different views that can be modeled using UML are: users’, structural, behavioral, implementation and environmental views [73]. The UML user’s view consisting of use case diagrams is mainly used for requirement-based testing and high level test design [16, 99]. In [4], the UML class diagrams along with state machine diagrams have traditionally been used for testing object-oriented programs at the unit level. On the other hand, interaction diagrams (sequence and collaboration diagrams) along with class diagrams have been used for designing test cases
2.6 Relevant UML Diagrams

to test interactions among a group or cluster of classes [3]. Behavioral models, such as, state machine diagrams are used to test the state dependent behavior of classes [120]. The behavioral view has also been recognized to be useful to test a system at class, cluster and system levels [12]. UML models are an important source of information for test case design. Traditionally, data flow and control flow information are obtained from the source code [82]. UML behavioral diagrams can provide both data flow and control flow information. Data flow and control flow information have significant bearing on test case generation. Hence, using behavioral UML diagrams, it is possible to generate class or cluster level test cases even before any code is written. Several issues must be resolved before UML models can be effectively applied across the testing process. For example, the models of the development process are abstract and typically lack several details present in the code and therefore are inadequate for comprehensive testing. To redress this situation, UML 2.0 adds several new capabilities to UML 1.x [1]. It has improved its precision and expressiveness to model large and complex architectures and this removes some of the major problems in test case generation.

2.6.2 UML 2.x

UML 2.0 defines fourteen types of diagrams, as shown in Figure 2.2. UML 2.x diagrams are divided into three categories: out of which seven diagrams represent static application structure, three of them represent general types of behavior, and four represent different aspects of interactions.

Figure 2.2: UML 2.0 Diagrams
1. The *structural diagrams* include the Class Diagram, Component Diagram, Composite Structure Diagram, Deployment Diagram, Object Diagram, and Package Diagram.

2. The *behavioral diagrams* include the Use Case Diagram, Activity Diagram, and State Machine Diagram.

3. The *interaction diagrams* have a further classification and all these derived from the Behavior Diagram, includes the Sequence Diagram, Communication Diagram, Timing Diagram, and Interaction Overview Diagram.

We review some of the UML [92, 116] diagrams that have been used either directly or indirectly in the rest of the thesis. UML is a visual modeling language that can be used to specify, visualize, construct, and document the artifacts of a software intensive system [16]. Unified Modeling Language (UML) has emerged as the industry standard for modeling software systems and has received significant attention from researchers as well as academia. A brief description of some important UML diagrams which are used in our proposed approaches, is given below.

### 2.6.3 Use Case Diagram

A use case diagram consists of four basic components: use cases, actors, associations and relationships. A brief description of these components is presented in the following.

**Use case**: Use case diagrams are considered for high level requirement analysis of a system. A use case represents a distinct piece of functionality of a system. It comprises of different possible sequence of actions and interactions between the user and the system under discussion. Each specific sequence of actions and interactions in a use case is called a scenario. A collection of scenarios for a use case constitutes all the defined ways the use case may be used.

**Actor**: An actor can be a human user, organization, some internal applications or may be some external applications that invokes the services of the use cases.

**Association**: An association between an actor and a use case indicates that the actor and the use case somehow interact or communicate with each other. An
2.6 Relevant UML Diagrams

association essentially represents communication between an actor and a use case. In UML use case diagrams, a use case is represented by an oval shape with the name of the use case as a label. A stick diagram represents an actor. An association between a use case and an actor is represented by a straight line as shown in Figure 2.3. A use case diagram models the users’ view of a system. In other words,

![Image](https://via.placeholder.com/150)

Figure 2.3: An Example of a Use Case diagram showing Use cases, Actors and Association for an ATM System

the functionalities provided by the system to the users are depicted in a use case diagram. Use case diagrams capture the different ways in which a system can be used by its potential users. Use cases describe the functionality of the software as seen by actors. An actor represents a role played by a user or another system that interacts with the system. There can be one or more use cases to describe the functionalities of the system. Each use case specifies a different usage of the system. By creating a use case view of a system, developers are motivated to design and build the software from a users’ point of view. Actors always model entities that are external to the system [25]. Figure 2.3 represents a sample use case diagram for an ATM system.
2.6.4 Activity Diagram

Activity diagrams describe the work flow behavior and dynamic aspects of the system. These are similar to state diagrams because activities are the state of doing something. Activity diagram emphasizes the path of concurrent or sequential control from activity to activity. The activity diagram describes about the ordering of the activities. The easiest way to visualize an activity diagram is to think of a flowchart of a code. In the general activity diagram describes the internal behavior of an operation. Activity diagram can show activities that are conditional or parallel.

![Activity Diagram Example](image)

Figure 2.4: An example of activity diagram for patient management system

Activity diagram shows how objects behave or how they collaborate within the system. Diagrams are read from top to bottom and have branches and forks to describe the condition and parallel activities. The edges represent sequences of activities including the control flows and message flows. The flows may be sequential, branched or concurrent. The flows are controlled by forks and joins. The different activities/actions are denoted by rounded corners. Flow/transitions are represented by arrowheads. The decision is shown as a diamond shaped boxes with one incoming
flow with multiple labeled outgoing flows. Forks are denoted by a synchronization bar with one incoming flow and multiple outgoing flows. Similarly, join is denoted by multiple incoming flows entering to a synchronization bar with one outgoing flow. The merge is represented as many concurrent flows are entering into a single point. Table 2.2 describes the control nodes and their actions used in the activity diagram. Figure 2.4 shows a UML activity diagram for a Book Order System.

### 2.6.5 Sequence Diagram

A sequence diagram is a UML structural diagram that provides a view of the chronological sequence of messages between objects or classifier roles that work together in an interaction or interaction instance. A sequence diagram consists of a group of instances, which are represented by lifelines, and the messages that they exchange...
Figure 2.5: An example of a sequence diagram for controller video game during the interaction. UML sequence diagrams are used to show the flow of functionality through a use case. They emphasize the ordering of messages and show the chronological sequence of messages, their names, responses and their possible arguments. A sequence diagram is formed by placing the objects that participate in the interaction at the top of the diagram, across the X axis. Typically the object that initiates the interaction is placed at the left, and increasingly more subordinate objects are placed to the right. The messages that these objects send and receive are placed along the Y axis, in order of increasing time from top to bottom. This gives the user a clear visual cue to the flow of control over time. An example of a
sequence diagram for a controller video game is shown in Figure 2.5.

UML 2.0 permits more complex interactions that can be created with “combined fragments”. A combined fragment consists of one or more interaction operands. An interaction operator specifies the purpose of the fragment. In UML 2.0, there is a provision of 12 different types of interaction operators. We briefly discuss only those interaction operators which we have used in our research work.

- **Alternatives (alt)**: It provides a choice or alternatives of behavior, out of which only one will be chosen. The interaction operands are evaluated on the basis of specified guard expression.

- **Option (opt)**: It defines an optional interactions segment. The model for an opt combined fragment looks like an alt that offers only one interaction.

- **Break (break)**: It is an alternative flow of message sequence. It operates on a single operand and the flow is terminated at that operand. *item Parallel (par)*: It supports concurrent processing. It shows that the associated interaction may be merged and executed in parallel.

- **Loop (loop)**: It model a series of message exchanges that are to be executed some number of times.

Interaction constraints can guard each interaction operand. Messages on their own cannot cross the boundaries of combined fragments, they need a *gate* which links the two parts of the message. Through the use of combined fragments, the understanding of the number of traces will be more easier. Figure 2.5 also shows the operators loop, alt, opt and ref in the combined fragments.

### 2.6.6 Interaction Overview Diagram

An interaction overview diagram (IOD) is a UML behavioral diagram that defines interactions and is a variant of the activity diagram, which emphasizes the high-level control flow. Interaction overview diagrams illustrate an overview of a flow of control, in which each node can be an interaction diagram. UML IOD combine elements of activity diagrams with sequence diagrams to show the flow of program execution. You can also use an interaction overview diagram to de-construct a complex scenario that would otherwise require multiple if-then-else paths to be illustrated as a single sequence diagram.
The interaction overview diagram tool bar contains icons for initial node, final node, decisions, merges, forks, and arrows, but no additional elements that could create invalid UML syntax. Control flow arrows will connect existing sequence diagrams and interactions in your model to illustrate execution flow. In an early development iteration you can quickly construct incomplete sequence diagrams to act as placeholders in the interaction overview diagram while you design the overall application flow. This functionality lets you design the high level program flow before all the details of each individual sequence are completed. You can complete each sequence diagram later, or even delegate sequence diagrams among team members.

Interaction overview diagrams illustrate an overview of a flow of control, in which each node can be an interaction diagram. The nature of the IOD is used to describe the high level of abstraction of the components within the system. The IOD can show dependence between the important sequences of a system, which can
be presented with an activity diagram. The used notation incorporates constructs from sequence diagrams with fork, join, decision and merge nodes from activity diagrams. IODs are special kinds of activity diagrams where the activity nodes are interactions and the activity edges denote the control flow of the interaction. According to the specification of UML 2.x the object flow cannot be represented by an IOD. Figure 2.6 shows the IOD of the Book issue use case of Library Information System (LIS). More formally, an IOD can be defined by the tuple $\text{IOD} = < n_0, N_f, B, D, I, E, Ed >$ where:

$n_0$ = initial node.

$N_f = n_{f1}, \ldots, n_{fn}$ is a set of final nodes.

$I = I_1, \ldots, I_n$ is a set of interaction nodes.

$B = b_1, \ldots, b_n$ is a set of join/fork nodes.

$D = d_1, \ldots, d_n$ is a set of decision/merge nodes.

$E = e_1, \ldots, e_n$ is a set of edges connecting the IOD nodes.

$Ed: n_0 \cup N_f \cup I \cup B \cup D \times n_0 \cup N_f \cup I \cup B \cup D \rightarrow E$ is a function which connects two IOD nodes by an edge.

### 2.7 Test Adequacy Criterion

A fault normally gets exposed only on certain inputs, whereas for several other inputs it may remain concealed. Hence, testing the effectiveness of a software product is as important as designing the test cases [29]. On the other hand, exhaustive testing is impractical from the point of view of cost and time [29]. Even for small programs the input domain can be very large. It is often important to know how thoroughly the software product has been tested. Certain criteria that can be used to assess the potential of test cases in revealing faults are known as test adequacy criteria.

Statement coverage is the most common example of a test coverage criterion. As per statement coverage criterion, each statement in a program must be executed at least once during testing. In other words, a test adequacy criterion (or coverage criterion) can be used to determine how effectively a given test suite tests a software. Generating test cases that satisfy a test adequacy criterion is often not simple and certainly not that easy to automate [8]. Even generating test cases for achieving statement coverage in large programs is difficult, as we need to ensure that the set of test cases has to cover each and every statement. Many times, test case design
techniques are to a large extent influenced by some test adequacy criterion.

Further, the effectiveness of a software product is solely determined by the quality of the test cases prepared and used to perform testing. A fault might get revealed only for certain types of inputs, whereas other inputs may fail to expose them. Therefore, test cases are often generated based on certain criterion. This criterion is known as test adequacy criterion (or coverage criterion). A test adequacy criterion is often represented as a predicate that defines properties that must be covered if the test is to be considered adequate with respect to the criterion [28]. A test adequacy criterion helps in defining test objectives or goals that are to be achieved while performing a specific software testing. For example, branch coverage requires that every branch in a program under test is to be exercised by at least one test case. Test criteria can also be used to determine when testing should stop; testing can stop when tests that satisfy all the defined criteria have been met.

Now, we introduce few test adequacy criteria that are relevant for testing object-oriented software. We also discuss a few coverage criteria which are used for generating the test cases in context to UML based testing.

2.7.1 Path Coverage Criterion

Path coverage criterion is too strong to be practically useful for most programs. A path is feasible if there exist some input data which causes the path to be traversed during a program execution, otherwise the path is infeasible [36]. In the rest of the thesis, we assume that the paths are feasible. There can be an infinite number of different paths in a program. In such a case, an infinite set of test cases must be executed for adequate testing. This is not practical. Hence, for practical testing applications, path coverage is usually defined in terms of linearly independent paths as follows:

The path coverage requires the design of test cases such that all linearly independent paths are executed at least once. A linearly independent path is any path in the control flow graph of the program that introduces at least one new edge or a node that is not included in any other linearly independent path. In the rest of the thesis, we use the term path coverage to mean the coverage of linearly independent paths. Path coverage criterion has been extended to message sequence paths criterion and transition path coverage criterion for application of object-oriented software.
2.7 Test Adequacy Criterion

2.7.2 Transition Path Coverage Criterion

Any sequence of transitions from the start state to a terminal state in a UML state chart diagram is considered to be a transition path. Transition path coverage requires that; given a test suite T and state chart diagram SD, T must cause each possible transition path in SD to be taken at least once [38].

2.7.3 Full Predicate Coverage Criterion

Full predicate coverage requires that each clause in the predicate is tested independently by a test suite. In other words, a test suite is said to achieve full predicate coverage if each clause in each predicate on every transition is made to independently affect the outcome of the predicate. Given a test suite TS and state chart graph G, TS must cause each clause in every predicate on each transition in G to take on the values TRUE and FALSE in turn. But all the other clauses in the predicate have values such that the value of the predicate will be the same as the clause being tested [88]. This ensures that each clause in a condition is separately tested. Given a test suit T and a behavioral diagram BD, full predicate coverage requires that, T must cause each clause in every condition in BD to assume the values TRUE and FALSE while all other clauses in the predicate have values such that the value of the predicate is the same as the clause being tested.

2.7.4 Message Sequence Path Criterion

Any sequence of messages from the start of an interaction to the end in a UML interaction diagram is called a message sequence path or a message path. A message sequence path represents a behavior to be tested during an interaction among objects to achieve some functionality. Given a test suite T and an interaction diagram ID, in order to satisfy the message sequence paths criterion, it is required that T must cause each possible message path in ID to be taken at least once [11,82].

2.7.5 All Basic Path Coverage Criterion

A basic path in an activity diagram, is a sequence of activities where an activity in that path occurs exactly once [54]. Note that a basic path considers a loop to be executed at most once. Given a set of basic paths $P_b$ obtained from an activity graph and a test suite T, for each basic path $p_i \in P_b$, there must be at least one test case $t \in T$ such that when the system is executed with test case $t$, $P_i$ is exercised. A basic path is a complete path through an activity diagram where each loop is
exercised either zero or one time. This ensures that all iterations in an activity diagram are exercised.

2.7.6 All Activity Path Coverage Criterion

An Activity Path is any sequence of activities from the initial activity into the terminal activity in the activity diagram (AD). Given a test suite \( T \) and an activity diagram \( AD \), \( T \) must cause each possible activity path in \( AD \) to be taken at least once. Given a set of activity paths \( P_A \) for an activity diagram and a test suite \( T \), for each activity path \( p_i \in P_A \) there must be a test case \( t \in T \) such that when the system is executed with a test case \( t \), \( p_i \) is exercised.

2.8 Fault Model

A fault model tells us where to look for faults for practical testing purposes. A fault model is a map of possible fault locations. A fault model is necessary to guide any rational testing strategy. It may be constructed from analysis of failure data. There are two types of fault models that have been defined based on the testing strategy adopted. The first type is the nonspecific fault model which is used in conformance directed testing. A conformance directed testing tries to establish conformance of the implementation to the requirements or specifications. Conformance testing relies on a nonspecific fault model, which targets any fault suffices to prevent conformance [16]. The second type is the specific fault model. A specific fault model is used in fault directed testing, which attempts to reveal certain implementation faults. Since the combination of input, state, output, and paths can be astronomically large, efficient probing of an implementation requires a specific fault model to direct the search for faults [16]. UML behavioral diagrams can be used to generate test cases for nonspecific as well as specific faults. Here we, defined some of the important fault models for UML behavioral models.

2.8.1 Scenario Fault Model

A sequence diagram depicts several operation scenarios. Each scenario corresponds to a different sequence of message paths in the sequence diagram. For a given operation scenario, sequence of messages may not follow the desired path due to incorrect condition evaluation, abnormal termination etc. [11].
2.8 Fault Model

2.8.2 Activity Fault Model

An activity diagram can reveal bugs related to operational aspects of a system [108]. Activity diagrams can be used to test whether the control flow in a system is incorrect or misplaced. Wrong actions, unwarranted termination of activities, improper initiation of activities, failure to consider all cases of conditional clauses, missing paths etc. are considered here. Incorrect sequencing of actions are also being considered. For example, certain action $A_1$ is required to be followed by another action $A_2$.

2.8.3 Interaction Fault Model

Test case design methods using interaction diagrams can reveal several faults related to object interactions such as when a client fails to send a message to a server or vice versa. A message may be lost or misdirected, a wrong object or method might be invoked, an erroneous sequence of messages may be generated, all alternate paths might not have been considered, an oversight might have occurred in considering different possibilities of a conditional clause etc. are some of the important faults related to object interaction.

UML interaction diagrams are more suited to generate test cases directed at detecting the cluster level faults arising out of the incorrect interactions among objects. Hence, test cases generated using interaction diagrams can reveal bugs related to integration of the classes of the system [108]. This is of importance since object-oriented programs are primarily event-driven in nature and there is no clearly defined integration structure. There is as such no decomposition tree to impose any integration ordering among objects. Hence, it is important to identify in what sequence, objects interact to achieve a common behavior. In this context, UML behavioral diagrams form a useful means by which we can test for faults at cluster level. Reported studies [8, 25] indicate that, test cases based on state based models have better capability of revealing class level faults than cluster level faults, and interaction diagram based test sets have better capability of revealing cluster level faults than class level faults. Further studies by Abdurazik et al. [8] have shown that UML diagrams can be used to generate test data systematically, and different UML diagrams can play complementary roles in revealing faults.
2.8.4 Synchronization Fault Model

This fault occurs when some message or activity begins execution before completion of execution of a group of all preceded messages or activities. In case of state chart diagram, this fault may arise if states or transitions are not synchronized within a scenario. If this fault is not detected, the system behaves incorrectly.

2.8.5 Decision Fault Model

This fault occurs in a decision/condition of a state or sequence or activity diagram. For example, a fault in the decision that decides validity of a registration may display registration information and payment information of some registrant for invalid registration ID, whereas for valid registration ID, it may display invalid account.

2.9 Program Slicing and its different categories

Slicing is a well known technique for program analysis. This technique can be used for extracting the relevant statements of a program P with respect to a slicing criterion. A slicing criterion for a program point is denoted as \((s, V)\) where \(s\) is the program statement and \(V\) is the subset of the program’s variable used or defined at \(s\).

A more general and widely accepted definition of a slice is the following:

*A slice is the subset of the statements of the programs which directly or indirectly affect the values computed at the slicing criterion.*

Weiser [117] was the first to introduce the concept of a program slice. His method constructs a slice \(S\) of a program \(P\) by removing statements which do not affect the slicing criterion such that the slices replicates part of the behavior of program \(P\). The program slices introduced by Weiser [117,118] are called static backward slices. Static slices are computed by considering all possible execution of a program with all possible input values. Backward slices contain the statements which have direct or indirect effect on the value of the inspected variable.

Figure 2.7 (a) shows an example program. The slice of the program with respect to slicing criterion \((15, p)\) is shown in Figure 2.7(b).

It is essential that a large program should be decomposed into a number of
2.9 Program Slicing and its different categories

Program slicing is essentially a decomposition technique that extracts only those program statements that are relevant to a particular computation [76]. In recent years, the following different forms of slicing have been reported.

- **Static Slicing**: Static slicing [3] technique performs static analysis of the source code to generate slices. Static slices are computed using static intermediate representation and remain valid for all possible input values to a program. Since predicate may evaluate either true or false for different input values, conservative assumptions have to be made, which may result into relatively
large slices. That is, a static slice may contain some statements (redundant) which might not be executed during an actual run of the program.

- Dynamic Slicing: Korel and Laski [4] introduced a new method of slicing known as dynamic program slicing. In the case of dynamic program slicing, only the dependencies that occur in a specific execution of the program are taken into account. A dynamic program slice with respect to a slicing criterion $\langle s, V \rangle$ for a particular execution, contains those statements that affects the slicing criterion in a particular execution. Dynamic slices are useful in interactive applications such as debugging and testing.

- Backward slicing: Backward slicing is perhaps the most intuitive form of program slicing. Here the slice is computed by working backwards from the point of interest finding all statements that have an effect on the slicing criterion.

- Forward slicing: A backward slice contains all parts of the program that might have directly or indirectly affected the slicing criterion where as a forward slice contain all those statements that might be affected by the variable V used or defined at the program point p. Forward slicing finds the statements that can be affected by changes to the specified variables at the point of interest [9]. A forward slice with respect to statement s shows how a value computed at s is being used in subsequent statements. This can help the programmers in ensuring that s establishes the invariants assumed by the late statements. Forward slicing can also be used to inspect the parts of the program that may be affected by a proposed modification, to check that there are no unforeseen effects on the program’s behavior.

### 2.10 Test Case Prioritization

The ordering or scheduling of the test cases based on certain coverage criteria is known as test case prioritization [48, 67]. The prioritization techniques schedule the execution of the test cases in an order that attempts to increase their effectiveness meeting some performance goal. The purpose of prioritization is to reduce the set of test cases based on some rational, non-arbitrary criteria, while among to select the most appropriate tests. The objective of prioritization is to develop test cases, order them and execute them to maximize an objective function like rate of fault detection, rate of coverage and reliability. The basic goals for prioritization of test cases are:
2.10 Test Case Prioritization

- To increase the rate of fault detection in the testing process.
- To achieve more test coverage earlier in the test process.
- To increase the likelihood of revealing regression errors related to specific code changes earlier in the regression testing process.
- To increase the confidence in the reliability of the SUT at a faster rate.

Rothermel et al. [30, 33, 97] had given a formal definition of test case prioritization as follows:

Given: T, a test suite; PT, the set of permutations of T; f, a function from PT to the real numbers.

Problem: Find $T' \in PT$ such that $(\forall T'') (T'' \in PT) (T'' \neq T') [f(T') \geq f(T'')]$.

Here, PT is the set of all possible orders of T, and f is an objective function that is applied to any such order and yields an award value for that ordering.

It may be noted that the function $f$ as mentioned in the problem statement of test case prioritization measures the effectiveness of a prioritization technique. For prioritization of test cases, a prioritization metric is considered based on certain coverage criteria.

Out of many possible goals of prioritization, we restrict our attention to the techniques that would improve the test suite’s efficiency, detect faults earlier, achieve faster fault detection rate and increase the confidence in the reliability without sacrificing the achieved coverage. To formally illustrate how rapidly a prioritized test suite detects faults, Rothermel et al. [33] introduced a metric called Average Percentage of Faults Detected (APFD) to measure the weighted average of the percentage of faults detected during the execution of the test suite.

Let $T$ be a test suite which contains $n$ test cases, and let $F$ be the set of $m$ faults revealed by $T$. Let $TF_i$ be the first test case in ordering $T'$ of $T$ which reveals fault $i$. According to [35], the APFD for test suite $T'$ is given by Equation 2.1.

\[
APFD = 1 - \frac{TF_1 + TF_2 + \ldots + TF_m}{n \times m} + \frac{1}{2 \times n}
\] (2.1)
In Equation 2.1, \( m \) is the number of faults contained in the program \( P \) and \( n \) is the total number of test cases. \( APFD \) value ranges from 0 to 100. An ordered test suite with higher \( APFD \) value has faster (better) fault detection rates than those with lower \( APFD \) values.

2.11 Summary

In this chapter, we have presented an overview of the background concepts. The idea was to provide some basic concepts and definitions which would be helpful to understand the work in the subsequent chapters. We have tried to keep the amount of information to a bare minimum by presenting only those information which are directly relevant to our research work. We have started by presenting the types of testing and the definitions of some basic testing terminologies. Next, the different approaches and levels of testing in object-oriented programs are also discussed. Next, we presented an overview of UML behavioral models that are relevant to our work. Then, we discussed the different test adequacy coverage criteria and fault models specifically found in behavioral UML diagrams. Finally, we have discussed some terminologies of program slicing and test case prioritization techniques.