CHAPTER 3. ISSUES AND IMPORTANCE OF SOFTWARE TESTING WITH RESPECT TO RELIABILITY

3.1 Introduction

It is because of the human designer’s vulnerability for errors and its own abstract and complex nature, software development must be accompanied by quality assurance activities. It is not unusual for developers to spend 40% of the total project time on testing. For life-critical software (e.g. flight control, reactor monitoring), testing can cost 3 to 5 times as much as all other activities combined. The destructive nature of testing requires that the developer discards preconceived notions of the correctness of his/her developed software. Since the advent of high-level languages, the practice of developing software in a different environment compared to the environment in which it will eventually be used has become common. The development environment is referred to as the host, and the environment in which the software will be used is referred to as the target. Such a development strategy is referred to as host-target development, and the associated testing practices as host-target testing or cross testing.

Traditionally, host-target development has been used for embedded systems, where a powerful multi-user host environment is used to develop software, which is ultimately executed in an embedded microprocessor target environment. The Personal Computer (PC) explosion has opened new avenues for host-target development, with PCs being used as a development host for embedded systems, and also as a host to develop software, which will eventually be executed on mini or mainframe systems.
3.2 Software Testing

Testing is a process of executing a program with the intent of finding an error. A good test case is one that has a high probability of finding an as yet undiscovered error. A successful test is one that uncovers an as yet undiscovered error. Testing should systematically uncover different classes of errors in a minimum duration of time and with a minimum amount of effort. A secondary benefit of testing is that it demonstrates that the software appears to be working as stated in the specifications. The data collected through testing can also provide an indication of the software's reliability and quality. But, testing cannot show the absence of defect. It can only show that software defects are present [3.1].

3.2.1 White Box Testing

White box testing is a test case design method that uses the control structure of the procedural design to derive test cases. Test cases can be derived to

1. Guarantee that all independent paths in a module have been exercised at least once,
2. Exercise all logical decisions on their true and false sides,
3. Execute all loops at their boundaries and within their operational bounds, and
4. Exercise internal data structures to ensure their validity.

3.2.2 The Nature of Software Defects

The logic errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed. General processing tends to be well understood that while special case processing tends to be prone to errors. It is often believed that a logical path is not likely to be executed when it may be executed on a regular basis. Unconscious
assumptions of the developer about control flow and data lead to design errors that can only be detected by path testing.

### 3.2.3 Basis Path Testing

This method enables the designer to derive a logical complexity measure of a procedural design and use it as a guide for defining a basis set of execution paths. Test cases that exercise the basis set are guaranteed to execute every statement in the program at least once during testing.

### 3.2.4 Flow Graphs

Flow graphs can be used to represent control flow in a program and can help in the derivation of the basis set. Each flow graph node represents one or more procedural statements. The edges between nodes represent flow of control. An edge must terminate at a node, even if the node does not represent any useful procedural statements. A region in a flow graph is an area bounded by edges and nodes. Each node that contains a condition is called a predicate node. Cyclomatic complexity is a metric that provides a quantitative measure of the logical complexity of a program. It defines the number of independent executable paths in the basis set and thus provides an upper bound for the number of tests that must be performed [3.2].

### 3.2.5 The Basis Set

An independent executable path is any path through a program that introduces at least one new set of processing statements (move along at least one new edge in that path) [3.3].
The basis set is not unique. Any number of different basis sets can be derived for a given procedural design. Cyclomatic complexity, $V(G)$, for a flow graph $G$ is equal to

1. The number of regions in the flow graph.
2. $V(G) = E - N + 2$ where $E$ is the number of edges and $N$ is the number of nodes.
3. $V(G) = P + 1$ where $P$ is the number of predicate nodes.

### 3.3 Deriving Test Cases

Towards deriving a test case, the following procedure needs to be adopted.

1. From the design or source code, derive flow graph i.e-independent executable paths.
2. Determine the cyclomatic complexity of this flow graph.
   
   Even without a flow graph, $V(G)$ can be determined by counting the number of conditional statements in the code.
3. Determine a basis set of linearly independent paths.
   
   Predicate nodes are useful for determining the necessary paths.
4. Prepare test cases that will force execution of each path in the basis set.
   
   Each test case is executed and compared with the expected results.

### 3.4 Loop Testing

The white box technique focuses exclusively on the validity of loop constructs. The following are the four different classes of loops that can be defined:

1. Simple loops
2. Nested loops
3. Concatenated loops and
4. Unstructured loops.
3.5. Other White Box Techniques

Other white box testing techniques include:

1. Condition testing that exercises the logical conditions in a program.
2. Data flow testing that selects test paths according to the locations of definitions and uses of variables in the program.

3.6. Black Box Testing

Black box testing attempts to derive sets of inputs that will fully exercise all the functional requirements of a system. It is not an alternative to white box testing [3.4]. This type of testing attempts to find errors in the following categories:

1. Incorrect or missing functions
2. Interface errors
3. Errors in data structures or external database access
4. Performance errors and
5. Initialization and termination errors.

Tests are designed to answer the following questions:

1. How is the function's validity tested?
2. What classes of input will make good test cases?
3. Is the system particularly sensitive to certain input values?
4. How are the boundaries of a data class isolated?
5. What data rates and data volume can the system tolerate?
6. What effect will specific combinations of data have on system operation?
White box testing should be performed early in the testing process, while black box testing tends to be applied during later stages. Test cases should be derived which

1. Reduce the number of additional test cases that must be designed to achieve reasonable testing and

2. Prompts us something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

3.7 **Equivalence Partitioning**

This method divides the input domain of a program into classes of data from which test cases can be derived. Equivalence partitioning strives to define a test case that uncovers classes of errors and thereby reduces the number of test cases needed. It is based on an evaluation of equivalence classes for an input condition. An equivalence class represents a set of valid or invalid states for input conditions.

Equivalence classes may be defined according to the following guidelines:

1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.

2. If an input condition requires a specific value, then one valid and two invalid equivalence classes are defined.

3. If an input condition specifies a member of a set, then one valid and one invalid equivalence classes are defined.

4. If an input condition is Boolean, then one valid and one invalid equivalence classes are defined.
3.8 Boundary Value Analysis (BVA)

This method leads to a selection of test cases that exercise boundary values. It complements equivalence partitioning since it selects test cases at the edges of a class. Rather than focusing on input conditions solely, BVA derives test cases from the output domain also. BVA guidelines include:

1. For input ranges bounded by a and b, test cases should include values a and b and just above and just below a and b respectively.
2. If an input condition specifies a number of values, test cases should be developed to exercise the minimum and maximum numbers and values just above and below these limits.
3. Apply guidelines 1 and 2 to the output.
4. If internal data structures have prescribed boundaries, a test case should be designed to exercise the data structure at its boundary.

3.9 Mutation testing

The Mutation testing, Mutation analysis and Program mutation are used to design new software tests and evaluate the quality of software tests. Mutation testing involves modifying a program's source code or byte code in small ways [3.5]. Each mutated version is called a mutant and tests detect and reject mutants by causing the behavior of the original version to differ from the mutant. This is called killing the mutant. Test suites are measured by the percentage of mutants that they kill. New tests can be designed to kill additional mutants. Mutants are based on well-defined mutation operators that either mimic typical programming errors such as using the wrong operator or variable name or force the creation of valuable tests such as driving each expression to zero. The purpose is
to help the tester to develop effective tests or locate weaknesses in the test data used for the program or in sections of the code that are seldom or never accessed during execution.

Tests can be created to verify the correctness of the implementation of a given software system, but the creation of tests still poses the question whether the tests are correct and sufficiently cover the requirements that have originated the implementation. Mutation testing is done by selecting a set of mutation operators and then applying them to the source program one at a time for each applicable piece of the source code.

For example, consider the following C code fragment:

```c
if (a && b) { c = 1; } else { c = 0; }
```

The condition mutation operator would replace `&&` with `||` and produce the following mutant:

```c
if (a || b) { c = 1; } else { c = 0; }
```

Now, for the test to kill this mutant, the following three conditions should be met:

1. A test must reach the mutated statement.
2. Test input data should infect the program state by causing different program states for the mutant and the original program. For example, a test with `a = 1` and `b = 0` would do this.
3. The incorrect Program State, the value of ‘c’ must propagate to the program's output and be checked by the test.

However, it is not possible to find a test case that could kill some mutant. The resulting program is behaviorally equivalent to the original one and they are called equivalent mutants. Equivalent mutants detection is one of biggest obstacles for practical usage of
mutation testing. The effort needed to check if mutants are equivalent or not can be very high even for small programs [3.6].

Here are some examples of mutation operators for imperative languages:

Replace each boolean sub-expression with true and false.

Replace each arithmetic operation with another, e.g. + with *, - and /.

Replace each boolean relation with another, e.g. > with >=, == and <=.

Replace each variable with another variable declared in the same scope.

mutation score = number of mutants killed / total number of mutants

These mutation operators are also called traditional mutation operators.

3.10 Fault injection testing

In software testing, fault injection is a technique for improving the coverage of a test by introducing faults to test code paths, in particular error handling code paths, that might otherwise rarely be followed [3.7]. It is often used with stress testing and is widely considered to be an important part of developing robust software [3.8]. The propagation of a fault to an observable failure follows a well defined cycle. When executed, a fault may cause an error, which is an invalid state within a system boundary. An error may cause further errors within the system boundary. Therefore each new error acts as a fault, or it may propagate to the system boundary and be observable. When error states are observed at the system boundary they are termed failures. This mechanism is termed the fault-error-failure cycle and is a key mechanism in dependability.
Software Implemented fault injection techniques for software can be categorized into two types and they are compile time injection and runtime injection. Compile-time injection is an injection technique where source code is modified to inject simulated faults into a system. A simple example of this technique could be changing
\[ a = a + 1 \quad \text{to} \quad a = a - 1 \]

Code mutation produces faults, which are very similar to those unintentionally added by programmers. A refinement of code mutation is Code Insertion Fault Injection which adds code, rather than modifies existing code. This is usually done through the use of perturbation functions which are simple functions which take an existing value and perturb it via some logic into another value, for example

```c
int pFunc(int value) { return value + 20; }
int main(int argc, char * argv[]) {
  int a = pFunc(aFunction(atoi(argv[1])));
  if (a > 20) { /* do something */ }
  else { /* do something else */ }
}
```

In this case, pFunc is the perturbation function and it is applied to the return value of the function that has been called introducing a fault into the system.

Runtime Injection techniques uses a software trigger to inject a fault into a running software system. Faults can be injected via a number of physical methods and triggers can be implemented in a number of ways. Runtime injection techniques can use a number of different techniques to insert faults into a system via a trigger like corruption of memory space i.e., corrupting memory, processor registers, and I/O map. These techniques are often based around the debugging facilities provided by computer processor architectures.
3.11 Testing of Safety systems of Fast reactor

The Safety systems of fast reactors are tested and verified by systematic approach employing the techniques listed in sub-sections 3.1 to 3.8. The Table 3.1 shows the selective list of instrumentation and control (I&C) systems [3.9], for which verification was carried out using the above techniques.

The I&C systems are basically Triplicated or Dual redundant fault tolerant systems with switch over logic configured as distributed control systems and connected to the plant backbone network. This network is in-turn connected to the high-end servers to view any of the plant parameter at one single place in different format. The servers also log the data for future analysis. Each system runs the application software written in “C” programming language. The independent Verification / Testing team generates the test case for each sub-system. The validation of the I&C system is carried out using the dynamic simulator which can generate the scenario of the running nuclear power plant. The simulator is capable of generating the disturbance / Event as it can occur in the plant.

The errors generally encountered during testing of software developed for deploying in Nuclear power plant installations are listed in Table 3.2. A typical sample “Test Cases” for Instrumentation and control of Air Conditioning and Ventilation (AC&V) is represented in Table 3.3.
### Table 3.1 List of I & C Systems

<table>
<thead>
<tr>
<th>Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discordance Supervision</td>
</tr>
<tr>
<td>Reactor Startup Authorisation</td>
</tr>
<tr>
<td>Fuel Handling Startup Authorisation</td>
</tr>
<tr>
<td>Chilled water and Service Water system</td>
</tr>
<tr>
<td>Argon Supply and Distribution</td>
</tr>
<tr>
<td>Nitrogen Supply and Distribution</td>
</tr>
<tr>
<td>Supervision and control system for spent</td>
</tr>
<tr>
<td>Subassembly Storage Bay</td>
</tr>
<tr>
<td>Biological Shield Cooling System</td>
</tr>
<tr>
<td>Reactor Assembly Components</td>
</tr>
<tr>
<td>Control Building, Steam Generator Building &amp;</td>
</tr>
<tr>
<td>Reactor Containment Building Air Conditioning</td>
</tr>
<tr>
<td>and Ventilation system</td>
</tr>
<tr>
<td>Secondary Sodium Purification Circuit</td>
</tr>
<tr>
<td>Steam Generator Tube Leak Detection System</td>
</tr>
<tr>
<td>Safety Grade Decay Heat Removal system</td>
</tr>
<tr>
<td>Primary Sodium Main Circuit</td>
</tr>
<tr>
<td>Secondary Sodium Main Circuit</td>
</tr>
<tr>
<td>Top Shield Cooling System</td>
</tr>
<tr>
<td>Core Temperature Monitoring</td>
</tr>
<tr>
<td>Process Disturbance Analyser</td>
</tr>
<tr>
<td>Event Sequencer Recorder</td>
</tr>
<tr>
<td>Fuel Handling Systems</td>
</tr>
<tr>
<td>Radioactive Effluent System</td>
</tr>
<tr>
<td>System</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>Process Disturbance Analyzer</td>
</tr>
<tr>
<td>Generator Temperature Monitoring</td>
</tr>
<tr>
<td>Attendance System</td>
</tr>
<tr>
<td>Attendance System</td>
</tr>
<tr>
<td>Attendance System</td>
</tr>
<tr>
<td>SCADA</td>
</tr>
<tr>
<td>Guide tube profile</td>
</tr>
<tr>
<td>Control System</td>
</tr>
<tr>
<td>Category</td>
</tr>
<tr>
<td>------------------------</td>
</tr>
<tr>
<td>Pump Interlock</td>
</tr>
<tr>
<td>Permissive check for Startup</td>
</tr>
<tr>
<td>Rotating plug alignment system</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Sodium fill and Drain system</td>
</tr>
<tr>
<td>-------------------------------</td>
</tr>
<tr>
<td>Decay Heat Removal System</td>
</tr>
<tr>
<td>Alarm System</td>
</tr>
<tr>
<td>No.</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

Prepared by: 

Testing carried out by:
3.12 Inferences

This chapter covered the testing techniques deployed in the I&C systems of fast reactor that are listed in table 3.1. These systems are basically embedded systems with MC68020 as the processor, which uses the VME bus. The table 3.2 shows the errors found during the inspection of the software code written in “C” language. A typical sample of “Test Cases” is also shown in Table 3.3. Elaborate “Test Cases” were written for each of I&C systems. Planned systematic software testing and the error removal increase the reliability.
REFERENCES


[3.9] M.A. Sanjith, K. Kameswari, B. Ramasamy Pillai, S. Ilango Sambasivan & P. Swaminathan, Real Time Computer Based Control Systems for Prototype Fast Breeder Reactor, National Symposium on Applications of computer and Embedded Technology (SACET09), BARC, Mumbai