**Chapter 4 Dynamic Testing and Test Case Design Techniques**

**Introduction:**

* The process of testing is similar to the process of development. We need to work out the test specification and design, carry out the detailed test case design, document the test cases and execute them.

**4.1 Dynamic Testing**

* In dynamic testing, the code is executed and checked for any defects. This testing has to be done very systematically and involves the following steps:

1. Review work products.
2. Identify test objectives.
3. Identify test specification and carry out test design.
4. Design the test cases.
5. Document test cases.
6. Execute the test cases.
7. Generate incident reports.
8. Log the defects.

**4.2 Review work Products**

* When a software product is given to you for testing, before taking up the actual testing you need to study the work products that have been written during the development phases. These work products include:

1. SRS document
2. Design document
3. Source code
4. Draft user manual (if available)

* From these documents we need to develop the test specifications. The entity based on which you develop a test specification is called “***test basis***”.
* Test basis can be a functional specification, a design strategy or a portion of the code found in the work products.

**4.3 Identify Test Objectives**

* After getting a good feel of the software, the next step is to study the details of the quality requirements of the product and identify what quality parameters need to be tested.
* Some quality parameters are:
* Usability
* Reliability
* Performance
* Code coverage
* Portability
* Security
* Conformance to a standard
* External interfaces
* If you have to test the software that is being developed in-house or through a sub contractor, you need to do the testing a number of times till the software is accepted by the user.
* In case, you are testing an off-the-shelf software package, your job may be just to test whether the software meets the requirements or not.
* In either case, you need to define the objectives, but testing off-the-shelf software is not complicated as testing the in-house developed software.

**4.4 Test Specification and Test Design**

* Test specification describes what to test. Test specifications are obtained based on the knowledge about the software, documentation and risks to be covered.
* While carrying out detailed review we come out with list of test specification based on the answers to the following questions:
* In addition to functional requirements, what non functional tests have to be done?
* Is it necessary to do structural testing?
* What types of test are required in addition to black box and white box testing?
* If conformance testing is required, what are the details of conformance requirements?
* What type of acceptance testing is required? Is it user, operational, conformance or any special contractual obligations need to be met?
* If operational testing is required, how long the field trial or beta testing has to be conducted?
* Recall your risk based testing, depending on the risks identified for the project, you need to prioritize your testing strategy.
* While working out test specification, you need to identify those tests that have to be done on a top priority.
* For ex, portability of software would have been identified as a quality parameter. So if there is a time constraint, portability can be assigned low.
* Once the specification are identified, the next step is to test design which involves:
* Based on prioritization of specification, identification of test environment.
* In case of embedded software, identification of hardware platform on which the software has to be tested.
* Identification of test instruments such as oscilloscopes, spectrum analyzer, frequency meter etc.
* Identification of necessary tools used for testing.
* Test script generation.
* Working out the details of the software stimulator/ driver software / stub software to be written.

**4.5 Design Test Cases**

* Test cases have to be designed for each test specification. For one test specification there may be one or more test cases.
* Test cases involve the following steps:
* Identify the test inputs.
* Identify the test data.
* Work out the test procedure.
* Write any scripts, if required.
* If there any special driver/ stub software required?
* Identify the entry criteria.
* Identify the exit criteria.
* Identify the expected result.
* Test cases have to be designed based on two criteria: reliability and validity.
* A set of test cases is considered to be reliable if it detects all errors. A set of test cases is considered as valid if at least one test case reveals the errors.
* So the following guidelines have to be considered while designing test cases:
* Test cases should be simple. For an input, there should be only one expected output.
* Test cases should reveal errors.
* There should be no redundancy.
* You always need to take valid input and also invalid inputs.
* Test cases should closely match with the actual usage of the software.
* In addition to functional requirements, for each quality parameter also you need to design test cases.
* For designing the test cases, the checklists are very useful.
* Test case design techniques are broadly classified into following categories:
* Black box test case design techniques.
* White box test case design techniques.
* Experience based test case design techniques.

**4.5.1 Black Box Test Case Design Techniques**

* In black box testing, we are only interested in the functionality of the software. So we do not bother about the implementation issues.
* Test case design is effectively based on how the end user uses the application to carry out his/her activities with the software. The test cases have to be done for both positive and negative testing.

**Specification Based Testing**

* In this technique, the test cases are designed based on the specifications. For each specification, the input values are chosen and the output is checked.
* Example: The specifications of a program to calculate the roots of quadratic equations are as follows: read the values of a,b,c.
* We need to choose some test cases such that roots become real and also some test cases such that roots become imaginary.
* Test case 1:

Test inputs: a=1, b=-5, c=6

Expected result: Roots are real, root1=3, root2=2

* Test case 2:

Test inputs: a=1, b=-4, c=4

Expected result: Roots are real, root1=2, root2=2

* Test case 3:

Test inputs: a=1, b=4, c=8

Expected result: Display of the message “Roots are imaginary”

**Equivalence Partitioning**

In this technique, all the possible valid input and valid outputs are identified. The test cases consist of both valid inputs and invalid inputs.

A group of tests form an equivalent class if:

All these tests test the same thing.

If one of the tests in the group catches a bug, then all the tests in the group will catch the bug.

It is not only equivalent input values, you can take equivalent output events or values, equivalent operating system.

Example: consider a program that takes three numbers as input and check whether the numbers form a triangle or not. The two equivalence classes are: the set of numbers that form a triangle and the set of numbers that do not form a triangle.

Test case1:

Input: 3,3,3

Expected result: The message “the numbers form a triangle”

Test case2:

Input: 3,4,5

Expected result: The message “the numbers form a triangle”

Test case3:

Input: 0,0,0

Expected result: The message “the numbers do not form a triangle”

Test case4:

Input: 78000, 32000, 81900000

Expected result: The message “the numbers are too big and cannot be handled”

Exit criteria: program should terminate because of overflow.

Executing test cases with multiple data is called data-driven testing. The test data can be stored in a database and for each record in the database, the test cases can be executed.

**Boundary Value Analysis**

* In boundary analysis, you need to identify the boundary values for the inputs at which the software is like to fail.
* The programmer is likely to make mistake at boundary values and hence there is a high probability of detecting errors at boundary values.
* Example : Suppose you need to test the form, for entering employee data in a database. You need to test the salary field. Suppose the salary field is declared as integer which takes the value from 1000 to 65535. In this case 65535 is a boundary value. We need to give valid and invalid boundary value and check whether the data is being accepted or an error message is being displayed.
* Similarly, we need to test the lower limit of the salary. Here the boundary is defined by 1000. So, we need to give 999, 1000 and 1001 and check whether the operation is correct or not. In case of 999 and 1000 the database should be updated, and in case of 1001 an error message should be displayed.

**Decision Tables**

* Decision tables are used when there are many conditional executions. The test cases need to be selected to ensure that all the conditions are tested at least once.
* Example: consider the income tax laws. The income tax to be paid depends on whether you are male or female, whether you are a senior citizen or not etc. imagine a country in which the tax laws are as per the following table

|  |  |  |  |
| --- | --- | --- | --- |
| **Salary** | **Individual(male)** | **Individual (female)** | **Senior citizen** |
| <=1,00,000 | 5% | 2% | Nil |
| <=2,00,000 | 10% | 5% | 5% |
| >2,00,000 | 20% | 15% | 10% |

* When you have to test the software, you need to ensure that at least one test case is chosen for every true/false condition in the table. You need to use the boundary value analysis also to design test cases.
* The boundary values for salaries are: 1, 00,000 and 2, 00,000. The boundary value for age is 65. Sometimes the specifications are not clear. Either way, boundary values are high yield test cases.

* Similarly, we need to test the lower limit of the salary. Here the boundary is defined by 1000. So, we need to give 999, 1000 and 1001 and check whether the operation is correct or not. In case of 999 and 1000 the database should be updated, and in case of 1001 an error message should be displayed.

**Cause Effect Analysis**

* In this approach, for each cause(input), effect(action) is identified and a cause effect graph is drawn. This graph, in turn, can be converted into a cause effect table which generates the test cases.
* Example: consider an example of withdrawal form verification in a bank account.
* Cause 1: account number is correct.
* Cause 2: signature of the withdrawal form matches with that in the bank database.
* Cause 3: enough money is available in the bank.
* Action 1: give money to the customer.
* Action 2: inform the customer that enough money is not there in the account.
* Action 3: call the police and inform that it is a fraud case.
* Each of the causes can be true or false. For the various combinations of these causes, the action or effect can be Action1 or Action 2 or Action 3. For each test case, it has to be checked whether the right action is being taken.
* So the various theoretically possible test cases are as follows:
* Test case1: correct account number, signature matches, enough money.
* Test case2: correct account number, signature matches, not enough money.
* Test case3: correct account number, signature does not match, enough money.
* Test case4: correct account number, signature does not match, not enough money.
* Test case5: Incorrect account number, signature matches, enough money.
* Test case6: Incorrect account number, signature matches, not enough money.
* Test case7: Incorrect account number, signature does not match, enough money.
* Test case8: Incorrect account number, signature does not match, not enough money.

**State Transition Diagrams**

* State transition diagrams (STDs) are extensively used to indicate the different states of a system. The STD help in choosing the various test cases- you need to ensure that each and every state is tested.
* Example: you need to test a PBX(Private Branch Exchange) call processing software. In PBX, you can dial another extension using say, three digit numbers. When all the three digits are dialed, the call is established.
* Initially, the telephone from which you are making a call is On-hook. When you pick up the phone, the state changes to Off-hook. Then you dial the first digit, then the second digit and then the third digit.
* When all the three digits are dialed the call is established and you can converse with the called party. But this is not a realistic situation- you may lift the receiver, then change your mind and keep it back.
* Imagine a telephone switch whose software hangs if you pick up a phone and then keep it back! Here is a partial list of test cases:
* Test case 1: Go off hook, and disconnect.
* Test case 2: Go off hook, dial 1 digit and disconnect.
* Test case 3: Go off hook, dial 2 digits and disconnect.
* Test case 4: Go off hook, dial 3 digits and disconnect.
* Test case 5: Go off hook, dial 3 digits, disconnect after conversation.
* Test case 6: Go off hook, and dial a wrong extension number.

**Use Case scenarios**

* Use case scenarios are the series of actions that a user is likely to take while using the software.
* Test cases can be designed in such a way that all these actions are performed one after the other and the desired result is obtained.
* Example: consider testing banking software. A use case is the customer gives a cheque to the bank employees. The bank employee verifies with the bank database about its validity by issuing a query to the database. The database will give a response indicating whether the cheque is valid or not; if valid cash is given to the customer.
* The series of action are tested in one shot in use case testing. In addition to this we need to generate test cases for validating the cheque by using equivalence partitioning, boundary analysis and cause effect analysis.

**4.5.2 White Box Test Case Design Techniques**

* In white box testing, the implantation details are taken into consideration. The implementation details are:
* Code at unit/component level.
* Unit/component interconnection at module level.
* Module interconnection at sub-system level
* Sub-system level interconnection at system level.
* White box testing is more difficult, but reveals more errors as the minute details of implementation are taken into consideration.

**CODE COVERAGE**

* As discussed earlier, at unit/component level, structural testing involves generating test cases based on the code.
* Statement coverage, condition coverage and path coverage are generally done at unit/component level by the programmers.
* If the code coverage is done at module/system level, then better quality of software can be ensured. However rather difficult to do it manually.
* Using tools, we can check code coverage easily. For example, profilers are available which can be used to check the code coverage. While doing dynamic testing, when you execute a particular test case, the profiler will indicate which lines code are executed and which lines of code are not executed. Now you can design another test case that will execute the earlier unexecuted lines.

**Internal boundary value testing**

* We can apply the boundary value technique to design test cases in white box testing values that are on the boundary of the equivalence classes are high yield test cases.
* Hence, such values need to be chosen as test inputs
* Example :

Consider testing of a for loop the loop statement,

for (i=0; i<=n; i++)

{

Statements;

}

* Is executed 101 times. Does the programmer really want 101 times or only 100times? Invariably programmers make mistake at the boundary values. So, testing has to be done at loop boundaries.
* Consider testing a simple if condition.

if (systolic\_bp>=120 && diastolic\_bp>=80)

printf (“hypertension”);

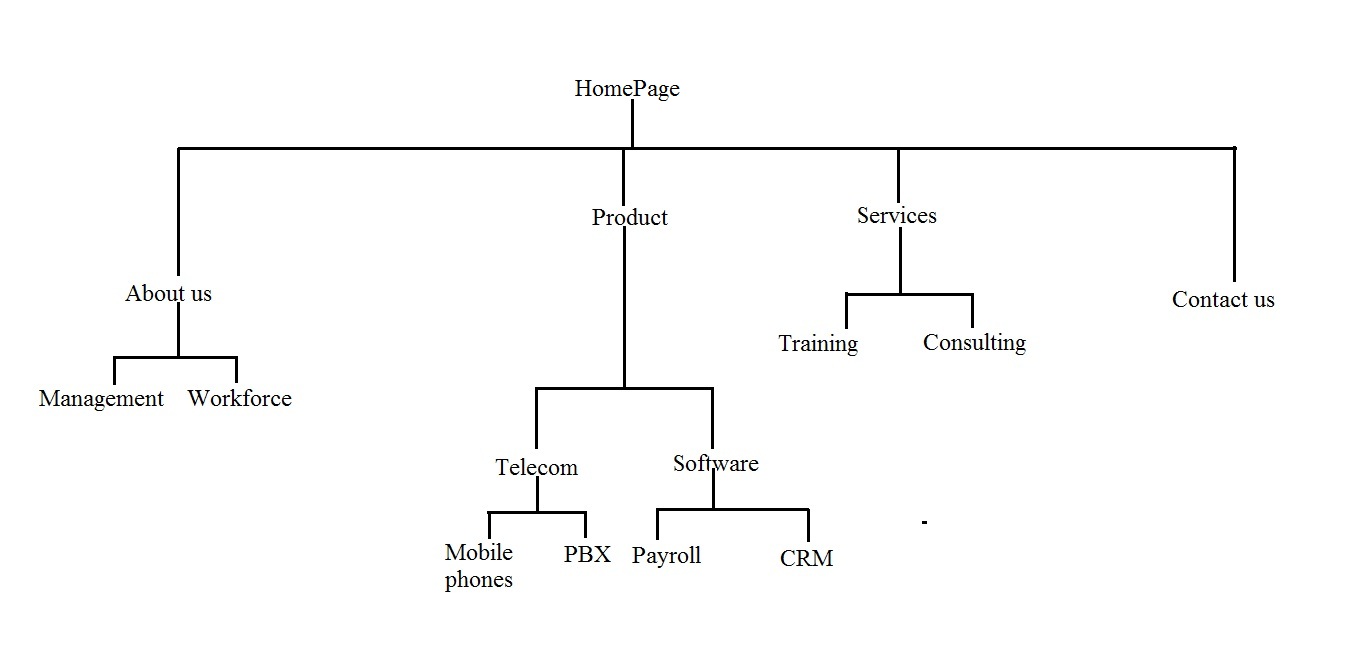
* In this condition, it is difficult to say whether the operator should be >= or >.
* Perhaps the programmer made a mistake in typing >=instead of >.if you are a test manger and if you happen to work with engineers fresh from college, check every condition.

**Structural testing at module level**

* When you integrate units into modules or classes into a functional module, then you can test the module taking into consideration the module structure
* Example: Consider object-oriented software for banking. As shown in figure below, there is an abstract class called ‘account’ from which accounts such as savings account, current account and fixed deposit are derived.
* You cannot form an object of the type ‘account’ because it is an abstract class, but you can form objects of the type savings bank account, current account and fixed deposit.
* To test this module, you can form accounts of various types and check the method/interfaces.

**Structural Testing at System Level**

* When you interconnect modules and develop the complete system, you will know how the modules are interconnected and hence using that information testing can be done.
* For example below figure shows the structure of a web site. There are four links on the home page: About us, Products, Services and Contact us.
* For each link, what are the next links are also indicated. You need to test whether the same structure is needed there in the web site. You also need to test all ensure that there are no missing links.



**4.5.3 Experience based test case design Techniques**

* Experience based test case design is more of an art than science. As you gain experience in testing, you become smarter in knowing the weaknesses of programmers and the requirements analysts. So you design test cases keeping in view these weaknesses generally you succeed.

**Error Guessing**

* Error guessing comes through experience as you gain expertise and experience in testing, you can easily guess what type of mistakes, developers make in implementing functionality and in writing code.
* Based on this ability to guess likely errors, you design test cases. Error guessing is simple test, based on your gut feeling. Highly experienced test engineers can detects many defects in rather short time by using this approach.

**Exploratory Testing**

* Generally while testing, we use the predefined test cases. This is called ‘***scripted testing***’. Testing carried out without using the predefined test cases is called exploratory testing.
* In exploratory testing the test cases are designed and executed in real time as against by using predefined test cases note that this different from ad-hoc testing, it is done rather unsystematically.
* Some process oriented people do not like exploratory testing to be done. But it is a very useful technique. When you’re testing the functionality of the software you may suddenly feel the need to try out a new test case.
* Invariably the new test case will detect a defect. If the exploratory testing yielded a defect, then the test case can be documented.
* Based on the principles of the test case design discussed above we will work out for the various test cases for an interactive voice response system and a finger print reorganization in the following sections.

**4.5.4 Case Study #1: Test Cases for an IVR System**

* To test an interactive voice response system, we will design the various test case keeping in view all the above principles of test case design and the various quality parameters that need to be tested an IVR system consists of (i) hardware (a PC add-On card)to which to telephone lines can be connected, and (ii) associated application software.
* Assume that two users can access the IVR system simultaneously through there fixed /mobile telephones. The user can dial an IVR number to obtain the arrival / departure time of a train, by keying in the train number. The various types of tests to be done along with the test case are listed in the below table:

|  |  |
| --- | --- |
| **Test types** | **Description** |
| Functional testing | Test the system for valid train numbers  Test the system for invalid train numbers |
| State transition testing | Dial the IVR number, get IVR response and disconnect  Dial the IVR number and disconnect  Dial the IVR number, key in train number and disconnect |
| structural testing | Draw the complete menu structure of IVR and test all possible paths |
| performance testing | Test the system when two users access simultaneously from their fixed telephones  Test the system when two users access simultaneously from their mobile phones  Test the system when two users access simultaneously, one from fixed telephone another from mobile phone. |
| Stress testing | Test the systems when three users access simultaneously, two users have should get IVR responses, third user should get an engaged tone |
| Gorilla testing | Dial the IVR number and key in digits randomly –does the systems which stand the random inputs or does it hang? |
| Experience based testing | When you use a IVR system repeatedly, you get used to the messages. so you may like to key in the train number without waiting for the long message to be completed. The IVR system should be capable of accepting in the digits even while a message is being played. And it should be capable of giving the arrival/departure information.  IVR systems do not work with old telephone instruments. Test the IVR system by dialing from an old instrument. |
| Reliability testing | Keep the IVR system on throughout the night and test the nest day by using functional testing test case  Switch off the IVR system (as though there is a power cut).Switch it, on and test the system by using functional testing cases. |
| Interface testing | The train arrival/departure information is stored in a database. Suppose it was earlier tested with MS access database. Replace it with Oracle database and repeat the about test case execution. |
| Portability Testing | Check the portability of the application software by using a portability testing tool.  Check whether the driver for the PC add-on cards for the other operating system is available(As the driver software is not portable) |
| Security testing | A user can dial an IVR system, listen to the welcome message and then he doesn’t key in the train number nor does he disconnect. You need to check whether the IVR software will automatically detect this type of misuse. If two users do such a thing internationally, the IVR system is not available for anybody else , this is called denial-of-service attack .The software has to be tested to check whether it is vulnerable to this type of attack |
| Conformance Testing | As the IVR system is connect to the telephone network, it should meet the requirements specified by the telecommunication authorities. However, only the hardware specifications will be tested-such as voltage levels, impedance etc |

**4.5.5 Case study #2: Test Cases for Fingerprint Recognition System**

* A finger print recognition system is used to allow the entry only to authorized persons at the entry point, the finger print taken is compared with the pre stored finger print database. Here are the various test cases for testing the system
* Test the system for valid cases(authorized person)
* Test the system for invalid cases(unauthorized persons)
* Test the performance by checking the time taken to allow an authorized user and reject an unauthorized user.
* Test accuracy rate by checking the time taken to calculate percentage of false recognitions
* Test the tolerance level of the system by applying ink to the finger of an employee.
* Check the ‘overriding’ feature: if an authorized person is disallowed by the system, what is the procedure to override that decision?
* Check the security of the computer in which the fingerprint data is stored by checking the password screen to access the computer/database.

**4.6 Document Test Cases**

* The test cases need to be documented as they will repeatedly used in regression testing. Each test case should contain the following description :
* Test case ID.
* Test case description.
* Principle used for test case design (equivalence partitioning, boundary value analysis, use case scenario etc).
* Revision history.
* Author name.
* Functionality to be tested.
* Test basis (references to the section in the work product based on which the test case id designed).
* Environment in which the test case has to be executed (hardware , operating system)
* Test setup.
* Pre-condition (status of the software before giving the test input)
* Post-condition (expected status of the software after executing the test input, including any error messages)
* Test procedures
* Expected results.
* Pass/fail criteria.
* Any changes to be made to the test case design document must be done only through the procedure defined in the configuration management plan.

**4.7 Execute Test Cases**

* To execute test case, the exact test procedure has to be worked out. In the case of GUI test, it may be just some mouse clicks and keyboard entries.
* In some cases, the exact procedure for interconnecting of modules, test data to be entered need to be indicated in the test cases design document.
* Based on the test procedure, each test case is executed. The actual results are compared with the expected results. Based on the pass/fail criteria, whether the test has passed or failed is decided.
* If you are using testing tools, then execution of the test cases is rather easy. You need to record a test case by actually executing the testing case manually. The tool will automatically generate a test script.
* This script can be executed for executing the test case. This is very useful in regression testing when the same test case is repeatedly executed.
* In addition to the pass/fail information, the tools also given important information as to when the test was run and how long it took the test case to be executed.
* This information can be of great use if you want to keep track of the total time of execution of all the test cases.

**4.8 Generate Incident Report/Anomaly Report**

* Any incident that needs to be probed into further is reported by using an incident report.
* When you execute a test case, if the expected the actual result do not match, then it is an incident that needs to be probed into and hence, an incident report is generated.
* Note that incident report may also need to be generated. While doing the desk checking of the various work product such as SRS document, design document, source code, test plan etc.
* The incident report should cover the following details:
* Date on which the incident or normally was discovered.
* Person/Organization who discovered the anomaly.
* Development phase in which the anomaly was observed(requirements review, design review, code review ,dynamic testing , maintenances).
* Description of the anomaly.
* Work product or test case that revealed the problem.
* Expected result.
* Actual result.
* Severity of the anomaly.
* Priority to be assigned to probe into the anomaly / fix the problem.
* Present status (to be assigned for fixing, under rectification, rectified, kept on hold duplicate anomaly as it was pointed out earlier).
* Change history (when the status was changed).
* Date of rectification and closure.

**4.9 Log the defects**

* For every project, a defect log has to be maintained, preferably by using defect tracking software tool or using Microsoft Excel.
* Note that defect log is different from the collection of incident reports-a defect may cause multiple incident reports. On the other hand, an incident may or may not be a defect.
* Each defect is entered into the defect log by the person who discovered the defect. The test manager will study the defect details, assign severity level to it and also assign priority for rectification.
* For instance, the priority will be high if testing of the entire product will come to a standstill or if a module testing cannot be done unless that defect is removed. Then the developer attends to the defect, and then the status of the defect is changed.

**4.10 Test Documentation Standards**

* Every test engineer has to document his/her work. A number document need to be generated such as:
* Test specification document/test design document.
* Test case design document.
* Incident report.
* Test log/defect log.
* Test closure/completion report.
* It is advisable to follow a standard format for documentation. The IEEE standards on software engineering will be very useful for this purpose. The standards that are of special signification for test engineers are:
* IEEE 829-1998 Standards for test documentation.
* IEEE 1028-1997 Standards for software reviews.
* IEEE 1044-1993 Standard classification for software anomalies.
* IEEE 1044.1-1998 IEEE Guide to classification for software anomalies.

**4.11 Formal methods of testing**

* How well wettest software depends on how well we design test cases. To deliver quality software, we need to follow the process of defined.
* Identify test objectives, work out the test specifications and the test design, design the test cases, execute the test cases and compare the expected results with the actual results.
* For all these years, the researchers have been trying to find out whether it is possible to do all these activities automatically.
* In other words, is it possible to prove that software works correctly by using tools? Proving that the software works correctly involves two things.
* To prove that the program has no errors
* To prove that the program is a true representation of user requirements.
* Any program that is written by using a programming language can be converted into a finite state machine- after all, the program changes from one state to another as it keeps executing the statements one after the other.
* Suppose we can develop a tool that converts the program into its equivalent state transition diagram and check its correctness. If all the states are correctly defined and moves from one state to another then we can prove that program is correct.
* If we use natural language to describe user requirements, there will be lot of vagueness. So we need to use formal methods to represent the user requirements.
* The formal methods such as Unified Modeling Language (UML), data flow diagrams etc. we are unable to achieve 100% success in representing all user requirements by using formal methods.
* Once if it is achieved we can integrate that tool with the tool that checks the correctness of the program and then the entire software testing process can be automated.