CHAPTER 4. EVALUATION OF SOFTWARE METRICS AND TOOL

4.1 Evaluation of Software Metrics

Identification of the software metrics that affect the reliability in terms of quality attribute and the development of tool to evaluate the metrics from the software code are very important. The various quality parameters that contribute to reliability [4.1] are described in the following subsections.

4.1.1 Cyclomatic Complexity (CC)

This attribute measures the complexity of each function in terms of number of independent executable paths, i.e., number of branches. Less the number of branches, better for the reliability. CC is a software metric developed by Thomas J. McCabe [4.2] and is the indicator of complexity. It is computed using the control flow graph of the program, where the nodes of the graph correspond to indivisible groups of commands, and a directed edge connects two nodes. It can also be applied to individual functions within a program. The number of test cases in the Basis Path Testing strategy is proportional to the cyclomatic complexity of the program.

4.1.2 Nesting Level

The depth of “condition branches” or the “loops” is the measure for each function. When the depth is less, the reliability is better, “test cases” are less and the testability and coverage are good.
4.1.3 Comment to Code ratio

Comments in the code is one way of documentation that goes with the code. More the comments is an indication of good understanding of the internal details of the function. Comments play a major role as part of maintenance in terms of fixing the problem or upgrading the software.

4.1.4 Ternary Operator

Programming languages use the feature, ternary operator, “?;,” which defines a conditional expression. It is sometimes referred to as "the ternary operator", though it is more accurately referred to as the “conditional operator”. The functional programming does not need such an operator as their regular conditional expression, e.g., if (a > b) {result = x;}
else { result = y; } can be rewritten as the following ternary statement: result = a > b ? x: y; This kind of convention is prone to error forcing difficulties during software maintenance.

4.1.5 Dynamic Memory

Dynamic memory allocation also known as heap-based memory allocation is the allocation of memory storage for use during the run-time. It can also be seen as a way of distributing ownership of limited memory resources among many pieces of data and code. Hence it is vulnerable for memory overflow and resulting in failure [4.3].

4.1.6 Goto and Continue Statements

The use of the ‘go to’ statement has an immediate consequence that it becomes hard to find a meaningful set of coordinates. The ‘go to’ statement as it stands is too primitive and could be a potential source of error. The “Continue” statement is used to add delay and it is
used to keep the label number to jump in. This leads to program un-structured and sensitive to location.

**4.1.7 Number of code lines in a function**

Less the number of lines of code, it will be easier to understand, test and maintain. On the other hand, when the code is lengthy and complex it is prone to error. So the number of code lines should be less and it should be simple in logic [4.4].

**4.1.8 Recursive functions**

This creates ambiguity and in-turn it is vulnerable for failures because of the unknown number of iterations which may result in stack overflow [4.5].

**4.1.9 Unused functions and Variables**

The unused functions and variables are called as “Dead Codes”. It will unnecessarily occupy memory and leading to ambiguity during maintenance and in impact analysis.

**4.2 Software Quality Metrics IEEE-1061 standard**

In developing a Set of Metrics, IEEE Standard 1061 [4.6] lays out a methodology for developing metrics for software quality attributes. The standard defines an attribute as “a measurable physical or abstract property of an entity”. A quality factor is a type of attribute, “a management-oriented attribute of software that contributes to its quality”. A metric is a measurement function, and a software quality metric is “a function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software adhere to a given attribute that affects its
quality”. To develop a set of metrics for a project, one creates a list of quality factors that are important for it. Associated with each quality factor is a direct metric that serves as a quantitative representation of a quality factor. For example, a direct metric for the factor reliability could be mean time to failure (MTTF). Identify one or more direct metrics and target values to associate with each factor, such as an execution time of 1 sec, which is set by project management. Otherwise, there is no way to determine whether the factor has been achieved [4.6].

For each quality factor, assign one or more direct metrics to represent the quality factor to serve as quantitative requirements for that quality factor. For example, if "high efficiency" was one of the quality requirements, assign a direct metric "actual resource utilization / allocated resource utilization" with a value of 90%. Use direct metrics to verify the achievement of the quality requirements [4.7]. Use only validated metrics (i.e., either direct metrics or metrics validated with respect to direct metrics) to assess current and future product and process quality. The Section 4.5 of IEEE Standard 1061 lays out several criteria for validation, which are summarized as follows:

1) Correlation: The metric should be linearly related to the quality factor as measured by the statistical correlation between the metric and the corresponding quality factor.

2) Consistency: Let F be the quality factor variable and Y be the output of the metrics function, M: F->Y. M must be a monotonic function. That is, if f1 > f2 > f3, then we must obtain y1 > y2 > y3.

3) Tracking: For metrics function, M: F->Y. As F changes from f1 to f2 in real time, M(f) should change promptly from y1 to y2.
4) Predictability: For metrics function, M: F->Y. If we know the value of Y at some point in time, we should be able to predict the value of F.

5) Discriminative power: A metric shall be able to discriminate between high-quality software components (e.g., high MTTF) and low-quality software components (e.g., low MTTF). The set of metric values associated with the former should be significantly higher (or lower) than those associated with the latter.

6) Reliability: A metric shall demonstrate the correlation, consistency, predictability, tracking and discriminative power properties of the application metric.

The validation criteria are expressed in terms of quantitative relationships between the attribute being measured (the quality factor) and the metric. The IEEE Standard 1061 recommends the use of direct metrics. A direct metric is a metric that does not depend upon a measure of any other attribute.

4.3 Thirty Software Engineering Measures

The set of thirty measures considered in the University of Maryland (UMD) study is listed below. The resulting measures constitute the basis of their study [4.8].

Bugs per line of code (Gaffney estimate), Cause & effect graphing
Code defect density Cohesion
Completeness Cumulative failure profile
Cyclomatic complexity Data flow complexity
Design defect density Error distribution
Failure rate Fault density
Fault-days number Feature point analysis
Function point analysis Functional test coverage
Graph-theoretic static arch. complexity  Human hours per major defect detected
Mean time to failure  Minimal unit test case determination
Modular test coverage  Mutation testing (error seeding)
Number of faults remaining (error seeding)  Requirements compliance
Requirements specification change requests  Requirements traceability
Reviews, inspections and walkthroughs  Software capability maturity model
System design complexity  Test coverage

4.3 Static Analyser Tool

This tool evaluates the software quality parameters and generates the report on the quality attribute metrics [4.9]. The report of Cyclomatic complexity with respect to each function of a typical safety system of NPP is shown in Figure 4.1. The right side of the screen shows the cyclomatic complexity of all the functions with the scroll bar.

Figure 4.1 Typical output screen of the SA with Cyclomatic Complexity
The Figure 4.1 shows the typical output screen with cyclomatic complexity detailing. All the parameters are listed on the left side of the screen. User can view any one parameter in detail by clicking the “Lens Icon”. The details of that parameter will appear on the right side of the screen. Here it is shown that “scan” and “pdsr” cyclomatic complexity is much higher than the prescribed limits. Each parameter “Help” is provided which explains the algorithm used in calculating the parameter and how to use the application software.

Figure 4.2 Screen output of the SA detailing Comment to Code ratio
The Figure 4.2 is detailing the Comment to Code ratio of the safety critical system software, which comprises of the complete project file. The comment percentage is only 44%, limit values shows that it should be minimum 50%. Table 4.1 presents the recommended limit values of Quality parameters on the code. This data is used to measure in terms of maintainability and the project development time [4.10]. The SA software is
developed using “Microsoft Visual Basic” as the programming language. The typical screen output of the developed tool of “Static Analyser” is shown in Figure 4.2.

**Table 4.1 Recommended Limit Values on code**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Limit Value</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comment to Code ratio</td>
<td>Greater Than or Equal to 50%</td>
<td>i) As per number of lines</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ii) Comments should be fairly distributed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>iii) Quality comments</td>
</tr>
<tr>
<td>Cyclomatic Complexity</td>
<td>Less than 20 for SC II</td>
<td>As per CASE tool</td>
</tr>
<tr>
<td></td>
<td>Less than 15 for SC I</td>
<td></td>
</tr>
<tr>
<td>Nesting Level</td>
<td>Less than 4</td>
<td>As per CASE tool</td>
</tr>
<tr>
<td>Ternary Operator</td>
<td>Not Allowed</td>
<td>As per MISRA guidelines</td>
</tr>
<tr>
<td>Dynamic Memory</td>
<td>Not allowed</td>
<td>As per MISRA guidelines</td>
</tr>
<tr>
<td>Goto, Continue</td>
<td>Not Allowed</td>
<td>As per MISRA guidelines</td>
</tr>
<tr>
<td>Number of code lines in a function</td>
<td>Less than 50 lines</td>
<td>Including comment lines</td>
</tr>
<tr>
<td>Recursive functions</td>
<td>Not allowed</td>
<td>Vulnerable to Stack Overflow</td>
</tr>
<tr>
<td>Unused functions</td>
<td>Not allowed</td>
<td>Dead code,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>As per MISRA guidelines</td>
</tr>
<tr>
<td>Variable number of arguments in a function</td>
<td>Not allowed</td>
<td></td>
</tr>
<tr>
<td>Unused Variables</td>
<td>Not allowed</td>
<td>As per MISRA guidelines</td>
</tr>
</tbody>
</table>
4.4 Inferences

The in-house developed Static Analyzer tool lists out the quality parameters with the measurements for each function of application software, which will be useful in calculating the software reliability. The recommended limit values on code based on the systems with the safety level criteria, which we have analyzed in almost 50 software modules of fast reactor, is also listed. Figure 4.3 shows the typical variation of failure intensity (Number of incorrect code segments) observed over the time (in hours) for three different embedded software, viz., a, b and c.

![Figure 4.3 Typical variation of failure intensity.](image-url)
REFERENCES


