Chapter One

Research Introduction

1.1 Introduction

Measurements are used in everyday life [1]. Measurements are used extensively in most areas of production and manufacturing to estimate costs, calibrate equipment, assess quality, and monitor inventories. Science and engineering disciplines depend on the rigor that measurements provide, but what does measurement really mean? [2]. According to Fenton, "measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules" [3]. In the context of software, measurement is important to improve Software quality which becomes a critical issue in current software evolution. The vast amount of software that has swept the markets in addition to the presence of a lot of software that manages delicate and dangerous tasks, making the quality of these systems important in specifying the continuation of business.

The 2009 Standish Group CHAOS Report [4] states that 24% of all software projects fail. This means they are cancelled prior to completion or delivered and never used. One of the contributing factors is that modern software is almost never completely developed from scratch, but is rather extended and modified using existing code and often includes third party source code. This can lead to poor overall maintainability, difficult extensibility and high complexity. To better understand the impact of code changes and track complexity issues as well as code quality software metrics are frequently used in the software development life cycle. “Software metrics provide measurement of the software product attributes and the process of software production” [4].

With an increasing complexity in information and software systems design as well as the emergence of new software design and development paradigms, the focus of software measurement widened to include measurement during the earlier stages of the software development lifecycle, not only at code level. Design level metrics can in theory be obtained much earlier in the development of a project thus providing information which can be used for many purposes [5].

1.2 Problem Statement and its significance

The software engineering field doesn't have a unified set of metrics that the community has agreed to use; instead there is a set of metrics that advised to use them [6]. Most large companies dedicated to create their own standards of software measurement; so the way metrics are applied usually varies from one company to another.

Current used metrics were defined and calculated using only syntactic aspects of software – using only aspects related to syntax and format of the code – such as LOC, complexity etc. All syntactically based metrics have the problem of mapping between values calculated by metrics and some quality attributes such as reliability, cohesion… etc. are arguable.

A significant drawback of syntactic metrics is that different structural aspects of code can result in different metrics values, even when the code is performing the same task. Syntactic metrics are not always accurate descriptors of quality. Metrics that provide a better mapping between software and its associated quality factors have the potential to be used in improving software quality, including quality of newly developed software as well as currently maintained software. Such metrics can help in identification of good reusable software components.

On the other hand, desirable quality attributes like reliability and maintainability cannot be measured until some operational version of the code is available [3]. In addition to that, there is a need to integrate some kind of measure to the semantic features of software which affects quality attributes. Yet we wish to be able to predict which part of the software is less reliable, more difficult to test, or require more maintenance than others, even before the system is completed.

1.3 Research Question/Hypothesis/Philosophy

1.3.1 Research Question

Two broad questions will be addressed:

1- How to use semantic metrics to improve measurement of software reliability?

2- How semantic metrics can be computed from a static analysis of the source code?

The answer for this question leads to another **sub questions** such as the following:

1- What is the measurement?

2- How to measure reliability in software.

3- What is most widely used measurement metrics that can be used?

4- What are the problems of traditional used metrics?

5- What is the main factors affecting software reliability? And how to measure it?

6- What kind of semantic metrics required achieving such software quality attribute “reliability”?

7- How to Linking semantic metrics to reliability or other quality attributes?

8- How to compute semantic metrics by static analysis?

1.3.2 Research Hypothesis

Software metrics can be used to examine the quality of software. It gives developers or designers a picture about the expected efficiency of running code. The use of semantic metrics has made a big contribution to the field of software quality measurement. The following is the research Hypothesis:

- Semantic metrics can be used to measure software reliability.
- Semantic metrics can be used to predict fault tolerance without affecting by code structure or programming language.
- Using metrics in different stages of software development may improve software quality.
- Using of semantic measurement is much better than traditional used metrics (synthetic).

1.4 Research Philosophy

The philosophy of suggested metrics is based on the concept that different programming languages and structures can result in different measurement values for some quality attributes.

With an increasing complexity and quality requirements of information and software systems designs as well as the emergence of new software design and development paradigms, the role of software measurement has increased in recent years [7]. Measurement techniques widened to include measurement during the earlier stages of the software development lifecycle, not only at code level. Using of software metrics becomes an important issue to be discussed.

1.5 Research objective

The main research objective is to:

Define and construct semantic metrics that are contributing to the area of software reliability measurements and monitor/control product reliability. This can be achieved by the following **Specific objectives:**

1- Define set of metrics that examine the software semantically.
2- Correlate metrics to reliability attributes.
3- Using semantic metrics to estimate relevant quality attributes such as reliability, fault tolerance, and the like.
4- Validate the proposed metrics using empirical observations.
5- Use the metrics to build an analytical model of software reliability

1.6 Open research areas

The software metrics field is an ongoing research area. Although, there is a number of software metrics that are widely used to test some software attributes, still the area is young and requires much research. There are still open research issues that need to be investigated such as:

1- Traditional metrics (syntactic metrics) doesn’t represent an accurate measure because it calculated only by using syntactic aspects of software. Therefore, different programming language or even different program structure of same function may result in different measurement values.
2- Unified set of metric: there is a need for standardized set of metrics in which there is ability to assess software attributes in different phases by integrating both semantic and syntactic metrics. Research in this direction is very little and still no optimum results have been reached.
3- Semantic metrics: using these kinds of metrics can help to overcome the limitation of syntactic metrics. But still not much work has been done in this area and no optimum solution has been found yet. two research directions:
    - Linking semantic metrics to meaningful software quality attributes.
    - Computing semantic metrics from source code and system specification.

Semantic metrics is a new trend in software measurement. Most studies in this area agreed upon evaluating software in early stages in its development life cycle are better for quality assurance. Since many studies have started suggesting metrics to work in the design phase or even after implementation and a few of them attempt to extract knowledge from system requirements. It is worth mentioning that not much research has been done in semantic metrics, instead most of the focus is on semantic web and ontology.

1.7 Scope:

The study represents an attempt to contribute to the field of software metrics. It considers semantic metrics that help to improve the monitoring of software quality by measuring some quality features. Four semantic metrics were suggested. During the research process metrics will be defined and evaluated.

In the study, only semantic metrics will be considered, In addition to their allied features, advantages and drawbacks of applying them on software products and using these metrics to give an indication of reliability. Semantic metrics have a broader scope, because they abstract away syntactic details to focus on program states and program functions, and can be applied uniformly across heterogeneous software systems.

1.8 Proposed Solution

The study is concentrated on defining semantic metrics which reflects what functions the software product defines, rather than how these functions are represented. Our proposed solution is an attempt to contribute to this area. In particular, the study considered the following metrics [8], which are defined using information theory functions:

- State redundancy: This metric reflects the extent to which a state is redundant, i.e. includes relationships between its various variables; programs which carry much state redundancy are more likely to be able to detect erroneous states, when these arise.
- Functional redundancy. This metric reflects the extent to which the function of the program is redundant, i.e. its results is represented in variables that have many relationships between them; programs whose functions are redundant are more likely to be able to detect errors in the results of their function execution, when these arise.
- Maskability. This metric reflects the extent to which the function of a program maps different inputs into common outputs; programs that have high maskability are more likely to map erroneous states into correct final states, thereby avoiding failure and making error recovery unnecessary.
- Non determinacy. Whereas the previous metrics dealt with the program (more specifically its semantics), this metric deals with the specification of the program, and represents the property that the same input may be mapped to a wide range of possible correct output; specifications that are nondeterministic are more likely to tolerate programs that produce erroneous final states.

Together, these four metrics reflect the ability of a program to be correct with respect to its specification; unlike syntactic software metrics, they depend on what function the program computes and what specification the program is intended to satisfy, rather what form the program takes.

1.9 Research Methodology

To begin with, a review to the current state of art is required to capture knowledge about kinds, classification and uses of different software metrics. Based on findings from the literature review a new set of metrics are suggested to overcome the limitations of existing tools. Suggested metrics are modeled and evaluated against the research objectives. The evaluation process is iterative each time research objectives are re-examined to ensure that work is going in the right direction.

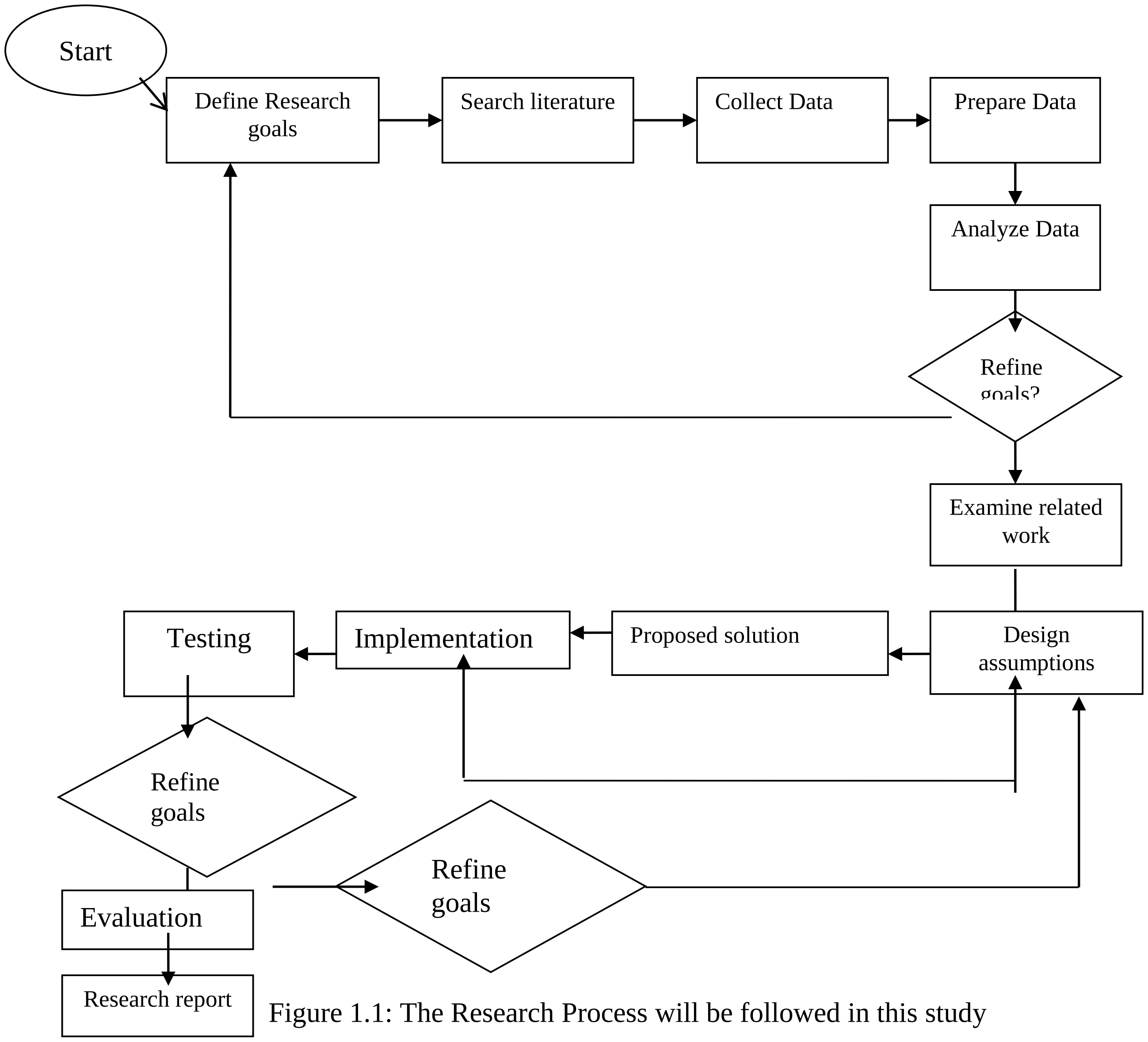


Figure 1.1: The Research Process will be followed in this study

1.10 Thesis organization:

The thesis chapters are organized as follow:

Chapter one: presents a general introduction including the problem statement, research objectives, question, hypothesis and the proposed solution. Chapter two: sheds light on the major developments in the field of software metrics. Chapter three, explore what has previously been done in the area, chapter four, introduces the following methodology. Chapter five and six: cover used reliability mechanisms in general, in addition to the proposed semantic metrics. Chapter seven presents the validation process, results and discussion. Finally, the research conclusion and future work are covered in chapter eight.

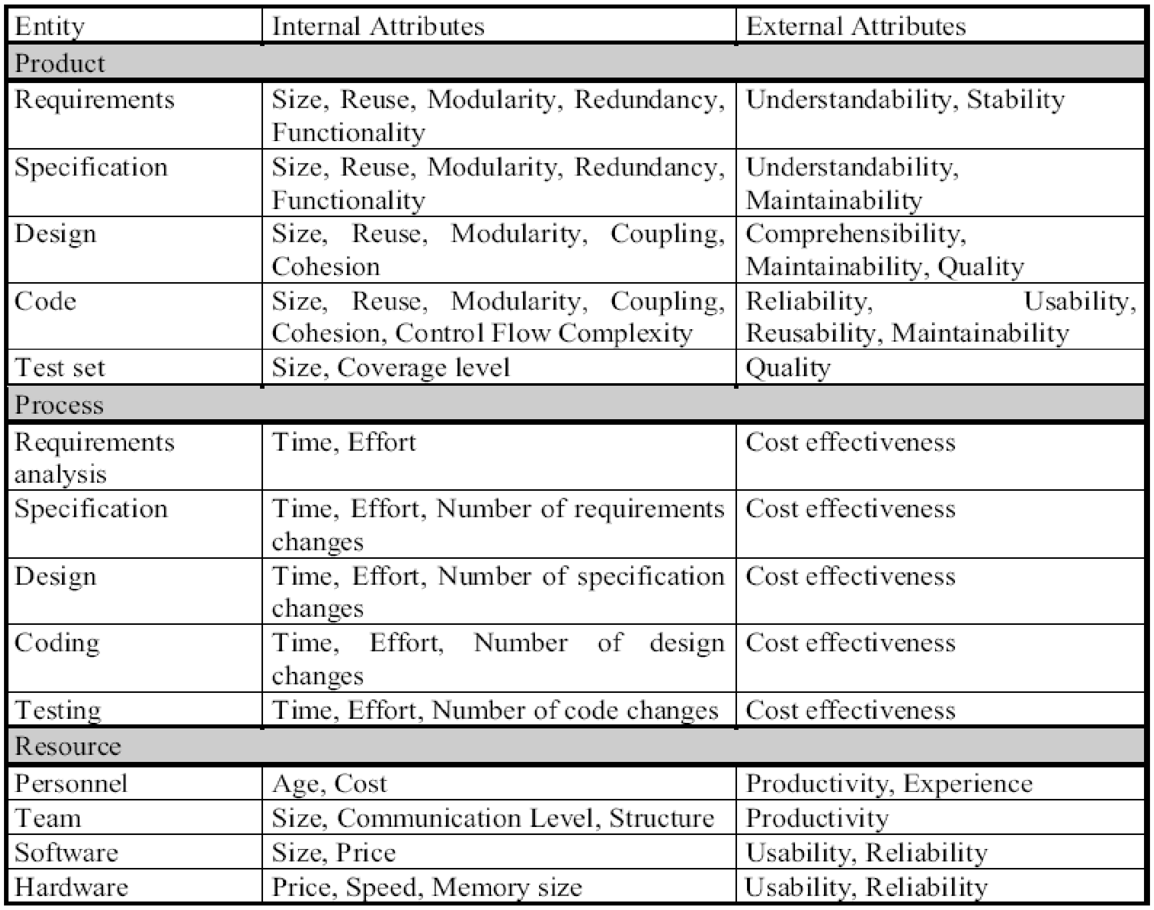
Chapter Two

Background

2.1 Introduction

Measurement is the process of assigning numbers or symbols to attributes of entities in the real world in such a way as to describe them according to clearly defined rules. [9] In general Measurement has two broad uses: for assessment e.g. Monitoring project progress to facilitate corrective decisions if required, and prediction such as planning certain project resources [9]. When using measurement for prediction, the value of an attribute is given by a mathematical model that relates the attribute to the measurement of other attributes. Table 2.1 shows the main software entities along with their internal and external attributes.

Table 2.1: Software Entities with their attributes [9]



Entity	Internal Attributes	External Attributes
Product		
Requirements	Size, Reuse, Modularity, Redundancy, Functionality	Understandability, Stability
Specification	Size, Reuse, Modularity, Redundancy, Functionality	Understandability, Maintainability
Design	Size, Reuse, Modularity, Coupling, Cohesion	Comprehensibility, Maintainability, Quality
Code	Size, Reuse, Modularity, Coupling, Cohesion, Control Flow Complexity	Reliability, Usability, Reusability, Maintainability
Test set	Size, Coverage level	Quality
Process		
Requirements analysis	Time, Effort	Cost effectiveness
Specification	Time, Effort, Number of requirements changes	Cost effectiveness
Design	Time, Effort, Number of specification changes	Cost effectiveness
Coding	Time, Effort, Number of design changes	Cost effectiveness
Testing	Time, Effort, Number of code changes	Cost effectiveness
Resource		
Personnel	Age, Cost	Productivity, Experience
Team	Size, Communication Level, Structure	Productivity
Software	Size, Price	Usability, Reliability
Hardware	Price, Speed, Memory size	Usability, Reliability

There are different ways to express the data collected in software measurement. As described in [6] statisticians recognize four different types of measured data or measurement scales with their associated possible operations. As the collection of data and their usage for estimates truly is a statistical method. If inappropriate operations are used for analysis, the results will be useless. The following table 2.2 gives an overview of the measurement scales and possible operations: [10]

Table 2.2: Measurements Scales [10]

Type of Data	Description of Data	Possible Operations	Explanation
Nominal	Classification	equal, not equal	named categories with no attached value
Ordinal	Ranking	greater/better, less/worse, median	named categories with ordered values
Interval	Differences	addition/subtraction, mean, variance	numbers without an absolute zero
Ratio	Absolute Zero	Relation	Numbers with an absolute zero

2.2 Software Measurement and Quality

2.2.1 Software quality Measurement concept

A principal objective of software engineering is to improve the quality of software products [10]. Quality must be defined in terms of specific attributes interested to user. Such attributes are classified into internal and external ones [9]. Internal attributes can be used as predictor to other attributes. The notion of quality is usually captured in a model that describes composite set of attributes along with its relationship. Many models show distinction between internal and external attributes. The following models gain acceptance within software engineering communities.

2.2.2 Software Quality Models

1- Early Models:

McCall and Boehm described quality using decomposition approach. McCall model was developed for US Air force, and used with in the US department of defense for evaluating software quality [3]. It includes 41 metrics to measure 23 quality criteria from factors. In such model to measure any criteria a list of check list have to be answered accordingly from requirement, design and implementation. Though Boehm's and McCall's models might appear very similar, the difference is that

McCall's model primarily focuses on the precise measurement of the high-level characteristics "As-is utility", whereas Boehm's quality model is based on a wider range of characteristics with an extended and detailed focus on primarily maintainability [3]. The following is the main features of such models:

a. Boehm and McCall model: model builders focus on formal products and identify key attributes of quality. From the user prospective there are three key attributes called "quality factors" such as reliability, usability and maintainability which are high level external attributes these factors are related to many internal attributes called quality criteria.[3]
b. In McCall model the factor reliability is composed of consistency, accuracy, correctness, fault tolerance and simplicity. Sometimes quality criteria are internal attributes such as "structures" and modularity, reflecting developer's belief. The internal attributes have the effect on external quality attributes. Further decomposition is required in which quality criteria are associated to low level directly measurable attributes (quality metrics). Figure 2.1and Fig 2.2 shows quality attributes and their decomposition [3] respectively.

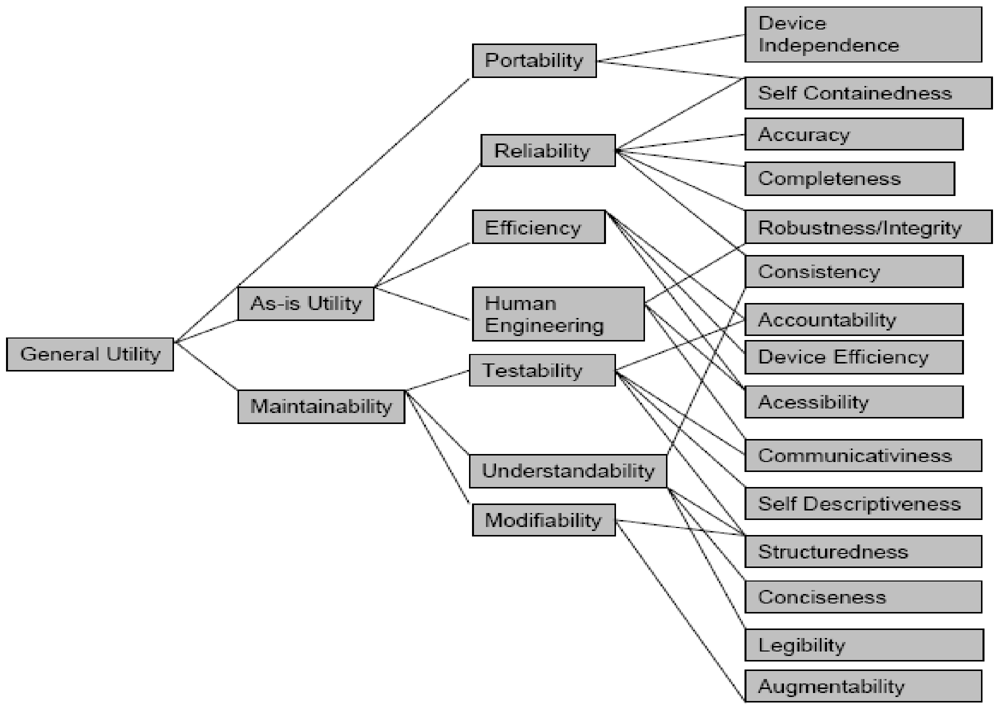


Figure 2.1: Boehm 'software Quality characteristics Tree [11]

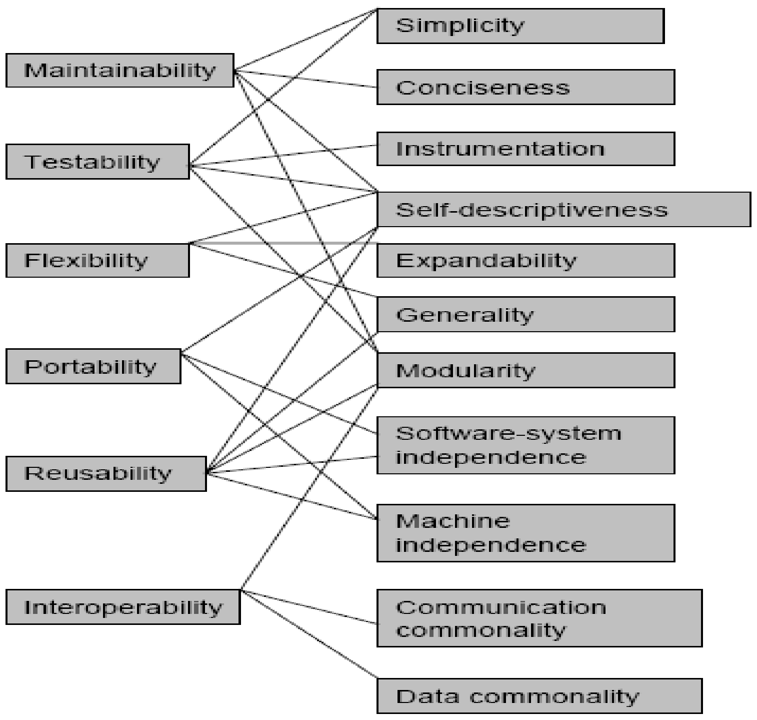


Figure 2.2: McCall's Quality Model [12]

Software quality can be seen in two different ways one is by using fixed models approach, another is to define their own models by adopting the current models to meet specific quality requirement.

2. Dromey's Quality Model

An even more recent model similar to the McCall's, Boehm's and the FURPS quality model, is the quality model presented by R. Geoff Dromey [12,13]. Dromey proposes a product based quality model which recognizes that quality evaluation differs for each product and that a more dynamic idea for modeling the process is needed to be wide enough to apply for different systems. The main focus of the Dromey is on the relationship between the quality attributes and the sub-attributes, as well as attempting to connect software product properties with software quality attributes.

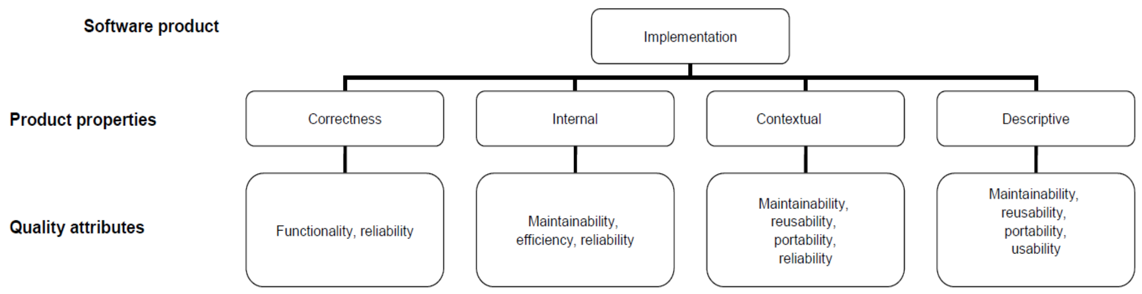

Software product
Implementation
Product properties
Correctness
Internal
Contextual
Descriptive
Quality attributes
Functionality, reliability
Maintainability, efficiency, reliability
Maintainability, reusability, portability, reliability
Maintainability, reusability, portability, usability

Figure 2.3: principles of Dromey's quality model [14]

3. Define own model:-

This approach was pioneered by Gillb et.al [12]. Their thought was to design a measurable objective. User identifies the key measurable attributes in specification, then software engineer design the product according to these attributes. The product will be re-checked by the user to make sure that the objectives have been met. Kitchinham [14] extended Gilb's idea and support it with automated tools. In 1992, derivation of McCall model was proposed as a basis for international standard software quality ISO9126 (ISO 1991). In this standard software quality is defined to be "the totality of features and characteristics of software product that pear on its ability to satisfy stated needs"[14]. The model decomposes the quality into six factors as follow: functionality, efficiency, usability, maintainability and portability. Each of these factors is defined as a set of attributes e.g. reliability in ISO 9126 is defined as "asset of attributes that bear on capability of software to maintain its levels of performance understand condition for a stated period of time. The standard is an important milestone in development of software quality measures. Nemours companies used ISO model to support quality evaluation [12]. Although, objective measurements is much better than subjective one, the measurement of many quality factors described in formal models including McCall and Boehm models is dependent on subjective ratings. Figure (2.4) shows the ISO 9126 model.

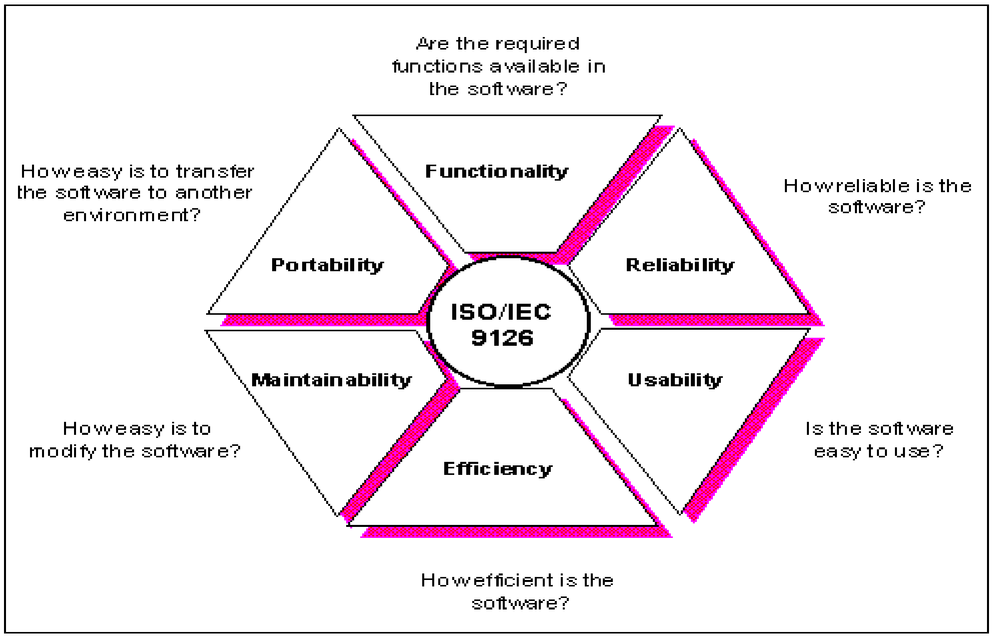
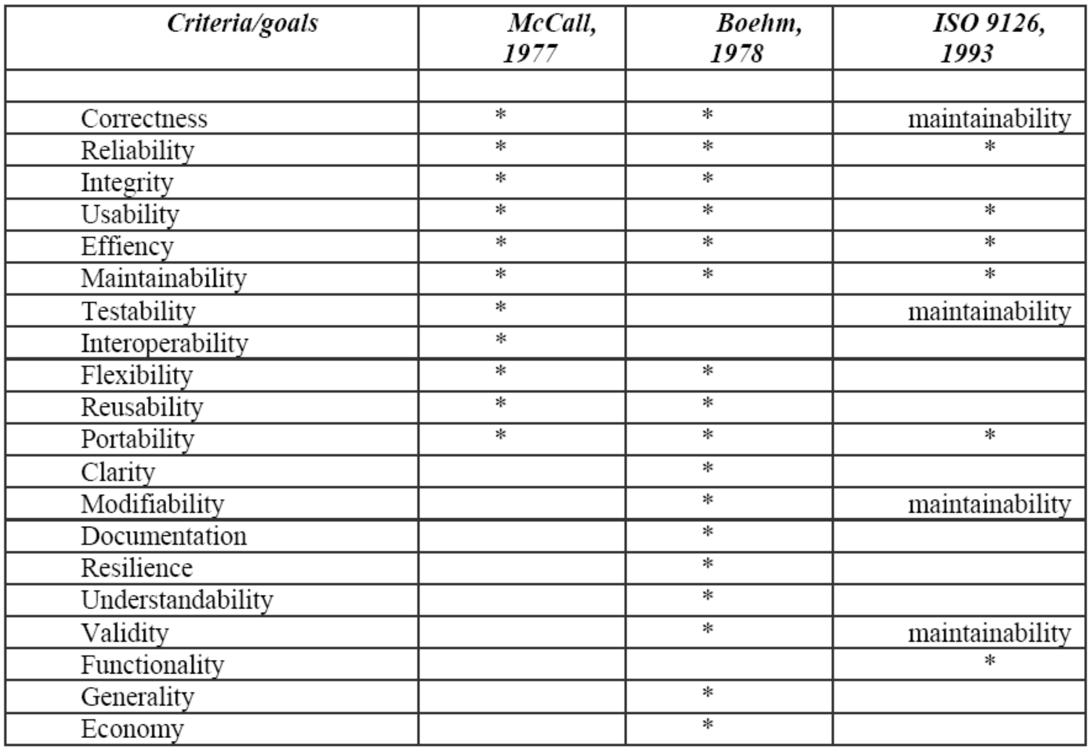

Are the required functions available in the software?
Functionality
How easy is to transfer the software to another environment?
Portability
How reliable is the software?
Reliability
ISO/IEC 9126
Maintainability
Usability
How easy is to modify the software?
Is the software easy to use?
Efficiency
How efficient is the software?

Figure 2.4: The ISO 9126 quality model [14]

The following table (table 2-3) compares between mentioned quality models based on quality attributes.

Table 2-3: Comparison between criteria/ goals of the McCall, Boehm and ISO 9126 quality models [13]



Criteria/goals	*McCall, 1977*	*Boehm, 1978*	*ISO 9126, 1993*
Correctness	*	*	maintainability
Reliability	*	*	*
Integrity	*	*	
Usability	*	*	*
Effiency	*	*	*
Maintainability	*	*	*
Testability	*		maintainability
Interoperability	*		
Flexibility	*	*	
Reusability	*	*	
Portability	*	*	*
Clarity		*	
Modifiability		*	maintainability
Documentation		*	
Resilience		*	
Understandability		*	
Validity		*	maintainability
Functionality			*
Generality		*	
Economy		*	

ISO 9126 proposes a standard which specifies six areas of importance, i.e. quality factors, for software evaluation. Each quality factors and its corresponding sub-factors are defined. The following is the two factors (functionality and reliability):

- **Functionality**: A set of attributes that relate to the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.
  - Suitability: Attribute of software that relates to the presence and appropriateness of a set of functions for specified tasks.
  - Accuracy: Attributes of software that bare on the provision of right or agreed results or effects.
  - Security: Attributes of software that relate to its ability to prevent unauthorized access, whether accidental or deliberate, to programs and data.
  - Interoperability: Attributes of software that relate to its ability to interact with specified systems.

- Compliance: Attributes of software that make the software adhere to application related standards or conventions or regulations in laws and similar prescriptions [14].

➢ **Reliability**: A set of attributes that relate to the capability of software to maintain its level of performance under stated conditions for a stated period of time.

- Maturity: Attributes of software that relate to the frequency of failure by faults in the software.
- Fault tolerance: Attributes of software that relate to its ability to maintain a specified level of performance in cases of software faults or of infringement of its specified interface.
- Recoverability: Attributes of software that relate to the capability to re-establish its level of performance and recover the data directly affected in case of a failure and on the time and effort needed for it [14] .
- Compliance: See above.

Figure 2.5 Shows ISO 9126 software product evaluation: quality characteristics and guidelines for their use [13].

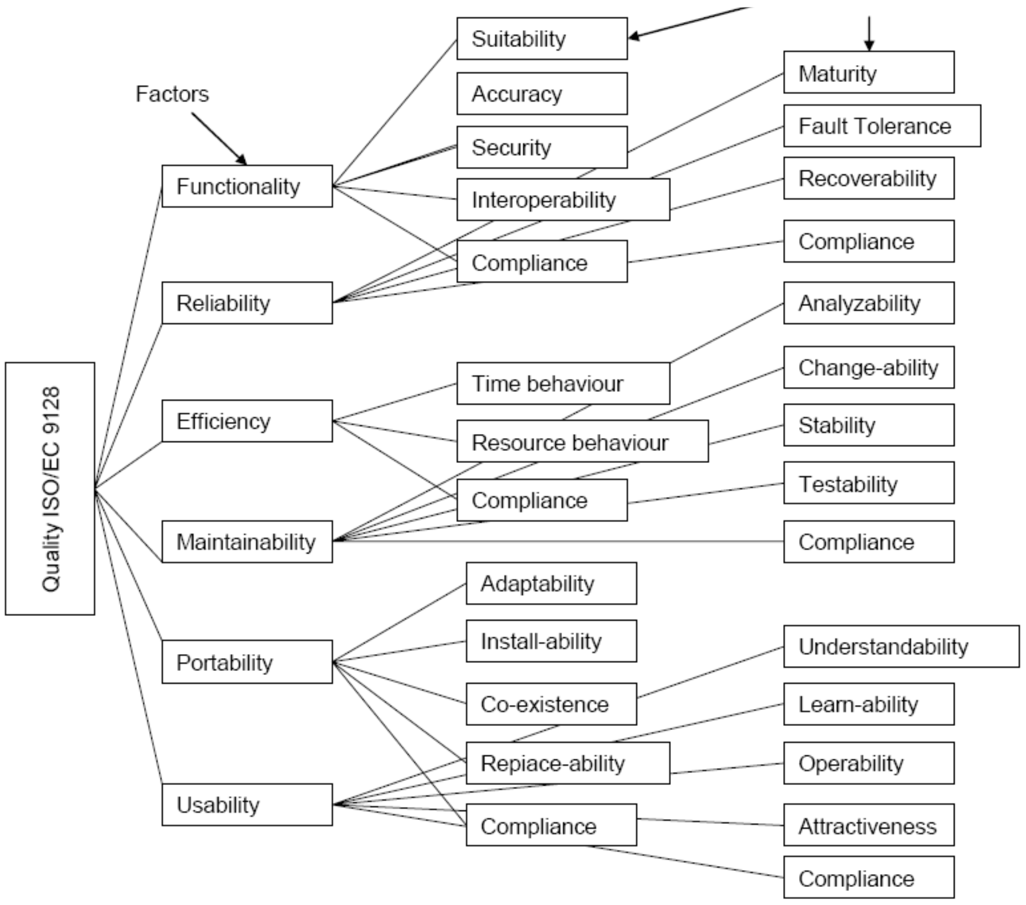


Figure 2.5: ISO 9126 software product evaluation: quality characteristics and guidelines for their use.[13]

4. Capability Maturity Model(s) (CMMs)

Is a development model created after study of data collected from organizations that contracted with the U.S. Department of Defense. The term "maturity" relates to the degree of formality and optimization of processes, from *ad hoc* practices, to formally defined steps, to managed result metrics, to active optimization of the processes. The CMM/SW-CMM depicted in Figure 2.6 below addresses the issue of software quality from a process perspective.

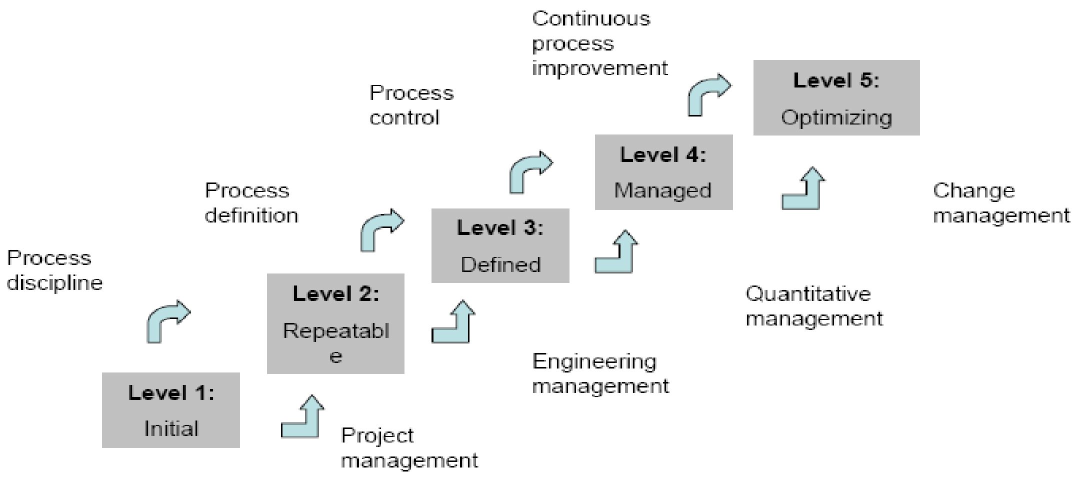


Figure 2.6: Maturity Levels of (SW-) CMM [13]

5. Defect based measures:-

Software quality measurements using decomposition approach requires good planning and data collection. Many software engineers think of software quality in much narrower sense where quality is considered to be only the lack of defects. Defects are interpreted as known errors, fault or failure [3]. A defacto standard measure here is defect density for a given product. Two types of defects are available: 1- known defects: that have been discovered through testing, inspection or another techniques. 2- Latent defects: defects that presents in system but still not appeared. Generally defect density measured though the following equation:-

Defect density= Number of known defects / product size [3].

Product size is usually measured in terms of line of code (LOC). Defect density is an acceptable measure that provides useful measurement information. There are 5 main issues should be considered when using this type of measure which are as follows:

1- Defects can be either a fault discovered during the review and testing, or failure that observed during software operation. Defect count includes:
   a. Post release failure
   b. Residential faults
   c. All known faults [3]

2- Defect rate is the number of defects is being recorded with respect to measure time. This can be an important measure for measuring software reliability.
3- Defect density is calculated using same consistent definition of size.
4- Defect density tells us more about quality of defects than quality of product itself.
5- Even if we are able to know exactly the number of residential faults, we should be careful about making definitive statement of how system will operate in practice because it is difficult to:
    a. Determine in advance the seriousness of faults.
    b. Predict which fault will lead to failure.

Adam in 1984 [15] stated that finding large number of faults may not necessary lead to improve reliability. Reliability is biased on failure data, not faults. It also follows that a very accurate residential faults density prediction may be poor predictors for operational reliability. Some researchers concentrated only on user detected defects, in other words, the defect densities are really failures per unit of size. There are inevitably many dormant software faults that have not yet led to such failure. Japanese companies define quality in terms of spoilage.

Spoilage=Time to fix post release defects / Total system development time [13].

2.3 Software reliability measurement and predictions:-

Most of software quality models identify reliability as a key high level attributes. Quantitative methods for its assessment back to early 1970. It is important to note that no current methods can feasibly assure software system with ultra-high reliability requirements. The basic problem of reliability theory is to predict when system will eventually fail. In hardware we concerned with component failure due physical wear. Such failure is probabilistic in nature, that is, we usually do not know exactly when something will fail, but we know that the product will fail at a particular time. Based on that prediction models are identified to predict when the next failure is [3].

Same approach applies in software. Researchers build basic model of component reliability and create a probability density function (PDF) f of time t (written f(t)) that describes uncertainty about when component will fail [18]. If the software component is equally likely to fail in a given time interval, we can say it is uniform pdf over that interval e.g. [0, x]. On the other hand, if failure time occurs randomly, the function is expressed as exponential function. Having defined a pdf f(t), The probability of component fails in a given time interval [t1- t2] calculated as follows[3]:

Probability of failure between t1 and t2= $\int_{t1}^{t2} f(t)dt$

Usually there is desire to know how long component will behave correctly before it fails. The distribution function (the cumulative distribution function) F(t) is the probability of failure between t1 and t2. Thus reliability can be defined as R(t):

$$R(t) = 1 - F(t)$$

Where *F*(*t*) is the cumulative distribution function. The function generates the probability that the component will function properly up to time t. it is important to note that when probability of failure is high reliability will be low and verse versa. The mean time to failure (MTTF) is the mean of the probability density function. The mean of pdf f(t) computed by the following equation:

$$E(T) = \int tf(t)dt$$

Median time to failure is the point in time t at which the probability of failure after t is the same as the probability of failure before t. We can calculate m that satisfy F(m)=1/2.

$$\mathrm{m} = \mathrm{i}/\lambda \log_{\mathrm{e}} 2$$

The median time to fail gives middle value that splits the interval of failure possibility in two equal parts. We can consider a given interval and calculate the probability that component t will fail in the interval (hazard rate) h(t).

$$h(t) = \frac{f(t)}{R(t)}$$

$h(t)\delta t$ is the probability that the component will fail during the interval [t, t+ δt] in hardware reliability, simply the failed component are replaced by another one. In this situation reliability will be improved this called **reliability growth**. This is a goal of software maintainability. Other measures like hazards rate helped to identify the likely occurrence of a first failure in an interval [3].

A system run successfully, and then fails. The measures have introduced so far have focused on the interruption of successful use. However once a fail occurs there is additional time lost as faults causing failure are located and repaired. Thus, it is important to know the mean time to repair (MTTR) for a component that has failed. Combining this time with the mean time to failure tells how long the system is unavailable for use [16].

MTBF=MTTF+MTTR [16]

This can give a measure about how long system will be available (availability)

*Availability= (MTTF/(MTTF+MTTR)) * 100% [16]*

2.3.1 The software reliability problem:

There are many reasons for software to fail such as: lack of user participation, badly defined system requirements, changing requirements etc. [x]. The key distinction between hardware and software reliability is difference between intellectual failure and physical failure. In software, during long run, reliability accepted to be improved. However, short term decreases caused by ineffective fix or novel faults. Monitoring the time between failures can help in assessing reliability. At a given point in time, the time of next failure is uncertain it is a **random variable [3].**

The previous measures like pdf and F_i and R can be used theoretically to measure reliability. But, the actual values for the functions are unknown. So, the history of failures should be observed firstly. In other word we are not computing an exact time for the next failure, we use the history to help us predict the failure time. It is important to know all attempts done to measure reliability represent prediction problems [3]. To solve predication problem prediction system must be defined and this requires:

1- Prediction model.
2- Inference procedure e.g. Mean time to next failure $1/x = (t_{i-2} + t_{i-1})/2$
3- Prediction procedure.

Many prediction systems have been proposed, some of which use models and procedures very sophisticated.[3]

2.3.2 Parametric Reliability models:

Program is defined here as transformation of inputs (I) to outputs (O). in most cases, a complete description of input space is not available. The output consist of two types those are acceptable and those are not [3]. The program will fail if the input doesn't transform to an acceptable output. There are two sources of uncertainty in the failure behavior:

1- Uncertainty about the operational environment
2- Uncertainty about the effect of faults removal.

Good reliability model should address both types of uncertainty. The most difficult problem is to model uncertainty type 2. Following is the most known reliability models:

a) The Jelinski – Moranda model:

The model is the earliest and probably the best known reliability model (1972)[17]. It assumes that, for each I,

$$F_i(t_{i)=1-e^{-\lambda_i t_i}}$$

$$\lambda_i = (N - i + 1)\Phi$$

N is the initial number of faults and ϕ is the contribution of each fault to overall failure rate. The model is exponential, so type 1 uncertainty is random and exponential. There is no type-2 uncertainty in this model. It assumes that fault detection and correction begins when program contains N faults. And that fixes are perfect. And also all faults have the same rate. The interface procedure for the model is called maximum likelihood estimation [3].

There are three related critics on the model:

1- The sequence of rates that considered by the model is deterministic and this is not always realistic.
2- The model assumes all faults contribute equally to hazards rate.
3- Poor reliability prediction (too optimistic) [17].

b) Other models based on Jelinski – Miranda [17]

Several models are variations of Jelinski-Moranda. Shooman's model is identical (1983). The Musa model extend Jelinski model. It introduces some novel features on top of previous model. It was the first model insists on using execution time to capture inter-failure times. Musa model encourage using of reliability model especially on communication [3].

c) The Little wood model

Attempts to be more realistic finite fault model than jelinski by treading hazard rates as independent random variables. These rates are assumed to have a gamma distribution with parameters (β,α). Unlike jelinski this model introduces two sources of uncertainty for the rates. Both jelinski and little wood models are in general called exponential order static models [18]. In this type of model the faults can be seen as competing risks at any point in time. The distribution of little wood model equals:

$$\boldsymbol{P(X < x) = 1 - \left(\frac{\beta}{\beta + x}\right)^{\alpha}}$$

d) The Little wood - Verrall model [3]

Is the simple model similar to little wood model. It captures the nature of the conceptual model of failure process. The assumption here is that the inter failure time T_i, are conditionally independent exponentials with probability density functions:

$$pdf\ (t_i|\hat{}i = ƛi) = ƛ_i e^{-ƛi\ ti}$$

$$pdf\ (ƛi) = \frac{\psi\ (\mathrm{i})^{\alpha-1} e^{-\psi(\mathrm{i})ƛ}}{\acute{\Gamma}(\alpha)}$$

e) Non- homogenous Poisson process models

Non homogeneous passion process (NHPP) is the way to model process that is statistically independent of the past [3]. It determined by failure occurring time. A minor drawback is that such process have rate function that change continuously in time. This is not real for software.

Others models found like Goel-Okumoto model is a NHPP variant of Jelinski model. Similarly, the little wood NHPP model is variant of original little wood model. All the above models are parametric models, in the sense that they are defined by values of several parameters. Using these model involve 2 steps: selecting the model then estimating the values of its parameters. Some researchers use different approach to estimate the parameters by using Bayesian posterior distribution of known parameters. On the other hand, predictive accuracy can be analyzed for the models then select the best working one. Unfortunately, these techniques work effectively only if software's future operational environment is similar to the one in which the failure data was collected. Worse still, there is no current methods that are feasibly assuring software system with ultra-high reliability requirements [3].

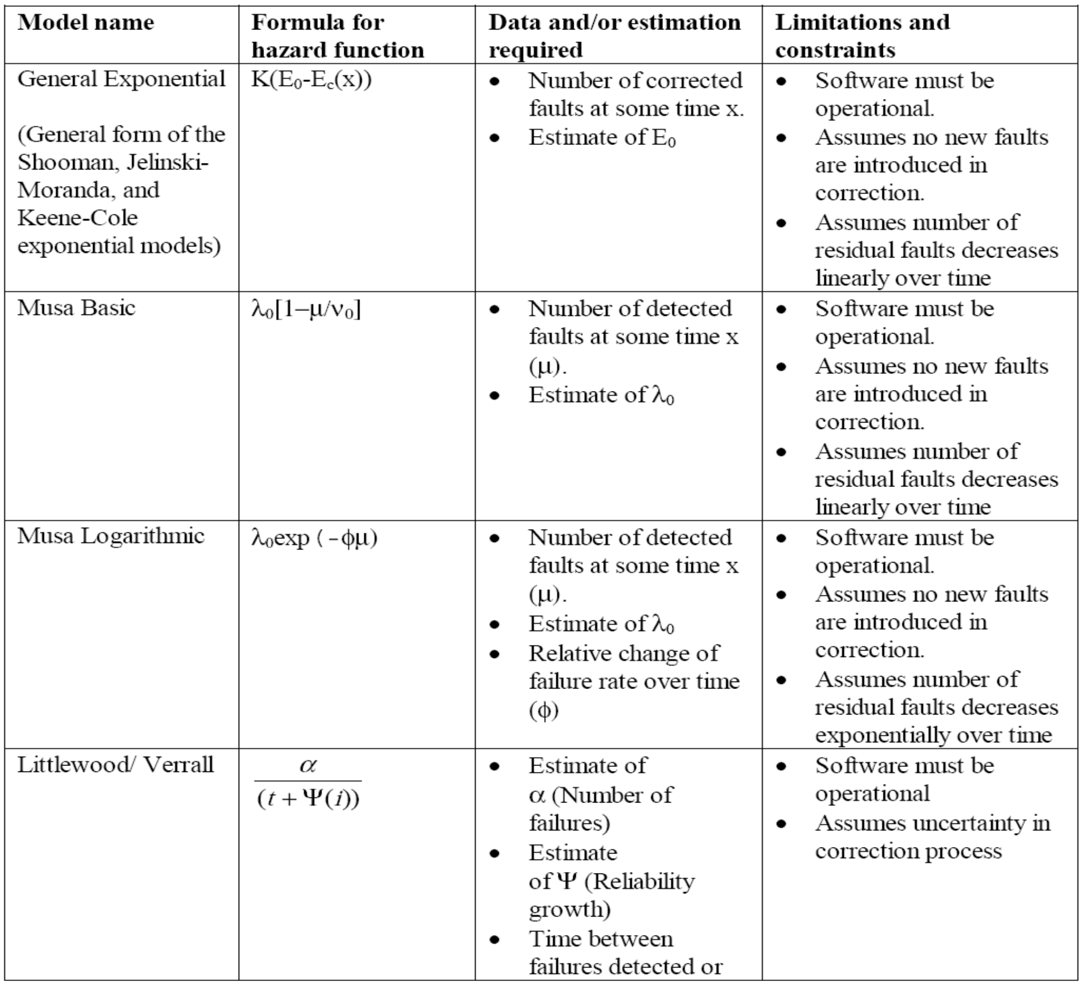
2.4 Software Reliability Growth Modeling/Testing.

Reliability growth for software is the positive improvement of software reliability over time, accomplished through the systematic removal of software faults. The rate at which the reliability grows depends on how fast faults can be uncovered and removed. A software reliability growth model allows project manager to track the progress of the software's reliability through statistical inference and to make projections of future milestones. If the assessed growth falls short of the planned growth, management will have sufficient notice to develop new strategies, such as the re-assignment of resources to attack identified problem areas, adjustment of the project time frame, and re-examination of the feasibility or validity of requirements. Measuring and projecting software reliability growth requires the use of an appropriate software reliability model that describes the variation of software reliability with time. The parameters of the model can be obtained either from prediction performed during the period preceding system test, or from estimation performed during system test. Parameter estimation is based on the times at which failures occur [3].

The use of a software reliability growth testing procedure to improve the reliability of a software system or to a defined reliability goal implies that, a systematic methodology will be followed for a significant duration. In order to perform software reliability estimation, a large sample of data must be generated to determine statistically, with a reasonable degree of confidence that a trend has been established and is meaningful [3].

There are several software reliability growth models available. Table 2-4 summarizes some of the software reliability models used in industry.

Table 2-4: Software reliability model [19]



Model name	Formula for hazard function	Data and/or estimation required	Limitations and constraints
General Exponential (General form of the Shooman, Jelinski-Moranda, and Keene-Cole exponential models)	$K(E_0-E_c(x))$	• Number of corrected faults at some time x. • Estimate of E_0	• Software must be operational. • Assumes no new faults are introduced in correction. • Assumes number of residual faults decreases linearly over time
Musa Basic	$\lambda_0[1-\mu/\nu_0]$	• Number of detected faults at some time x (μ). • Estimate of λ_0	• Software must be operational. • Assumes no new faults are introduced in correction. • Assumes number of residual faults decreases linearly over time
Musa Logarithmic	$\lambda_0 \exp(-\phi\mu)$	• Number of detected faults at some time x (μ). • Estimate of λ_0 • Relative change of failure rate over time (ϕ)	• Software must be operational. • Assumes no new faults are introduced in correction. • Assumes number of residual faults decreases exponentially over time
Littlewood/ Verrall	$\frac{\alpha}{(t+\Psi(i))}$	• Estimate of α (Number of failures) • Estimate of Ψ (Reliability growth) • Time between failures detected or	• Software must be operational • Assumes uncertainty in correction process

The following checklist determines which model or models to choose from given the following constraints. This checklist is summarized as follows:

- Failure profiles
- Maturity of software product
- Characteristics of software development
- Characteristics of software test
- Existing metrics and data [3]

2.5 Software Metrics

The software metric is the measurement, usually using numerical ratings, to quantify some aspects or attributes of a software entity [4]. Typical measurements include the quality of the source codes, the development process and the accomplished applications. The field of software metrics is a relatively young one, whose origins can be found in the work by Halstead published in 1972. Software metrics allow to use a real engineering approach to software development, providing the quantitative and objective base that software engineering was lacking. In fact, their use in industry is becoming more and more widespread. Good metrics should enable the development of models that are efficient of predicting process or product spectrum. Thus, optimal metrics should be [4]: Simple, Objective, Easily obtainable, valid and Robust. As shown in table 2.1 software metrics are classified mainly into:

1- **Process metrics:**

Metrics highlights the process of software development. It mainly aims at process duration, cost incurred and type of methodology used.

2- **Project or resources Metrics:**

Project metrics are used to monitor project situation and status. And identify risk. E.g.. Staff number and their patterns, cost, etc…

3- **Product Metrics**:

Product metrics describe the attributes of the software product at any phase of its development.[9]

Software quality attributes has to be evaluated through considering different views such: users, manufacture, Product and value based view. This should be measured by different users of different roles. The following tables (2-5) and (2-6) summarizes these issues.

Table 2-5: Software quality attributes evaluation against different views

Views	Description
User view	evaluates the software product against the user’s needs
Manufacturing view	Concentrates on the production aspect of the software product.
Product view	Take a look at the internal features of the products.
Value based view	This becomes important when there are lots of contrasting views, holds from different departments

Table 2-6: Different Measurements in terms of different roles.[9]

Role	Measurements
User	Usability, simplicity, Stability, Cost….
Designer	Extendibility, scalability, Manageability…
Programmer	Complexity, Maintainability….

Different classification criteria have been introduced for software metrics according to what has been measured. Main types of metrics can be categorized such as code, Programmer productivity, Design, Testing, Maintainability, Management, Cost, Duration, time, Staffing metrics [9].

Metrics are described as direct or indirect. The distinction between direct and indirect metrics is based on the way a metric is measured. Size for example, can be directly measured whereas quality or complexity can only be measured indirectly by breaking them down into different aspects. Most metrics are indirect. This must not be confused with the distinction between primitive and computed (derived) metrics. Primitive metrics provide raw data, "physical" attributes of the software that are later used as inputs for computed metrics [9]. Such attributes are:

- Bugs: can simply be counted as they are found and fixed, bugs can be interpreted as the number of corrections resulting from a review.
- Cost/Effort: used to calculate critical measures that is important for evaluating e.g. an organization's position with respect to its competitors and the market
- Duration: This refers to both the duration of either all or part of certain process.
- Size: Software size is probably the most important primitive metric can be calculated directly from LOC.
- Line of code: The traditional way of measuring program size is by counting lines of code (LOC).
- Function Point Analysis: Developed by A.J. Albrecht of IBM in 1997, this approach tries to eliminate some of the disadvantages of LOC by deriving the size of a program not from the code but from its (specified) functions as viewed by the user. This leads to a metric which is independent of the programming language and technology used. Thus, it can be used to normalize and compare results from different environments [20].
- Halstead's metrics: Devised in the 70's by Maurice Halstead [21], this is a very formal approach to define program size and derive various estimates. It

is not really a primitive metric but as it measures size similar to LOC it fits here and makes for a nice transition to computed metrics.

Computed Metrics are derived from primitive metrics such as:

- Complexity: metrics concerns with measuring software complexity e.g FP.
- McCabe's Cyclomatic Complexity: Amongst the most popular methods to measure implementation complexity is the cyclomatic complexity defined in McCabe76 [9]. His approach is based on the control flow graph.
- Productivity: According to [9], productivity is measured as the amount of work (size) completed with a given effort, where "completed" usually means "has passed quality control",
- Quality: Like complexity, quality is not easy to define, much less to measure. A common metric that can be found in [9] defines quality as the degree to which a product is bug-free.

Software quality is the degree to which software possesses a desired combination of attributes such as maintainability, testability, reusability, complexity, reliability, interoperability, etc. In other words, quality of software products can be seen as an indirect measure and is a weighted combination of different software attributes which can be directly measured. Moreover, many practitioners believe that there is a direct relationship between internal and external software product attributes. For example, a lower software complexity could lead to greater software reliability [3].

2.5.1 Syntactic metrics

Syntactic metrics reflect how programs are represented in source code, but not what functions programs define. The terms metric and measure have some overlap. We use measure for more concrete or objective attributes and metric for more abstract, higher-level, or somewhat subjective attributes. For instance, a line of code (LOC) is a measure: it is both objective and concrete [22]. Researchers investigate four types of measures based on different criteria:

A) Length measurements

Metrics use this kind of measure focus only on code length without taking into account software complexity. Such as:

- **Line of code (LOC):** it is a traditional way of measuring program size by counting its number of lines. All lines are counted except comments and empty lines [22]. Although, this is the easiest way to measure program length, it doesn't give an accurate measure of actual program length in terms of time and effort.
- **Number of signs**:
  LOC does not take into account any factors other than total number of lines. Number of lines doesn't represent an accurate measure for program length. To overcome this limitation the metric focus on code content rather than total number of lines. It counts number of operands and operators as follow: n1 is the number of operations, N1 total operation frequency, n2 is the number of operands and N2 is the total operands frequency [22]. All these parameters are used in program length calculation.

B) Depth measurements

This measurement considers code complexity regardless of it is length. It depends on the concept of having two programs with same length and different complexity [22]. Such as:

- McCabe: Cyclomatic complexity was developed by Thomas J. McCabe, Sr. in 1976 and is used to indicate the complexity of a program [23]. This metric is amongst the most popular methods to measure implementation complexity [23]. It represents program in a control flow graph. The nodes of the graph correspond to indivisible groups of program commands, and a directed edge connects two nodes if the second command might be executed immediately after the first command. Number of edges and nodes are used to calculate the following equation:

  $$V=e-n+1.$$

  Where e is the number of edges and n is the number of nodes. Researches confirmed that McCabe metric can be used to give a glance of faults density.[23]

- **Size measurements:**

These metrics focused on both program length and depth to measure complexity. One of the famous metrics that uses this method is Halsted metric [21]. Halstead suggests a measure to program length by using the following equation:

$$N = n1x \log2(n1) + n2 \times \log2(n2).$$

Where n1 is the number of operators and n2 is the number of operands in the code. In addition to that to calculate the program size Halsted metric represents the program as a message written by a programmer. According to that if we need to calculate the actual value for this message we must calculate H (n) [21]. Where n is the number of symbols and H is the massage name.

$$H(n) = \log2(n)$$

Based on that we can conclude to the result that number of symbols = n1 + n2.

- **Data measurements:**

Data measurement aims to measure the size and complexity of the program structure. It should be noted that the size of the program may differ according to the type of programming language [22].

Size = minimum size / actual size.

By other way we can calculate the density of the program data by calculating the number of known variables within the program [22]. These may contribute to estimate the effort that made by the software programmer.

- **Design measurement**

There are two main concepts introduced here; Cohesion and coupling. Cohesion reflects the extent to which internal elements of the system are related to each other. Where, coupling cares about the relation between different partial components.

Design quality = high cohesion + low coupling.

All the previous measurements fall in Syntactic metrics field. Which reflect attributes of the source code text; they do not reflect attributes of the execution of the program [22].

2.5.2 Semantic Metrics

Most software metrics are based on program structure and are determined statistically [24]. Nowadays, there is a great move towards the semantic metrics which reflect what functions the software product defines, rather than how these functions are represented. Semantic metrics are based on the meaning of software within the problem domain. Researchers use semantic metrics to provide insight into software quality early in the design phase of software development. Others extend semantic metrics to analyze design specifications. In spite of the success of semantic metrics in software quality field, but a few number of studies touch this issue. Chapter 2 and 5 will discuss more about this issue

2.6 Chapter Summary

This chapter gives an overview to a background of the study field. Major concepts and terminologies used were discussed started from general software measurement concepts along with software quality models such as McCall, Boehm, Dromey, CMM and ISO 9126. Then the chapter introduced software reliability measurement started from reliability problem and parametric reliability models, table 2.7 summarizes the parametric reliability models. Finally, both syntactic and semantic metrics were discussed.

Table 2.7: Summary of the main parametric reliability models

	Parametric reliability models	**Measurement Formula**
1	**The Jelinski – Moranda model**	$F_i(t_{i)=1-e^{-ƛ_i t_i}}$ $ƛ_i = (N - i + 1)\Phi$
2	**The Little wood model**	$P(X < x) = 1 - \left(\frac{\beta}{\beta + x}\right)$
3	**The Little wood - Verrall model**	$pdf\ (ƛi) = \frac{\psi\ (\mathrm{i})^{\alpha-1} e^{-\psi(\mathrm{i})ƛ}}{\acute{\Gamma}(\alpha)}$

Chapter Three

Related work

3.1 Introduction

The previous chapter highlights a general overview on current used metrics. Moreover, it surveys the most famous used kind of metrics - complexity metrics- that are widely used. The next section will shed some light on semantic metrics research and current research directions.

3.2 Current Research in software metrics

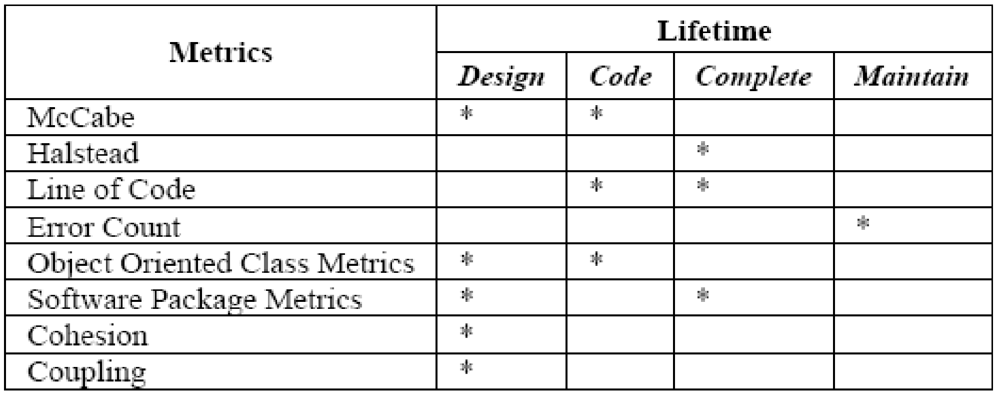
3.2.1 Complexity metrics

Complexity metrics can measure the degree of software difficulty. Measuring complexity of software products was, and still is, a widely distributed research subject. The scope of studying it was to control the levels of the external attributes of software via internal attributes, like complexity is. The most well-known internal attribute is software length. While in the case of length is a quite well defined consensus about the ways the length should be measured, in the case of complexity is still a lot of confusion [25]. It is not wrong to say that there is a relationship between complexity and the length of the program. But, all authors agree that when measuring complexity one should take into account other internal attributes in addition to, length itself. This approach was discussed by Törn et al. [26] where a new measure of software complexity called structural complexity is derived. The authors use the equations that combine between code length and structure complexity for the software collections and define new formulas that use some constants. In which one control structure assign different value from the others.

In [4] traditional complexity metrics are investigated. They divide complexity of software into three classes: the essential complexity, the selecting complexity and the incidental complexity [25].

The essential complexity is determined by the problems that the software tries to solve. The selective complexity is determined by the program languages, the problem modeling methods and the software design methods. The incidental complexity is determined by the quality of the involved implementer [25]. the following table shows the classification of complexity metrics used during software life cycle.

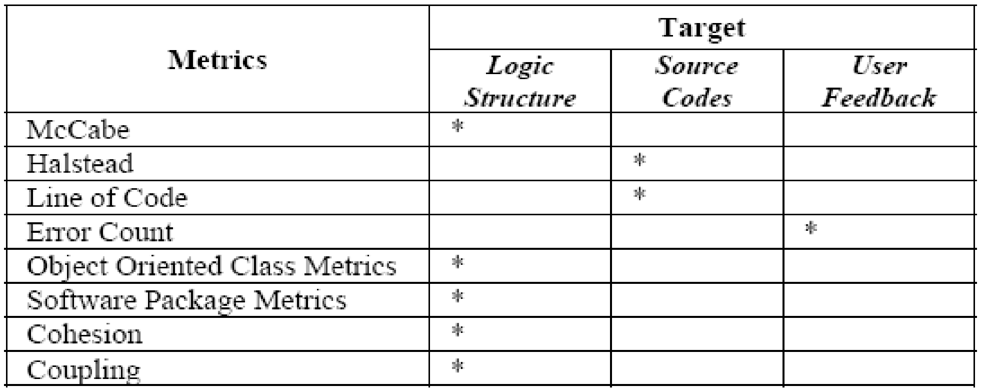
Table 3.1: The classification of complexity metrics
Used during software life cycle [25]



Metrics	**Lifetime**			
	Design	***Code***	***Complete***	***Maintain***
McCabe	*	*		
Halstead			*	
Line of Code		*	*	
Error Count				*
Object Oriented Class Metrics	*	*		
Software Package Metrics	*		*	
Cohesion	*			
Coupling	*			

Same metrics are classified based on the way of complexity calculation. Table 2.3 considers this issue.

Table 3.2: The classification of the complexity metrics by their calculation Basis



Metrics	**Target**		
	Logic Structure	***Source Codes***	***User Feedback***
McCabe	*		
Halstead		*	
Line of Code		*	
Error Count			*
Object Oriented Class Metrics	*		
Software Package Metrics	*		
Cohesion	*		
Coupling	*		

The study compares between traditional used metrics such as LOC which count lines of code. Researchers in [27] found that there is a relationship between code lines and bug density. Halstead metrics was introduced in 1977 by Maurice Halstead. HCM calculate number of operators and operands to measure program quality and complexity and based on these inputs it calculate difficulty, software length, volume, error estimation and time. One of it is main advantages that it doesn't require deep knowledge of program logical structure so it is easy to calculate but in the other hand, it doesn't give accurate measure because it doesn't consider program flow control [27]. other metrics are discussed such as WHCM which overcome the limitations of HCM. WHCM adds weight of the code instructions such as loops or branches. The WHCM takes the documents of the project into consideration. The

WHCM uses the Capability Maturity Model for Software (SW-CMM) to measure the project's documents and modify the HCM. In [23] Thomas J. McCabe introduced a software complexity metric named McCabe Cyclomatic Complexity Metric.

Tu Honglei et.al.[28] investigate the efficiency from using complexity metrics like McCabe and CK metrics. The selection of these metrics is done after comparative study between different available type's pf complexity metrics. As an extension for this study researchers try to expand the evaluation of complexity metrics. the main intent of their study was to compare three proposed code complexity metrics: McCabe's cyclomatic complexity, Halstead's software science and Shao and Wangs' cognitive functional size and identify which metric is the most suitable metric that can be used in the current state of the art with the help of thirty programmers. To conduct this empirical study ten freely available java programs were used as the base. From this study it was identified that Shao and Wangs' cognitive functional size is the best complexity metric that can be used in the real world [28].

3.2.2 Measuring the Complexity of web application:

Another direction of measurement is web based application (WAs). Web applications are similar to other software in that they have business logic in application domains, however, there are several characteristics that differ from traditional software [29]: WAs have hypertext structure, dynamically generate codes, and rapid evolution is required [29,30]. For these reasons, it is hard to apply existing metrics to WAs, and new metrics for WAs should be defined. In existing maintenance approaches, structural systems or object-oriented systems are the main focus and Web applications are not often considered [30]. Several studies have been conducted for a complexity measure; however, most studies have focused on the complexity of traditional software rather than the complexity of WAs. Zhang *et al.* proposed a navigational complexity measure for the web using a navigational structure and the number of links, from a user's point-of-view [31]. Mendes *et al.* introduced a count-based complexity measure of web applications [32]. However, there are some cases where those count-based measures cannot handle well. In [33], a web application is modeled as a graph composed of nodes and weighted edges. Jung *et al*. [34] assumed that the information quantity of a frequently referenced page is larger than that of an infrequently referenced page, when a maintainer reviews web pages statically.

3.3 Metrics for measuring software reliability

The major goal of a research in software engineering is to improve the quality of software. One of the most important quality attributes is reliability. The generally accepted definition of software reliability is the probability of a failure-free operation of software component or system in a specified environment for a specified time [35]. Although the reliability estimation is the goal of many researchers and also a wish of many customers, it is seldom used in the practice. There are other useful measures which do not yield a probability of failure but other figures related to the reliability attributes. For this reason authors decompose reliability attribute into four reliability parameters such as: Probability of a failure free operation R(t) , Mean time between failures MTBF, Failure intensity z(t) and Number of errors left in software N(t). The main goal of this research is to prove empirically that errors are correlated with the reliability parameters. Probably one of the most comprehensive factor investigations was made by Schneidewind [36]. He tried to determine the relationship between several complexity measures and different error characteristics. In his experiment, different categories of errors were taken into consideration. Many types of errors were defined within particular categories: design errors, coding errors, clerical errors, debugging errors, and testing errors. The total number of different types of errors was 63. On the same context, M. Takahashi and Y. Kamayachi studied the relationship between errors remained in the program and ten error factors [37] they considered the following factors such as frequency of program specification change, programmer's skill, organization and program category, difficulty of programming, amount of programming effort, volume of program design documents, levels of programming technologies, program complexity (McCabe's and Halstead's metrics) and percentage of reused modules. In this empirical study, it was experimentally found that there is statistically significant relation between the number of unsuccessful compilations (UC) and a simple reliability parameter (TVS - number of Tested Versions of Software). Therefore, UC can be used as an indicator of error proneness.

In [38] Eduardo, Constitutes a review of the State of the Art techniques that helps to improve the Software reliability. Such techniques are classified into 3 different categories: Fault avoidance, Fault detection and Fault tolerance. To grantee reliability the selection must include a combination of fault avoidance techniques, fault detection techniques and if required, fault tolerance. Another attempt to measure software reliability is done by Zeeshan Ahmed and Saman Majeed[39]. The aim of his research is to address the importance of preprocessed source code and project artifact measurement for better reliability analysis of software application by visualizing obtained results in different diagrams to take advantage in analyzing over all behavior of software project by predicting the level of complexities at different stages and estimating the rate of fault of proneness as well [39].

In [40] authors move to another quality issue by investigating how to reduce software maintenance costs by applying software renovation tools and methodologies which will improve the program's intrinsic quality. Their basic idea of the methodology presented here is to establish a diagnosis based on the program quality analysis. The use of the alone static analyzers to evaluate software quality is insufficient. Therefore, they suggest a quality analysis based on metrics but supplemented by "checklists" covering all the quality criteria of the programs and taking into account the semantic aspects which are not covered by the static analyzers. A knowledge-based system which integrates both semantic and syntactic aspects is proposed to implement such checklist. Their results show the ability of suggested tools whether the program has to be reengineered only or redevelop completely. In the same context Basson and Derniame [40, 41] have developed a kernel of software quality metrics devoted to Ada language. Nowadays, new metrics appear especially designed to measure specific aspects of object oriented languages such as C++ [38]. In [42] the approach to the estimation of program reverse semantic traceability (RST) influence on program reliability with assistance of object-oriented metrics is proposed. Their paper shows how to change the software reliability model parameters, that was received using logistic regression, in order to estimate influence of program RST on program reliability. Experiments show promising results.

Jerey M. Voas et.al.[43] Touch an important factor to increase software confidence; software testing techniques such as black box testing can be used to grantee that the software no longer contains faults. Authors use lack of faults as a measure of software quality. They also introduce Hamlet’s probable correctness model to assess confidence that the true failure probability of the program is less than some preset threshold. Research considers sensitivity analysis's predictions which are based upon repeated executions. They concentrate on estimating and controlling testability before it is written, during the design phase [43]. Ordonez et al. [44] examined various metrics used in software industry to measure code size and design complexity. They mentioned that NASA used the first five metrics presented in [43] in the tool they developed for analyzing source code with respect to its architecture. The author’s analysis was focused on how reliable are specific software modules with respect to their maintainability and the probability of causing defects.

3.4 Data mining techniques for semantic metrics:

There are strong connections among the metrics. But that doesn’t mean that we can replace one metric with other one. The article [45], using a data miner tool called Multi method, did some experiments on three data sets in the Metrics Data Project (MDP) of the NASA. The result shows, the effect of the software defect prediction model integrating kinds of metrics, is much better than that using only one. Using multiple metrics in the prediction or detection process may increase the accuracy and thus increase software quality. Sallie.et.al. [46] offer a good comparative study between code, structure and hybrid metrics. The Study has shown that structure and hybrid metrics are extremely useful at design time. Moreover, the use of prediction model can help to determine the complexity of the resultant code.

Data mining is positioned between different research domain as statistics, machine learning, database management and data visualization. It is defined as the process of identifying valid, novel, potentially useful, and ultimately comprehensible knowledge from data, used to help by crucial decision making. Current software quality estimation models often involve use of data mining and machine learning techniques for building a software fault prediction models. In [45] Mertik, M., et.al. Address this issue. They achieved better results by building the fault prediction model as with standard machine learning methods. Special data mining tool – Multimethod- has been used. They present the case study of building the fault prediction model based on the data from the Metrics Data Program Data Repository. They get benefit from the data mining researches that reaches to truth that using different techniques (algorithms) in data mining may improve the accuracy of detection / classification model [47]. So, different approaches have been employed during their study. In [48] Salwa K conducts the use of data mining in detection of function clones in software systems. She presents an efficient metrics-based data mining clone detection approach. First, metrics are collected for all functions in the software system. A data mining algorithm, fractal clustering, is then utilized to partition the software system into a relatively small number of clusters. Each of the resulting clusters encapsulates functions that are within a specific proximity of each other in the metrics space. Finally, clone classes, rather than pairs, are easily extracted from the resulting clusters. For large software systems, the approach is very space efficient and linear in the size of the data set. Evaluation has been done using medium and large open source software systems. The investigation of results reflects good improvement. T Menzies [49] touch a hot issue by investigating the use of data mining to generate defect predictor from static code attributes. Many researchers use static attributes to guide software quality predictions [49, 50, 51]. They take into their consideration the use of McCabe versus Halstead versus lines of code counts previously and they tried to compare between them and the new proposed method. Their predictors achieve good detection accuracy that reach to a mean probability of detection of 71 percent and mean false alarms rates of 25 percent.

3.5 SEMANTIC METRICS

Most software metrics are based on program structure and are determined statically [3]. Nowadays, there is a great move towards the semantic metrics which reflect what functions the software product defines, rather than how these functions are represented.

In 2008 Gall, C. S [52] suggests an approach using semantic metrics to provide insight into software quality early in the design phase of software development by automatically analyzing software specifications for object oriented system using natural language (NL) processing. In [53], researchers extend semantic metrics to analyze design specifications. Since semantic metrics can now be calculated from early in design through software maintenance, they provide a consistent and seamless type of metric that can be collected through the entire lifecycle. A comparison was done to compare semantic metrics from different phases of the lifecycle with syntactically oriented metrics calculated from the source code. Another direction has been touched related to the semantic is “web semantic” which is related to web applications.

In the context of the Semantic Web, many ontology-related operations, e.g. ontology ranking, segmentation, alignment, articulation, reuse, evaluation, can reduce to one fundamental operation: computing the similarity and/or dissimilarity among ontological entities, and in some cases among ontologies themselve [54]. Bo Hu et.al.[55] gives formal account of semantic metrics drawn from a variety of research disciplines, and enrich them with semantics based on standard Description Logic constructs. Authors argue that concept-based metrics can be aggregated to produce numeric distances at ontology-level and they speculate on the usability of their ideas in potential areas. Zschaler in 2004 [56] define elements of a semantic framework for non-functional specifications of component-based systems. Framework focuses on how the runtime environment uses components, whose non-functional properties have been specified. It is notable that very little research has been performed concerning non-functional properties.

In 2010 B.Wen. and L. Zhang [57] make good contribution to the area of semantic measurement. The paper presents a method to map process metrics to the enterprise information model automatically when the semantic features of metric are analyzed, a structure called semantic tree is defined with domain ontology. B.Gabriele. [58] Moves to another direction to consider the use of semantic information for software modularity. He proposes a new technique for automatic re-modularization of packages, which uses structural and semantic measures to decompose a package into smaller, more cohesive ones. The paper presents the new approach as well as an empirical study, which evaluates the decompositions proposed by the new technique. The results of the evaluation indicate that the decomposed packages have better cohesion without a deterioration of coupling and the re-modularizations proposed by the tool are also meaningful from a functional point of view [59]. In particular, Maletic and Marcus (2001) combined semantic and structural measures to identify ADTs in legacy code. They used Latent Semantic Indexing (LSI) [60], an Information Retrieval (IR) technique, to capture semantic relationships between source artifacts.

Emanuel et.al. consider how to predict code changes [61]. Fine-grained source code changes (SCC) capture detailed code changes and their semantics on the statement level. They explore prediction models for whether a source file will be affected by a certain type of SCC. These predictions are computed on the static source code dependency graph and use social network centrality measures and object-oriented metrics. The results show that Neural Network models can predict categories of SCC types. In summary, the results of their work confirm the empirical findings regarding the relation between coupling and changes.

Historically, software evolution has been studied at the file level. Lehman already used in his laws of Software Evolution the number of files as a measure of system growth [62]. Other authors have used SLOC (source lines of code) for the same goals. Gregorio et.al. [63] Tried to move from file/SLOC (physical) level to functions (physical and semantic) level to gain better understanding of the evolution of a software project. Their point of view, considering functions is closer to the way developers work and conceive a software system. They addressed two metrics from software evolution research that have already been studied at the file level, modification patterns and developer territoriality (also known as code ownership), but this time considering a granularity of functions.

New trend has introduced by Selim et.al. They tried to extend the concept of using semantic information to improve software quality. Authors consider software component identification from the code [64]. It is worth saying that it is one of the primary challenges in component based software engineering. Most of the software component identification techniques [65, 66, 67] start from semi-formal domain business models (Typically expressed in UML) and produce domain software components. Authors propose an approach for identifying components based on a fitness function to measure the quality of a component. To evaluate such function, they use a semantic- correctness model defined in their previous works [65,68]. Their approach gives to the architect the choice between two strategies to identify components. The first strategy is explorative. The second strategy is requirement-driven.

Some studies shed light on how to integrate entropy concept with semantic aspects of software as quality measure. Such concept dates back to 1997 when D. Melamed defines semantic entropy as the measure of semantic ambiguity and [69]. It is a graded lexical feature which may play a role anywhere lexical semantics plays a role. The study proposed a method for measuring semantic entropy using translational distributions of words in parallel text. The measurement method is well-defined for all words, including function words, and even for punctuation. The hypothesis behind the measurement method is that semantically heavy words are more likely to have unique counterparts in other languages, so they tend to be translated more consistently than semantically lighter words. Brown et al. [70] present a word-sense disambiguation algorithm involving minimization of semantic entropy weighted by word frequency. Yarowsky[71] compares the semantic entropy of homographs conditioned on different contexts. Another way to use semantic that developers of interlinguas for machine translation can use semantic entropy to predict the required complexity of lexical elements of the representation. Another interpretation of entropy is as the inverse of reliability. Machine learning algorithms may benefit from discounting the importance of data that has high entropy. Semantic entropy can help researchers decide not only how to work with words, but also which words to work with. Several applications in computational linguistics use stop-lists of unusual words. Salwa K and Abd-El-Hafiz, in 2004 [72] also address entropy as a means to measure software information content. They use the entropy metrics to study the evolution of the modules within the system. Such an analysis provides a deep understanding of the evolution of a software system. This study supported the use of entropy concept in the measurement of some software attributes.

In the context of using object Oriented (OO). The class is the basic term of concern, not the procedure or statement. Hence, the metrics used to measure such software should be class-oriented. A study in 2008 was empirically investigated the suite of object-oriented (OO) design metrics introduced in [73]. More specifically, their goal is to assess these metrics as predictors of fault-prone classes and, therefore, determine whether they can be used as early quality indicators. Their study represented as complementary to the work described in where the same suite of metrics had been used to assess frequencies of maintenance changes to classes. To perform validation accurately, they collected data on the development of eight medium-sized information management systems based on identical requirements. All eight projects were developed using a sequential life cycle model, a well-known OO analysis/design method and the C++ programming language. Based on empirical and quantitative analysis, the advantages and drawbacks of these OO metrics are discussed. Several of Chidamber and Kemerer's OO metrics appear to be useful to predict class fault-proneness during the early phases of the life-cycle. Also, on their data set, they are better predictors than “traditional” code metrics, which can only be collected at a later phase of the software development processes. On the other hand Larry J in [52] focus on what is the suitable semantic information should be considered during measurement. In [74] researchers proposed the design complexity of object-oriented software with Weighted Methods per Class metric (WMC-CK metric) expressed in terms of Shannon entropy, and error proneness. CK suite of metrics has been successfully applied in identifying design defects early during the design process. The analysis showed that components with high design complexities were associated with more maintenance activities than those components with lower class complexities.

In 1993 [8] some researches make spot on software faults that infrequently affect output cause problems in most software and are dangerous in safety-critical systems. When a software fault causes frequent software failures, testing are likely to reveal the fault before the software is released; when the fault “hides” from testing, the hidden fault can cause disaster after the software is installed. During the design of safety-critical software, certain sub functions of the software can be isolated and that tend to hide faults. A simple metric, derivable from semantic information found in software specifications, indicates software sub functions that tend to hide faults. The metric is the domain/range ratio (DRR): the ratio of the cardinality of the possible inputs to the cardinality of the possible outputs. By isolating modules that implement a high DRR function during design, programs that are less likely to hide faults during testing can be produced. The DRR is available early in the software lifecycle; when code has been produced, the potential for hidden faults can be further explored using empirical methods. Using the DRR during design and empirical methods during execution represents a better plan and implements strategies for enhancing testability. For certain specifications, testability considerations can help produce modules that require less additional testing when assumptions change about the distribution of inputs. Such modules can be seen as good candidates for software reuse. Norman in [6] found relationship between faults density and module size and analysis time thorough his study. He confirmed that the number of faults discovered in pre-release testing is an order of magnitude greater than the number discovered in 12 months of operational use. Marcus etal. In [58] try to improve this study by suggesting a way for predicting software faults in OO programs. They suppose that High cohesion is a desirable property of software as it positively impacts understanding, reuse, and maintenance. Currently proposed measures for cohesion in Object-Oriented (OO) software reflect particular interpretations of cohesion and capture different aspects of it. Existing approaches are largely based on using the structural information from the source code. Their study proposes a new measure for the cohesion of classes in OO software systems based on the analysis of the unstructured information embedded in the source code, such as comments and identifiers. The measure, named the Conceptual Cohesion of Classes (C3), is inspired by the mechanisms used to measure textual coherence in cognitive psychology and computational linguistics. This study presents the principles and the technology that stand behind the C3 measure. A large case study on three open source software systems is presented which compares the new measure with an extensive set of existing metrics and uses them to construct models that predict software faults. The case study shows that the novel measure captures different aspects of class cohesion compared to any of the existing cohesion measures. In addition, combining C3 with existing structural cohesion metrics proves to be a better predictor of faulty classes when compared to different combinations of structural cohesion metrics.

Different concept has been discussed by B Neate et.al. In 2006 [74] they turned toward measuring of the relative importance of components within the software structure which was examined in [29]. The use of page rank concept has proved it's successful in allowing search engines to identify important pages in the World Wide Web. The authors suggested a new family of metrics, Code Rank, based on the same concept used by the Google search engine [29] for ranking web pages. Software is modeled as a graph whose nodes represent components of various granularities (package, class, method,…etc) and whose edges indicate dependencies (invocation, inheritance, overriding, . . .) between components. Metric values are assigned to nodes according to an intuitively-appealing model which describes the process in terms of rank flowing through the graph edges. Interpretation of the CodeRank metric values indicates such things as the "importance" of a component, its coupling to the remainder of the system and the extent to which it is reused. Experiments prove the usefulness of the proposed metrics in different applications. In an earlier work [75], for the same purpose the authors suggested to use a similar metric called COMPONENT RANK. The main difference between these metrics is that the CODERANK is computed based on the weighted graph that represents various usage relations between the components and the number of time each usage occurs. This research is consistent with the finding that PAGERANK is an informative metric.

Zhou applied PageRank and HITS algorithms to measure the importance of classes [76]. Yi proposed metrics for measuring complexity of relationships among classes [77]. In his work, he proposed classes as web pages and relationships among classes as links among web pages. He inferred complexity of relationships according to the PageRank algorithm.

Lorenz et al. [78] recommend using a wide range of metrics to test the quality of models, classes and methods. Various metrics related to coupling, inheritance and size of classes and methods play the major role in deducting the quality of the software. In [79] authors make an attempt to help decide which metrics out of this wide range should be presented to the architect as the most important to look at. The information density property of software metric is proposed as a criterion for selecting candidates competing on these resources. Lajios et al. [80] investigated the correlation of various software metrics to the defect found in software modules and proposed an approach to determine a sets of metrics for quality assessment of complex software systems. First they calculated various quantitative, complexities, coupling and other metrics at the class level for several similar projects using different open source tools. Then they found the correlation of these metrics to the history of bugs using machine learning techniques. They found that although some of the metrics are more suitable for the assessment of software quality; these metrics differ between the analyzed projects even though their natures are similar. They also discovered that 5 out of 11 metrics were irrelevant for the analyzed systems. This research completes ours in the attempt to find which metrics are informative and which are irrelevant.

3.5.1 Metrics based on entropy

Entropy is used in various areas; in software engineering fields, it is applied to measure the cohesion and coupling of a modular system, to design a mathematical model for evaluating software quality, to define complexity measures, *etc*. [80, 81, 82]. Entropy-based metrics enable monitoring of a system's aging and they are also applied to evaluate software degradation [83, 84]. Aging and degradation of software are principal concepts in software maintenance; however, most studies using entropy have mainly focused on object-oriented systems [80, 84] or general modular software [83]. This issue has been investigates by Matinee Kiewkanya and Pornsiri Muenchaisri [85]. They present a new interpretation of the entropy metric. They argue that the uncertainty of occurrences of developer-defined tokens for class names, method names, parameters and variables in the source code is related to the quality of interfaces. This metric is superior to other metrics that assess the overall understandability of a software system in terms of metric properties. Although the original purpose of the metric is measuring the understandability and maintainability in order to estimate further maintenance efforts, this metric could also be used to measure the effectiveness of refractory. Zhou [67] and Kang et al. [85] proposed measuring the structural complexity of class diagrams based on an entropy distance. This method can measure the structural complexity of class diagrams objectively. In essence, the Zhou’s and Kang’s metrics are similar. The proposed metrics consider the number of relationships among classes, the interaction pattern of classes, and the kinds of relationship. Yi et al. [82] presented an empirical analysis of the entropy distance metric for class diagrams, specifically Zhou’s metric. This work explored a correlation between the entropy distance metric and the three sub-characteristics of maintainability: understandability, analyzability and modifiability measured from human rating. The experimental result indicated that the metric was basically consistent with human beings’ intuition.

Oleksandr et.al. [86] Proposes a novel interpretation of an entropy-based metric to assess the design of a software system in terms of interface quality and understandability. The proposed metric is independent of the system size and delivers one single value eliminating the unnecessary aggregation step. Although the use of entropy for measuring software artifacts is not new [87, 88], this research presents a new interpretation of the entropy metric. The authors argue that the uncertainty of occurrences of developer-defined tokens for class names, method names, parameters and variables in the source code is related to the quality of interfaces.

3.6 Summary of related studies

The following table summarizes list of previous work along with their strength and limitations.

Table 3.3: Summary of important related works

No	Investigators	Research /approach	Strength	Limitations
1	Törn et al. (1999)	Complexity metrics	Suggests a new measure of software complexity called structural complexity.	Programs have to be written in "node representation".
2	Sheng Yu, Shijie Zhou. **(2010)**		Comparing different complexity metrics	Comparative study doesn't consider studies for using semantic aspects for measuring complexity
3	De Silva, D.I.et.al. (2012)		Compare code complexity metrics: McCabe's, Halstead's and cognitive functional size proposed by Shao and Wangs' and identify which metric is the most suitable metric.	Doesn't consider semantic aspects in their comparison
4	Hartson, **H**. et.al. (1987)		Investigate the effect of using prediction model on reducing software complexity	Using companion of metrics may increase complexity
5	Nadine MESKENS 1996 IEEE	Integrated Metrics	Investigates how to Integrate semantic and syntactic aspect to reduce maintenance cost.	Concentrates on maintenance only
6	Zheng Jian-hua, and Wu Jia-pei, 2006	Building prediction models	introduced the Pseudo-path metric model (PPMM)	Computed range of the complexity is not reasonable and boor mathematical basis
7	(Mertik, M., et.al. (2006)		Construct fault Prediction model using data miner tool.	Using combination of method decrease performance
8	Tim Menzies (2007)		Construct defect predictor model by using static software	Only static software attributes are

			attributes.	considered
9	T.M. Khoshgoftaar and N. Seliya. (2003)		Evaluate different predictive performance of six commonly used fault prediction techniques. The results confirm that CART-LAD model is the best.	
10	Emanuel Giger, Martin Pinzger, Harald C. Gall		Propose prediction models for whether a source file will be affected by a certain type of source code change. The output a list of the potentially change-prone files ranked according to their change-proneness, overall and per change type category.	
11	Salwa K	Using data mining in prediction	Produce an efficient metrics-based data mining clone detection approach.	Applied on medium and large systems only
12	Ensan, F. and Du, W. *A*, 2013.		Propose metrics that measure cohesion and coupling of ontologies. Based on semantic information.	All metrics work in Employment of ontologies only
13	Yuehua Zhang, Ying Liu, Lingling 2010		Proposes a defect detection method using data mining techniques in source code	Doesn't detect implicit programming rules
14	Tao Xie ; Taneja, K.		They develop a novel techniques based on mining source code, assisting developers to improve software reliable.	Research focus on API library only
15	Gall, C. S 2008		Analyzing natural language (NL) design specifications for object-oriented systems by using semantic metric.	Applied for OO systems in design phase only.
	Salwa E.H 2004		Uses entropy to measure	Does not consider the

			information contents for (SW) based on function calls.(P=ni/N)	effect of this measure on quality attributes.
16	Letha H. tzkorn 2006	Semantic Metrics & frameworks	Discuss three types of conceptual and ontology based metrics.	Focused on IR systems
17	Bo Hu,et.al. 2006		Propose formal semantic metrics (web semantic).	Consider only ontology
18	Jeffrey M. Voas, Keith W. Miller 1993		Propose semantic metrics from specifications, to figure out functions that tend to hide faults.	Boor specification may lead to unreliable faults detectors.
19	**Larry J**. Morell, 1993		propose framework used to quantify semantic information	Include only information about program execution
20	Selim Kebir, Abdelhak-Djamel Seriai 2012		Propose an approach for identifying components based on a fitness function to measure the quality of a component.	The experiments done on limited versions of system.
21	B Neate Warwick Irwin Neville Churcher 2006		Implemented a tool, CODERANKER, to compute values of Code Rank metrics based on a full semantic model also suggested.	Focus on OO software
22	Steffen Zschaler **2004**		Define elements of a semantic framework for non-functional specifications of component-based systems.	Model doesn't able to identify unspecified non- functional requirements
23	Oleksandr Panchenko, Stephan H. Mueller, Alexander Zeier	Semantic entropy	Proposes a novel interpretation of an entropy-based metric to assess the design of a software system in terms of interface quality and understandability.	The resulting entropy value is lower than the entropy of The actual developer-defined tokens value.
24	D. Melamed		Study proposed a method for	He focus only on how

	1997		measuring semantic entropy using translational distributions of words in parallel text corpora.	to work with words, but not which words to work with
25	Abd-El-Hafiz, Salwa K 2004		Propose a model to measure information content by using entropy concept.	that syntactical rules of languages decrease entropy
26	Yossi Gil ! Maayan Goldstein Dany Moshkovich	Semantic metrics in OO systems	Paper describes a new criterion for evaluating the competing metrics based on a normalized version of Shannon's information theoretical	the decision of using one metric is very much application dependent
27	R. SE LVARANI1 , 2009		Study proposed the design complexity of object-oriented software with Weighted Methods per Class metric (WMC-CK metric) expressed in terms of Shannon entropy, and error proneness	Using entropy to measure only during design phase
28	Bilong Wen, Li Zhang 2010		Presents a method to map process metrics to the enterprise information model automatically when the semantic features of metric are analyzed. Two semantics tree is presented and a method to map metric to enterprise information model is put forward.	
29	Gabriele Bavota · Andrea De Lucia · Andrian Marcus · Rocco Oliveto. 2012		proposed a new technique for automatic re-modularization of packages, that uses structural and semantic measures to decompose a package into smaller and more cohesive ones	

Chapter Four

Research Methodology

4.1 Introduction

This chapter details out the research methodology used to the current study and how data collection, analysis and development of theory processed. It explains the research objectives and a suitable methodology to achieve those objectives. As mentioned previously in section 1.5 the main objective of the study was to use an evaluation approach depends on semantic features of software system as a tool to improve quality monitoring. One of the most important research questions addressed in section 1.3, whether semantic metrics can contribute to the field of software reliability measurement. It would be useful to know the probability of software failure, or the rate at which software errors will occur, and the relationship between semantic faults and failure rates as an indicator of software reliability.

The structure of this chapter is outlined in such a way that the first section 4.2 presents research strategies used in scientific researches. The subsequent section describes the generally accepted approaches to research and validation of the research followed by section 4.3 that describe how each step is carried out during research process.

4.2 Research Strategy

A large number of research methodologies have been identified, Galliers for example listing fourteen, while Alavi and Carlson , reported in Pervan, use a hierarchical taxonomy with three levels and eighteen categories [89]. Table 4.1 below, list the methodologies identified by Galliers [89]. Before introducing the methodologies used in this research, we summarize the key features of the key methodologies in the table, identifying their respective strengths and weaknesses. In the following sections, we justify our choice of methodologies and explain how they both operate and interoperate in our research.

Table 4.1 Taxonomy of Research Methodologies [89]

Laboratory Experiments	Subjective/Argumentative
Field Experiments	Reviews
Surveys	Action Research
Case Studies	Case Studies
Theorem Proof	Descriptive/Interpretive
Forecasting	Futures Research
Simulation	Role/Game Playing

Laboratory experiments permit the researcher to identify precise relationships between a small numbers of variables that are studied intensively via a designed laboratory situation using quantitative analytical techniques with a view to making generalizable statements applicable to real-life situations [89]. The key weakness of laboratory experiments is the "limited extent to which identified relationships exist in the real world due to oversimplification of the experimental situation and the isolation of such situations from most of the variables that are found in the real world" [89].

Field experiments extend laboratory experiments into real organizations and their real life situations, thereby achieving greater realism and diminishing the extent to which situations can be criticized as contrived. In practice it is difficult to identify organizations that are prepared to be experimented on and still more difficult to achieve sufficient control to make replication viable [98].

Surveys enable the researcher to obtain data about practices, situations or views at one point in time through questionnaires or interviews. Quantitative analytical techniques are then used to draw inferences from this data regarding existing relationships. The use of surveys permits a researcher to study more variables at one time than is typically possible in laboratory or field experiments, whilst data can be collected about real world environments. A key weakness is that it is very difficult to realize insights relating to the causes of or processes involved in the phenomena measured. There are, in addition, several sources of bias such as the possibly selfselecting nature of respondents, the point in time when the survey is conducted and in the researcher him/herself through the design of the survey itself.[89]

Case studies involve an attempt to describe relationships that exist in reality, very often in a single organization. Case studies can be considered weak as they are typically restricted to a single organization and it is difficult to generalize findings since it is hard to find similar cases with similar data that can be analyzed in a statistically meaningful way. Furthermore, different researchers may have different interpretations of the same data, thus adding research bias into the equation.

Simulation involves copying the behavior of a system. Simulation is used in situations where it would be difficult normally to solve problems analytically and Typically involves the introduction of random variables. As with experimental forms of research, it is difficult to make a simulation sufficiently realistic so that it resembles real world events [89].

Forecasting/futures research involves the use of techniques such as regression analysis and time series analysis to make predictions about likely future events. It is a useful form of research in that it attempts to cope with the rapid changes that are taking place in IT and predict the impacts of these changes on individuals, organizations or society. However, it is a method that is fraught with difficulties relating to the complexity of real world events, the arbitrary nature of future changes and the lack of knowledge about the future. Researchers cannot build true visions of the future, but only scenarios of possible futures and so impacts under these possible conditions [89].

Subjective/argumentative research requires the researcher to adopt a creative or speculative stance rather than act as an observer. It is a useful technique since new theories can be built, new ideas generated and subsequently tested. However, as an unstructured and subjective form of research, there is a strong chance of researcher bias.

Action research is a form of applied research where the researcher attempts to develop results or a solution that is of practical value to the people with whom the research is working, and at the same time developing theoretical knowledge. As with case studies, action research is usually restricted to a single organization making it difficult to generalize findings, while different researchers may interpret events differently. The personal ethics of the researcher are critical, since the opportunity for direct researcher intervention is always present [89].

4.3 Research Process and Methods

The research reported in the study was done in an iterative manner. To begin with, a review to the current state of art is done to capture knowledge about kinds, classification and uses of different software metrics. While, the major research in this area is focusing on syntactical aspects of software, few of them address semantic concepts. As discussed in chapter 3, measuring quality attributes by using only syntactic aspects has much limitation. Based on findings of literature review a new set of metrics are suggested to overcome the limitations of existing tools. Suggested metrics integrates semantic aspects of software to improve software reliability monitoring. Metrics are divided to cover three phases fault detection, error prevention and recover. Suggested metrics are modeled and evaluated theoretically and empirically against the research objectives. The evaluation process is iterative each time research objectives are re-examined to ensure that work is going in the right direction. Two methodologies are adopted in the research:

1. Empirical research, which attempts to highlight statistical relationships without attempting to justify them/ explain them.
2. Analytical research, which attempts to characterize software quality attributes from semantic metrics.

Referencing to figure 1.1 that illustrates detailed research process. The work can be divided into major phases as the following:

4.3.1 Define research goals

Research goals are identified previously, based on previous studies. The main goal is to monitor/control software reliability by using set of semantic metrics.

4.3.2 A literature study.

An extensive study was performed to survey what have been done in the area. Advantages \ limitations are pointed for each study and if it is will be considered or not in this study. Section 3.2 highlights this issue.

4.3.3 Data Collection

During this phase we must identify what quantifiable attributes can help to achieve the goals set forth in the previous phase. Study focused on computing quantitative functions that reflect a program's potential for fault tolerance; the suggested approach involves analyzing the program as well as its specification. The focus on fault tolerance comes from two findings derived from previous studies:

- Northrop et al. in [90] stated that, to control the quality of software specially large ones (Ultra Large Scale systems) it is better to consider Marco level analysis rather than minute statement-level detail;
- Patterson and Fox [91] argue the favor of controlling software quality through making error recovery, rather than straining to find and remove faults in software products.

Both these findings are considered in the present study.

4.3.4 Prepare / analyze data

Based on previous studies both reliability data and standard used programs in addition to, the most widely used metrics such as LOC, McCabe, Halstead, number of faults and fault density… etc. are collected to be used for both empirical and analytical research.

4.3.5 Design assumptions

In this stage assumptions are stated to be tested after work completion. Assumptions can be abbreviated as follow**:**

- Semantic metrics can be used to improve software reliability measurement.

- The statistical model could be used to predict probability of software failure based on semantic features.

Four metrics are suggested to cover the stages of fault tolerance: error detection, failure detection, error maskability and error recovery. Any of these measures is going through multiple development steps figure 4.1 shows simple explanation of these steps:

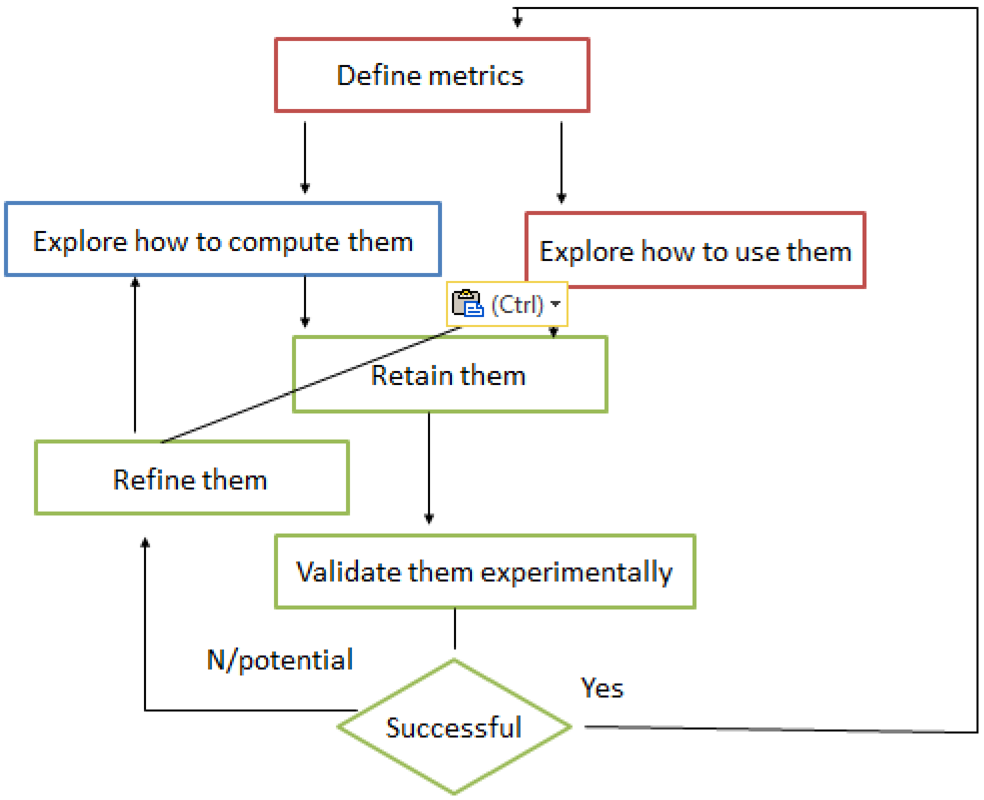


Figure 4.1: Steps for Defining Metrics

4.3.6 Proposed solution/ implementation/testing

According to defined goals four semantic metrics are suggested to measure program ability to be fault tolerance by detecting errors at run-time and avoid failure. The proposed solution are further tested and evaluated. Only one of the suggested metrics is implemented. The following are brief about these metrics [8]:

- A measure of state redundancy, which used to check state consistency.

- A measure of functional redundancy, abstract number, used to check program function correctness.
- A measure of maskability, consider program ability to mask error.
- A measure of recoverability indicates the bandwidth of loss that a program state can sustain while still satisfying the specification.

Metrics are discussed in more details in chapter 6.

4.3.7 Evaluation

In the evaluate phase, there is a need to evaluate the selected metrics to assess their fitness for the goals established in the first phase. Two methods are selected to evaluate the fitness of metrics: an analytical approach, which aims to compute or approximate quality attributes from semantic metrics; and an empirical approach, which collects statistical data regarding the link between semantic metrics and observations of quality in software systems. For empirical approach, the correlation between functional quality attributes (reliability, fault tolerance) and semantic metrics are estimated. Both regression and correlation techniques are used. For analytical approach, software failure life cycle are considered, Semantic metrics are integrated to measure factors affects failure. Statistical model are applied to measure probability of failure. Finally, multiple refinement process is done to insure achieving research goals. This issue will be discussed in (7.2 and 7.3).

4.4 Chapter summary

The chapter presents methodology followed during research steps including data collection and analysis then spot light on proposed solution and methods used to evaluate this solution.

Chapter Five

Software reliability mechanisms

5.1 Introduction

Nowadays a high number of software projects fail to follow their specified requirements regarding time, budget and specifications. Also their maintenance effort is higher than their implementation effort [4]. Thus, there is a great need for software metrics, in order to aid towards the overcoming of this “software crisis". Also, the results of the software metrics are not used efficiently enough to be able to direct those actions which will lead towards the improvement of the software's quality. This is because the metric's results are not fully analyzed and interpreted.

Software engineering relies on quantitative analysis to support decision making that pertains to the management of products and processes. To this effect, researchers have long been interested in defining and analyzing metrics results that capture properties of software products and software processes, to such an extent that software metrics have long since outgrown the laboratory stage and are now the subject of regular textbooks [3,10,14], and common industrial practice. One of the major software components that should be measured is reliability. IEEE 982.1-1988 defines Software Reliability Management as “The process of optimizing the reliability of software through a program that emphasizes software error prevention, fault detection and removal, and the use of measurements to maximize reliability in light of project constraints such as resources, schedule and performance”[10]. Three different techniques / mechanisms used to improve software reliability:

1. Error prevention.
2. Fault detection and removal.
3. Fault tolerance.
4. Fault/failure forecasting

The following sections explain these techniques used to improve reliability starting by distinction between fault, error and failure concepts.

5.2 Faults / Error/Failure concepts:

The terms errors, faults and failures are often used interchangeable, but they have different meanings. In software, fault is usually a programmer action or omission that may results in an error.[3]

An error is a software defect that causes a failure, and a failure is the unacceptable departure of a program operation from program requirements. When measuring reliability, only defects found and defects fixed are usually measured [92]. If the objective is to fully measure reliability the focus will be on prevention as well as fault tolerance. It is important to recognize that there is a difference between hardware failure rate and software failure rate. Software however, has a different fault or error identification rate. For software, the error rate is at the highest level at integration and test. As it is tested, errors are identified and removed. This removal continues at a slower rate during its operational use; the number of errors continually decreasing, assuming no new errors are introduced. Software does not have moving parts and does not physically wear out as hardware, but it becomes unable to achieve the renewable requirements [92].

5.3 Software Reliability mechanisms

5.3.1 Fault prevention:

This mechanism is the initial defensive mechanism against unreliability. A fault which is never created costs nothing to fix. Fault prevention is therefore the inherent objective of every software engineering methodology. General approaches include formal methods in requirement specifications and program verifications; early user interaction and refinement of the requirements, disciplined and tool-assisted software design methods, enforced programming principles and environments, and systematic techniques for software reuse [39].

5.3.2 **Fault removal:**

Used to detect, by verification and validation, the existence of faults and eliminate them. Fault prevention mechanisms cannot guarantee avoidance of all software faults. When faults are injected into the software, fault removal is the next protective means. Two practical approaches for fault removal are software testing and software inspection, both of which have become standard industry practices in quality assurance. Directions in software testing techniques are addressed in [47] in detail.

When inherent faults remain undetected through the testing and inspection processes, they will stay with the software when it is released into the field [39].

5.3.3 **Fault tolerance**:

It is the last defending line in preventing faults from manifesting themselves as system failures. Fault tolerance is the survival attribute of software systems in terms of their ability to deliver continuous service to the customers. Software fault tolerance techniques enable software systems to:

(1) Prevent dormant software faults from becoming active.

(2) Recover software operations from erroneous conditions.

5.3.4 **Fault/failure forecasting:** to estimate, by evaluation, the presence of faults and the occurrences and consequences of failures. This has been the main focus of software reliability modeling. It involves formulation of the fault/failure relationship, an understanding of the operational environment, the establishment of software reliability models, developing procedures and mechanisms for software reliability measurement, and analyzing and evaluating the measurement results [39].

The ability to determine software reliability not only gives us guidance about software quality and when to stop testing, but also provides information for software maintenance needs.

5.4 Information theory and entropy

5.4.1 Information theory

Information theory is a branch of applied mathematics, electrical engineering, and computer science involving the quantification of information. Information theory was developed by Claude E. Shannon to find fundamental limits on signal processing operations such as compressing data and on reliably storing and communicating data. Since its inception it has broadened to find applications in many other areas, including statistical inference, natural language processing, cryptography, neurobiology, thermal physics, plagiarism detection and other forms of data analysis [93].

A key measure of information is entropy, which is usually expressed by the average number of bits needed to store or communicate one symbol in a message. Entropy quantifies the uncertainty involved in predicting the value of a random variable. The following subsections define these concepts [93].

5.4.2 Relational mathematics

The main source of this section is [94], to which the interested reader can referred, for further details. Consider a set S defined by the values of some program variables, say x, y and z; typically denote elements of S by s, and note that s has the form s = <x, y, z>. The following notation x(s), y(s), z(s) are used to denote the x-component, y-component and z-component of s, respectively. A relation on S is a subset of the Cartesian product S × S. Constant relations on some set S include the universal relation, denoted by L (=S×S), the identity relation, denoted by I, and the empty relation, denoted byΘ.

Because relations are sets, the usual set theoretic operations can be applied between relations such as: union (∪), intersection (∩), and complement ($\overline{R}$). Operations on relations also include the converse, denoted by $\hat{R}$, and defined by $\hat{R} = \{(s, s')|(s', s) \in R\}$. The product of relations R and R′ is the relation denoted by R ∘ R′ (or RR′) and defined by $R \circ R' = \{(s, s')|\exists\, s'' : (s, s'') \in R \wedge (s'', s') \in R'\}$. The domain of relation R is defined as $dom(R) = \{s|s': (s, s') \in R\}$. The range of relation R is denoted by rng(R) and defined as dom($\hat{R}$). We admit without proof that for a relation R, RL = {(s, s′)|s ϵ dom(R)} and LR = {(s, s′)|s′ ϵ rng(R)}. The nucleus of relation R is the relation denoted by μ(R) and defined as R$\hat{R}$. The *co-nucleus* of relation R is the relation denoted by (R) and defined as $\hat{R}$R.

We say that relation R is *total* if and only if μ(R) = L and we say that relation R is *surjective* if and only if co-*nucleus* (R) = L. Given two relations R and R′ that have the same domain, we say that R is *more-injective* than R′ if and only if μ(R) ⊆ μ(R′) and we say that R is *injective* if and only if it is more-injective than I; the name *more-injective* may be misleading, given that we are talking about a reflexive ordering (it should be more-injective-than-or-as-injective-as), but we adopt it for convenience. Given two relations R and R′ that have the same range; we say that relation R is *deterministic* if and only if it is more-deterministic than I.

5.4.3 Entropy

Entropy is a measure of unpredictability of information content. In this context, the term usually refers to the Shannon entropy, which quantifies the expected value of the information contained in a message.[93] The following is the main equation of entropy measured for variable X:

$$H(X) = -\sum_{i=1}^{n} P(xi)\log(P(xi))$$

Where:

- log is the base 2 logarithm,
- $X = \{x_1, x_2, x_3, \ldots x_n\}$,
- $P\ (x_i)$ is the probability of the event: $X = x_i$.

Entropy is typically measured in bits [93]. Shannon entropy is the average unpredictability in a random variable, which is equivalent to its information content. Shannon entropy provides an absolute limit on the best possible lossless encoding or compression of any communication, assuming that the communication may be represented as a sequence of independent and identically distributed random variables.

Entropy, especially the Shannon entropy, is used in various and diverse software engineering applications. For example, the entropy concept is used in designing mathematical models for software quality evaluation [68] and in providing mechanisms for selecting optimum reuse candidates.

Recently, many researchers move toward using entropy in software quality measurement as discussed previously (see section 3.5.1).

The following are some possible interpretations of entropy in software [71]:

- Entropy of a probability distribution is the expected value of the information of the distribution.
- Entropy is related to how difficult it is to guess the value of a random variable X.
- Entropy indicates the best possible compression for the distribution, i.e. the average number of bits needed to store the value of the random variable X.

According to above equation, study admit without proof that $H(X) \geq 0$;and the expression $p \log(p)$ equals zero when p equals 0, hence the entropy function may be applied to probability distributions that are not necessarily non-zero for all xi [8].

Intuitively, the entropy of random variable X represents the amount of uncertainty regarding the outcome of the random variable, and takes its maximal value (which is $\log(n)$) when all the outcomes are equally likely ($\pi(xi) = 1\ n$ for all i).

Given two random variables X and Y on sets X and Y , and let πX and πY be probability distributions of X and Y over their respective sets; let πXY be the probability distribution of the events $(X = xi \wedge Y = y\ j)$ over the Cartesian product $X \times Y$ *[8]*.

The joint entropy of X and Y denoted by $H(X, Y)$*and represents* the entropy of the aggregate random variable (X, Y) over the set $(X \times Y)$. Using this

Definition, let the conditional entropy of X with respect to Y be denoted by $H(X|Y)$ and be defined as follows:

$$H(X|Y) = H(X, Y) - H(Y).$$

Whereas the entropy of X represents the amounts of uncertainty about the outcome of X, the conditional entropy of X with respect to Y represents the amount of uncertainty about the outcome of X once the outcome of Y *is known*. The conditional entropy is non-negative because the joint entropy of (X, Y) is greater than or equal to the entropy of Y.

Given a random variable *X* that takes its values in some space *S*, and given a function *G* on *X*, let *Y* be the random variable *Y* = *G(X)*,whose probability distribution is derived from that of *X*, i.e.,

$$\pi Y\ (Y = y) = \_\ \forall x{:}G(x)=y\pi X\ (X = x).$$

Then, the result is the inequality [93]: $H(X) \geq H(Y)$. In other words, applying a function to a random variable reduces its entropy (due to possible loss of information). If *G* is total and injective, then $H(G(X)) = H(X)$.
To conclude this section, following section introduces a concept used throughout this study to assign intuitive interpretations to semantic metrics.

Definition 1[8]:

Consider a set S and a predicate (A) on S, and let SA be the subset of S defined by elements of S that satisfy A(s). The bandwidth of assertion A is defined as:

$$H(S) - H(SA).$$

E.g. consider a set S defined by three integer variables, say x, y and z. Under the hypothesis of uniform probability distribution, and assuming that integers are represented by 32-bit words, the entropy of S is 96 bits. By considering the following predicate:

$$A(s) \text{ as } x = y.$$

H (SA) = 64 bits, then the bandwidth of Assertion is 32 bits, which is the width of the two expressions (x and y) involved in assertion A. entropy are used in the study to define semantic metrics that contribute to software reliability measurement.

5.4.4 Measuring information content:

Measuring software information content for the information measures to be as independent as possible of any product abstractions, by following the general definitions of software systems and modules introduced in [71]. A software system S = <E, R> is defined as a set of elements, E, and a binary relation, R, on them. Given a system S = <E, R>, a system m = <Em, Rm> is a module of S if and only if Em is a subset of E and Rm is a subset of R. For example, E can be the set of functions and R can be the set of calls from one function to another. A module m may be a group of functions. To measure the information content of software systems two different issues must be addressed.

The first issue is about which entropy is suitable for measuring software information. The second issue is about which parts of the source code should be treated as the symbols emitted from the information source (the software system).

The relation between entropy and information gain could be abbreviated in "the more Shannon entropy, the more information gained after learning the outcome of probabilistic event"[71].

On the other hand, the information content of a system consisting of two modules is not greater than the sum of the information expected from the individual experiments.

That is, $H(S) \leq H(m1) + H(m2)$.

M1 and m2 represent modules names. The information gained from a system can be less than the summation of the information gained from its two constituent modules because of several reasons such as repeated calls to the same functions or repeated usage of the same abstract data types. As long as two modules are in the same system, they are, somehow, dependent.

5.5 Chapter Summary

The chapter introduced major concepts related to reliability mechanisms such as fault prevention, fault detection and fault tolerance. Recently, much research considers the use of information theory to improve reliability. This issue is investigated starting by defining the main concepts of information theory – entropy and set theory. Finally, it introduced how to measure information gain using these concepts.

Chapter Six

Semantic Metrics

6.1 Introduction

Most of the software metrics that are being used nowadays (and certainly the most widely known) are based on syntactic attributes of software artifacts; they reflect how a program is represented, but not what a program does; yet, many important program attributes may have more to do with the latter than the former[8]. In addition, many software attributes of interest are not intrinsic to the software product and also involve the specification that the software product is supposed to satisfy; hence if we want metrics to reflect relevant quality attributes, we need to pay attention not only to the software product, but also to its specification.

All of the research that has been done on the correlation between software metrics on one hand and fault density, fault proneness, and fault forecasting on the other hand, do not consider given specifications; yet a fault is a fault only with respect to a specification. In order to be more comprehensive, software metrics ought to take into account attributes of specifications along with attributes of programs [10].

In the study, a number of software metrics that reflect semantic properties of software products are introduced, which is independent of the minute details of how products are represented.

As mentioned in section (4.2) the work will proceed according to multiple phases such as:

- The Establishment phase, in which goals should be defined.
- The Extraction phase, determines the attributes used to achieve the goals.
- The Evaluation phase, evaluate the selected metrics to assess their fitness for the goals established in the first phase. study envision two venues to evaluate the fitness of proposed metrics: an Analytical approach, which aims to compute or approximate quality attributes from semantic metrics; and an empirical approach, which collects statistical data regarding the link between our semantic metrics and observations of quality in software systems.
- The Execution phase, deploys the selected metrics, once they are validated.

The software metrics that will present in the coming sections are semantic in the following sense: they view software products as aggregates of spaces, functions and relations; furthermore they reflect the set theoretic properties of theses spaces, functions and relations. The discussion is conducted in the context of C-like procedural programs, but can be extended to include other types of programs.

6.2 Fault tolerance Methodology

The main source for this section is [8]. Consider a program g on some space S, of the form

g = {g1; L: g2;}

Where g1 and g2 are subprograms and L is a label preceding g2. We let R be a relation on S that represents the specification that g must meet, and we let s0 be an arbitrary initial state of g.

- A fault in program g is a feature of g that precludes it from satisfying its specification.
- An error of the program at label L for initial state s0 is a state that is distinct from the expected state at this label;
- A fault may or may not cause a fault at label L, depending on the initial state s0; when a fault does cause an error, we say that it has been sensitized by the initial state s0.
- A failure of program g occurs whenever the error that arises at label L causes the program to fail to produce a correct (with respect to R) final state for initial state s0. An error at label L may cause a failure of the program, in which case we say that the error has been propagated; it may also cause no failure, in this case the error said it has been masked.

Program g considered as fault tolerant if and only if it has provisions for avoiding failure after faults have caused errors. Study considers three phases in the fault tolerance process:

- Error Detection, when the program detects an inconsistency that indicates that the program state is erroneous.
- Damage Assessment, when the program analyzes the current state to determine whether it is maskable or recoverable (in which case recovery is necessary and sufficient) or unrecoverable (in which case recovery is insufficient).

- Error Recovery, when a recovery is invoked to map the recoverable state into a maskable state and let the computation resume from label L.

As an illustration, consider the space S defined by a natural variable, let the specification be relation R defined by

$$R = \{(s, s') | \; s' mod\; 3 = s^2 \; mod\; 3\}$$

Let g be the program g = {read(s); s=2*s; L: s = s mod 6; write(s);}
The intent of the programmer was for g to compute the following function:

$$G = \{(s, s') | \; s' = s^2 \; mod\; 6\}$$

Which would have been correct with respect to R (in the sense of [15]), since G and R are both total, and G ⊆ R, as shown below:

$$s' = s^2 \; mod\; 6 \;\; \Rightarrow s' mod\; 3 = (s^2 \; mod\; 6) mod\; 3 = \; s^2 \; mod\; 3 \; .$$

But the programmer wrote the statement s = 2*s instead of the statement s=s*s, creating a fault. This fault may or may not be sensitized, depending on the input value:

- For s_0 = 2, the fault is not sensitized, since the expressions $s * 2$ and $s * s$ return the same value for s = 2.
- For s_0 = 6, the fault is sensitized, causing an error (s = 12 rather than s = 36 at label L), but the error is subsequently masked (since 12 mod 6 = 36 mod 6 at the end of the program).
- For s_0 = 3, the fault is sensitized, leading to an error (s = 6 instead of s = 9 at label L); the error is subsequently propagated, causing a failure (s = 0 instead of s = 3 in the final state); in this instance, program g failed to behave according to its intended function G, but did not fail with respect to its specification R, since

  $$s_0 mod\; 3 = 9 \; mod\; 3 = 0 = 0 \; mod\; 3.$$

  Hence, strictly speaking, it satisfies its specification for s_0 = 3.
- Finally, for s_0 = 4, the fault is sensitized, leading to an **error** (the state at label L is $s * 2 = 8$ rather than $s * s = 16$); this **error is propagated**, leading to a final state that is distinct from the expected final state (the output is s = 2 rather than s = 4); this final state violates the specification, since 2

mod 3 6= 4 mod 3; in this case, the program failed to compute the expected final state, and also failed to satisfy the specification of the program.

The same fault may cause different chains of events, depending on the input. In order to be fault tolerant, a program must make provisions for error detection (to recognize when the potential of a failure may arise), error masking (to limit cases when recovery is necessary), and error recovery (to map a recoverable state into a maskable state, and let the computation proceed). The following sections describe these issues.

6.3 Error detection: redundancy

Broadly speaking, redundancy is the property of using more data than is needed to represent some information. Whereas redundancy is usually defined in terms of duplicating elements of data (bits, words, etc), we model it instead as an algebraic property of the representation function, i.e., the function that maps information onto data. We distinguish between two types of redundancy in a program: state redundancy and functional redundancy.

6.3.1 State redundancy

State redundancy can be seen as defining extra variable size than required. Given a program g on space S, it is fair to say that in general, not all elements of S represent valid program states. E.g. defining variable to indicate student age as an integer value, even though, only a limited range will be used. The simplest representation relations are those that are [8]:

- Total (each state value has at least one representation),
- Deterministic (each state value has at most one representation).
- injective (different states have different representations),
- Surjective (all representations represent valid states).

Not all representation functions satisfy these four properties. Study limited to representation relations that are deterministic, total, and injective—whence each state value has exactly one representation (by virtue of totality and determinacy) and different state values have different representations (by virtue of injectivity). Under this assumption, representation relations refer to as representation functions.

Study is interested to quantify the redundancy of the state of a program. To this effect, there is a need to distinguish between the actual state space of the program, which defined as the set of states that the program may be in, and the declared state space of the program, which is the set of values that the declared program variables may take. We let ρ be the function that maps each actual state onto its representation as an aggregate of values of the declared variables. The state redundancy of program is defined by means of the representation function, as follows.

Definition 1 Let g is a program, and let Σ be the set of actual states of g, and S be the set of declared states of g.

- If we let ρ be the representation function that to each actual state σ assigns its representation in S, then we define the redundancy of ρ as:

$$\kappa(\sigma) = H(S) - H(\rho(\sigma)).$$

If ρ is total, deterministic and injective, then H(ρ(σ)) is equal to H(σ); hence, when the representation function is total and injective, its redundancy can be written as:

$$\kappa(\rho) = H(S) - H(\sigma).$$

Typically, the set of declared states is fixed for a given program block (which is the scope of typical variable declarations), but the set of actual states varies as the program proceeds through its execution; hence the redundancy of a state representation may vary from one step to the next through the execution of a program.

The state redundancy of the initial state reflects the gap between the minimal bandwidth required to store the program state and the actual bandwidth reserved to that effect. The state redundancy of the final state reflects the maximum bandwidth of relationships that hold between program variables as a result of the execution of the program.

As an illustration of this definition, consider a simple program that reads two integers included between 1 and 1,024 and computes their greatest common divisor.

```
{   int x, y; cin << x << y;
    // initial state
    while (x!=y) {if (x>y) {x=x-y;} else
    {y=y-x;}}
    // final state        }
```

Example1: Computes greatest common divisor

The declared state space of the program includes two integer variables, which we assume to be of width 32 bits; then H(S) = 2 × 32 bits = 64 bits.

As for σI, it consists of two integer values ranging between 1 and 1,024;

H(σI) = 2 × log(1,024) bits = 20 bits.

We derive the state redundancy of the initial state as:

κ(σI) = 44 bits.

For the final state, the declared state space is the same, but the actual range of states is now reduced to a single value between 1 and 1,024, since variables *x* and *y* are identical. Then *κ(σF)* = 64 bits − 10 bits = 54 bits. The state redundancy of this program is ranging between *κ(g)* = [44 bits..54 bits].

6.3.2 Functional redundancy

Whereas state redundancy reflects the excess data in the representation of a state, and can be used to check consistency conditions within the variables of a state, functional redundancy reflects the excess output data generated by a program function, and can be used to check (partially or totally) whether the function has executed properly. Whereas the redundancy of a state is equated with the non-surjectivity of the representation function (mapping actual states to their representation), the functional redundancy of a program is equated with the non-surjectivity of the program function (mapping initial states to final states, or inputs to outputs).

Definition 2:

Consider a program *g* on space *S*, and let *G* be the function defined by *g* on *S*. Let *S* be a random variable that takes its values in set *S*, and *Y* be a random variable that takes its values in the range of *G*. The functional redundancy of program *g* is denoted by *φ(g)* and defined by:

$$\varphi(g) = (H(S) - H(Y))/ H(Y).$$

Intuitive interpretation the functional redundancy of a program *g* is the ratio of the excess information that represents the output of *g* prorated to the entropy of the output produced by *g*. The functional redundancy of a program *g* may be used to check (partially or totally) the correctness of the output produced by the program, or even to generate the correct output. So, if $\varphi(g) = 0$, there is no scope for checking any property;

If $0 < \varphi(g) < 1$

Then part of results could be checked against redundant information; if $\varphi(g) > 0$, then $H(G(S)) \times \varphi(g)$ represents the bandwidth of assertions that may be checked on the functional properties of *G*.

For example, if program *g* computes the values of five integers, and $\varphi(g) = 0.2$, then there may be sufficient redundancy to check that one of the five values is computed correctly. Knowing the value of $\varphi(g)$ does not tell us how to use the redundant information; but if we identify and use it, it can tell us whether we are using all the available redundant information. e.g. considers the following functions. A denotes 5 bit variable, so functional redundancy will be:

Table 6.1: Measured functional redundancy

Name	Expression	Input	Output	Redundancy	Comment
F1	X	A	A	0	All bits are used
F2	X * 2	A	A	0.25	Rightmost bit contains 0

6.4 Error masking: program non-injectivity

Whereas state and functional redundancy enable to detect errors, maskability enables to mask them, i.e., produce a subsequent state that bears no trace of the error. What makes this possible in practice is the non-injectivity of programs, i.e., their ability to map distinct states into a single image. The following definition offers a way to quantify the non injectivity of program functions.

Definition 5 Let *g* is a program on space *S*, whose function is *G*. Let *X* be a random variable that takes its values in the domain of *G* and let *Y* be defined as $Y = G(X)$. The non-injectivity of program *g* is denoted by $\theta(g)$ and defined by:

$$\theta(g) = H(X|Y).$$

The conditional entropies are non-negative, hence $\theta(g) \geq 0$. To justify this definition, study proceeds in two steps: first, assume a uniform probability distribution over variable *X*; then the entropy of *X* given *Y* measures the amount of uncertainty of the initial state of *g* if the final state is known; this quantity is a natural representation of non-injectivity, in the sense that the more initial states map to the same image, the bigger the entropy. Second, consider the question: why does a non-uniform probability distribution represent smaller non-injectivity? The answer is that with a non-uniform probability distribution, fewer possible input values have a higher probability of occurrence, culminating in a smaller set of inputs mapping to a single output, hence a less injective behavior.

Intuitive interpretation the non-injectivity of program *g* is expressed in Shannon bits and represents the bandwidth of error that the program can potentially mask. For example, if the program handles integer variables of width *w* each, and the non-injectivity of *g* is *w* bits, the program may potentially mask the loss of an integer variable; for the same amount of injectivity, the program may also recover from the violation of an assertion whose bandwidth is *w* (e.g., an equality between two integer expressions); if the non-injectivity is 2*w*, the program can potentially mask the loss of two integer variables, etc. Knowing the value of the program's non-injectivity does not tell us what variables may not be lost, nor which assertion may be violated, but gives us some indication of the magnitude of error that can be masked without outside intervention.

Proposition 1 *Let g be a program on space S, whose function is G. Let X be a random variable that takes its values in the domain of G and let Y be defined as Y = G(X). The non-injectivity of program g can be written as:*

$$\theta(g) = H(X) - H(Y).$$

Proof According to [93], $H(X|Y) = H(X, Y) - H(Y)$,where $H(X, Y)$ is the joint entropy of X and Y . Given that study consider deterministic programs, Y is a function of X, hence

$H(X, Y) = H(X)$. Then

$$\theta(g) = H(X) - H(Y).$$

In practice, there is a need to derive rules allows to compute the non-injectivity of a program by analyzing its source code. Figure 6.1, shows the meaning of non-injectivity metric.

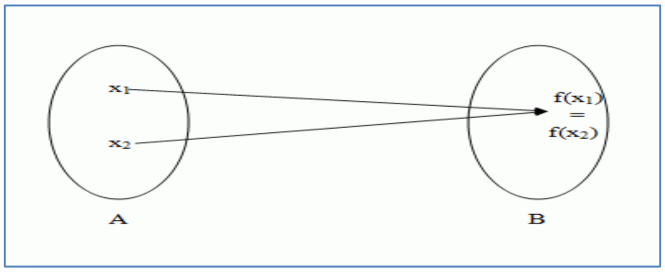


Figure 6.1: A diagram showing a function that is not injective [95]

Example: if we consider example (1) in section (6.3.1). The non-injectivity equals:

$$\theta(g) = H(X) - H(Y).$$

$H(X) = 2w$, assuming w is 32bit. $H(Y) = 2w$, then $\theta(g) = 0$.

Proposition 2 *The non-injectivity of a sequence of programs is the sum of their non-injectivities:*

$$\theta(g1; g2) = \theta(g1) + \theta(g2).$$

Proof: let X, Y, and Z be the random variables representing the state of the program before $g1$, between $g1$ and $g2$, and after $g2$. We have:

$\theta(g1) = H(X) - H(Y)$, and $\theta(g2) = H(Y) - H(Z)$, hence $\theta(g1) + \theta(g2) = H(X) - H(Y) + H(Y) - H(Z)$, which simplifies to $(H(X) - H(Z))$, which is $\theta(g1; g2)$

Program Non-injectivity can be computed by knowing program function without have to go through the inductive statement-by-statement analysis.

6.5 Error recovery: Specification flexibility

A program may fail to compute its intended function and yet still behave according to the specification it is intended to satisfy [8].

Definition 6: We consider a specification R under the form of a binary relation on some space *S*, and we let *X* be a random variable that takes its values in the domain of *R* and *Y* be a random variable that takes its values in the range of *R* in such a way as to maintain the condition $(X,Y) \in R$. The non-determinacy of specification *R* is denoted by $\chi(R)$ and defined by:

$$\chi(R) = H(Y|X).$$

A specification is all the more non-deterministic (flexible) that the conditional entropy of its output states for a given input state is greater; bigger entropies are equated with larger sets of possible outputs, and more uniform probability distribution of the occurrence of these outputs.

Intuitive interpretation the non-determinacy of a specification is expressed in Shannon bits and represents the bandwidth of deviation of candidate programs from their intended function that does not violate the specification, figure 6.2, shows the meaning of non-determinacy. For example, if state *S* includes integer variables of width *w* and we find that the non-determinacy of *R* is *w*, then we can lose up to one integer variable and still satisfy the specification. Non-determinacy of this specification can be computed by using the formula:

$$\chi(R) = H(X, Y) - H(X).$$

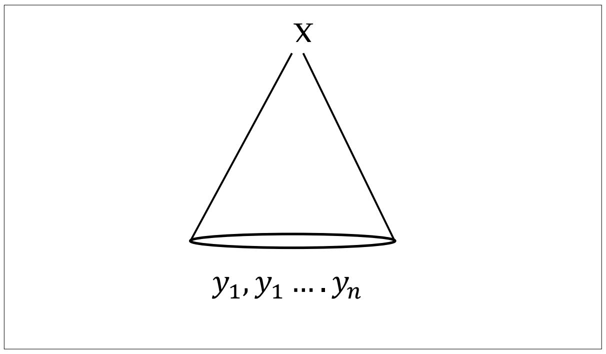


Figure 6.2: Meaning of non-determinacy.

Illustration: Consider the following specification defined on space S of natural variables, and is defined by:

$$R = \{(s, s') | s \bmod 3 = s^2 \bmod 3\} \quad [8]$$

Let X and Y are random variables that range over S in such a way as to maintain the property:

$$y \bmod 3 = x^2 \bmod 3.$$

The non-determinacy of relation R using the expression:

$$\chi(R) = H(X, Y) - H(X),$$

Using the uniform probability distribution of X and Y. We find, $H(X, Y) = 2w - \log(3)$, and $H(X) = w$. Hence,

$$\chi(R) = w - \log(3)$$

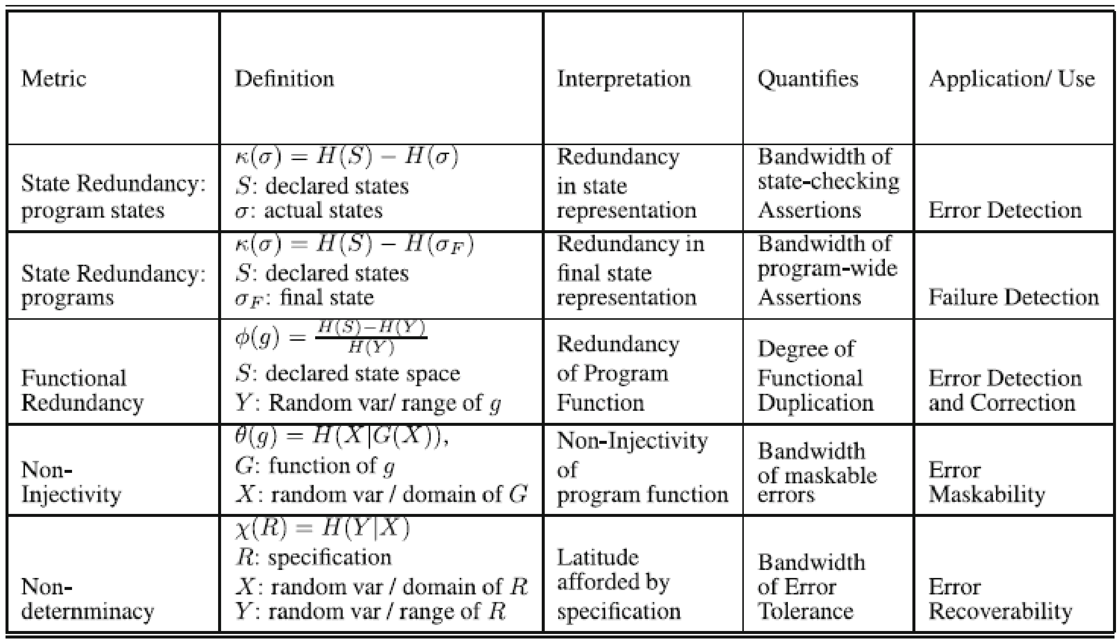
6.6 Summary of Semantic Metrics

In keeping with the foregoing premises, we have derived four semantic metrics, which measure a program's ability to detect errors at run-time and avoid failure.

1. A measure of state redundancy, which quantifies the non-surjectivity of state representations, is expressed in Shannon bits, and indicates the bandwidth of assertions that can be checked to ensure state consistency.
2. A measure of functional redundancy, which quantifies the non-surjectivity of program functions, is expressed as an abstract number, and indicates the ratio of the program function that can be checked for correctness.
3. A measure of maskability, which quantifies the non-injectivity of program functions, is expressed in Shannon bits, and indicates the bandwidth of error that may arise in the program state and still be masked by the program.
4. A measure of recoverability, which quantifies the non-determinacy of program specifications, is expressed in Shannon bits, and indicates the bandwidth of loss that a program state can sustain while still satisfying the specification.

Together, these four metrics ought to give the analyst some indication regarding the program's ability to tolerate faults and avoid failure. Table 6.2 Summarize semantic metrics, definition, and interpretation along their use.

Table 6-2: Metrics Definition and interpretations



Metric	Definition	Interpretation	Quantifies	Application/ Use
State Redundancy: program states	$\kappa(\sigma) = H(S) - H(\sigma)$ S: declared states σ: actual states	Redundancy in state representation	Bandwidth of state-checking Assertions	Error Detection
State Redundancy: programs	$\kappa(\sigma) = H(S) - H(\sigma_F)$ S: declared states σ_F: final state	Redundancy in final state representation	Bandwidth of program-wide Assertions	Failure Detection
Functional Redundancy	$\phi(g) = \frac{H(S)-H(Y)}{H(Y)}$ S: declared state space Y: Random var/ range of g	Redundancy of Program Function	Degree of Functional Duplication	Error Detection and Correction
Non-Injectivity	$\theta(g) = H(X\|G(X))$, G: function of g X: random var / domain of G	Non-Injectivity of program function	Bandwidth of maskable errors	Error Maskability
Non-deternminacy	$\chi(R) = H(Y\|X)$ R: specification X: random var / domain of R Y: random var / range of R	Latitude afforded by specification	Bandwidth of Error Tolerance	Error Recoverability

6.7 Chapter Summary

The chapter presents four semantic metrics based on entropy concept. The four metrics can contribute to provide a measure of the main factors of fault tolerance.

Chapter 7
Validation

7.1 Introduction

This chapter describes validation process by using both empirical and analytical validations. Empirical validation tends to explore correlations between different measures used. Analytical validation employs statistical concepts to predict new features based on probabilistic measures.

The structure of this chapter is outlined in such a way that the first section, 7.2, presents empirical research done along with its results and the subsequent section describes the proposed analytical model and its results.

7.2 Empirical research

Empirical validation is used to explore correlations between some functional quality attributes such as reliability, fault tolerance and semantic metrics. During experiments, the most famous standard programs – Siemens and Space [96]- are selected to test the proposed metrics. The "Siemens" programs were assembled by Tom Ostrand and colleagues at Siemens Corporate Research for a study of the fault detection capabilities of control-flow and data-flow coverage criteria [96]. The space program consists of 9564 lines of C code (6218 executable) and functions as an interpreter for an array definition language (ADL). The program reads a file that contains several ADL statements, and checks the contents of the file for adherence to the ADL grammar and to specific consistency rules. If the ADL file is correct, space outputs an array data file containing a list of array elements, positions, and excitations; otherwise the program outputs error messages [96]. Our four metrics are applied on 10 programs including 7 programs from Siemens collection in addition to, three other programs. Such as: tacs, schedule, schedule2, replace, totinfo, printtokens, printtokens2, Gzip, Sorting and Space, then results are recorded for validation purpose. Table 7.1 shows a brief description of the used programs.

Table 7.1: Description about used programs

	Program Name	Description
1	Tcas	Altitude separation - aircraft collision avoidance system.
2	Schedule2	Are priority schedulers
3	Schedule	
4	Replace	performs pattern matching and substitutions (pattern recognition)
5	Space	interpreter for an array definition language (ADL)
6	Sorting program	Algorithm receive unordered array and perform multiple substations to order array.
7	Printtokins	lexical analyzers
8	Printtokins2	
9	To_info	Information gain measure
10	Gzip	Unix utility

7.2.1 Applying metrics:

In this step 8 metrics are applied on selected programs. Metrics are: McCabe, Halstead, Number of fault, Fault Density in addition to, the proposed four semantic metrics. Tables 7.2 and 7.3, show the result of applying 8 metrics on selected programs. Both correlation and regression analysis methods are used to explore relation between syntactic and semantic metrics. The regression analysis method has been used to identify the closest relation between the above mentioned metrics [97]. It implements a linear regression model. Which means that the dependent variable(s) can be written in terms of linear combinations of the independent variable(s) [19]. The following section shows the results for the empirical validation step.

Table 7.2 shows the results of applying four syntactic metrics on selected programs and table 7.3 shows applying the four semantic metrics on selected programs.

TABLE ‘7.2’:

RESULTS OF APPLYING SYNTACTIC METRICS ON SELECTED PROGRAMS

P. name	V(G)	V	Number of faults	Fault density
Tcas	26	3800	41	0.01
Schedule2	49	7715	10	0.001
Schedule	37	7785	9	0.001
Replace	92	17293	32	0.001
Space	748	33015	35	0.02
Sorting	6	646	0	0.02
Totinfo	45	9311	23	0.01
Printtokins	72	12922	10	0.01
Printtokins2	79	9973	7	0.01
Gzip	1260	24149	40	0.0

Where P.name: program name, V (G): Complexity and V represents volume.

Fault density are measured by using the simple equation,

Fault density = number of faults/ size.

TABLE ‘7.3’:

SEMANTIC METRICS APPLIED ON 10 SELECTED PROGRAMS.

P. name	Functional redundancy	State redundancy	Non- injectivity	Non determinacy
Tcas	0.03	713.4 bits	34 bit	32
Schedule2	0.02	801.9 bits	64 bits	0
Schedule	0.02	124.7 bits	96 bits	0
Replace	0.03	601.6 bits	32 bit	32
Space	2.4	63996 bits	19200 bits	32
Sorting	14.6	2435 bit	564 bits	564
Totinfo	0.03	277.9 bits	224 bit	32
Printtokins	0.05	260 bits	318 bits	32

Printtokins2	24.6	480 bits	200 bits	32
Gzip	0.07	30150 bit	300 bit	32

7.2.2 Correlation Analysis:

The goal here is to verify that there is a statistically significant association between attributes estimated by semantic and syntactic metrics. Spearman rank correlation is a commonly-used robust correlation technique [97] because it can be applied even when the association between elements is non-linear. Table 7.4 Shows that there exists a statistically significant positive relationship between the following:

- Functional redundancy and State redundancy.
- Non- determinacy and functional redundancy.
- Non injectivity and fault density / complexity.
- Complexity and volume
- Complexity and fault density
- Volume and fault density.

TABLE 7.4: CORELATION RESULTS

		FR	NJ	V(G)	V	SR	NFaults	FDensity
FR	CC	1.000	.147	.087	.026	.578*	-.127-	-.258-
	Sig	.	.600	.757	.928	.024	.653	.353
NJ	CC	.147	1.000	.649**	.496	.054 -	.033	-.630*
	Sig.	.600	.	.009	.060	.849	.906	.012
V(G)	CC	.087	.649**	1.000	.928**	.016	.549*	-.680**
	Sig.	.757	.009	.	.000	.955	.034	.005
V	CC	.026	.496	.928**	1.000	-.023-	.657**	-.572*
	Sig.	.928	.060	.000	.	.934	.008	.026
SR	CC	.578*	.054-	.016	.023-	1.000	-.007-	-.214-
	Sig.	.024	.849	.955	.934	.	.980	.445
NFaults	CC	.127-	.033	.549*	.657**	-.007-	1.000	-.100-
	Sig.	.653	.906	.034	.008	.980	.	.723
Non-D	CC	.828	.000	.080	.158	.474	.399	.158
	Sig.	.021	1.000	.865	0.735	.282	.375	.605

Where:

FR: Functional redundancy, NJ: Non-injectivity , V(G): McCabe, V: Volume, SR: State redundancy , Nfaults: Number of faults, Fdesnity, Fault Density, Non-D: Non-determinacy and CC: correlation Coefficient.

7.2.3 Regression Results:

We now compare software metrics built based on syntactic features against those built using semantics. Table 7.5 shows a summary of the regression results. R^2 is a measure of variance in the dependent variable that estimated by the model built using certain predictors [97].regression was done by using SPSS software.

TABLE 7.5: SUMMARY OF REGRESSION RESULTS

	Semantic Metric	Syntactic and semantic metrics	R^2
1	State redundancy	Fault Density	0.859
2	Functional redundancy	McCabe	0.501
3	Non_injectivity	Fault Density	0.432
4	Functional redundancy	Number Of Faults	0.259
5	Non_injectivity	State redundancy	0.205
6	Non_injectivity	McCabe	0.213
7	Functional redundancy	Volume	0.110
8		Number Of Faults	0.029
9	Non_injectivity		
10	State redundancy	McCabe	0.010
11	Non-determinacy	Functional redundancy/Non-injectivity	0.067
12		State redundancy	0.008
13	State redundancy	Functional redundancy	0.006
14	Non_injectivity	Functional redundancy	0.002
15	Non-determinacy	McCabe	0

The regression results indicate that state redundancy and fault density have the closest relation. The value of R^2 for the first parameter denotes that 86% of the change in the dependent variables explained by the change in independent variables.

R^2= 0.859 reflects a strong positive relationship. At the same for the last one, where R^2 equals 0.002 (0%), reflects a very week +ve. relationship.

7.3 Analytical Research

The main goal of this section is to figure out how proposed semantic metrics can contribute to the prediction of software reliability in its early stages. As mentioned previously, the study concentrates on measuring the ability of the program to be fault tolerant, in order to, measure such properties we consider the lifecycle of failure as follows:

1. Existence of a fault
2. Fault sensitization
3. Fault propagation
4. Specification violation

The proposed metrics tend to measure these factors as indicator of failure probability. Figure7.1. describes the chain of reactions that happen when faults are executed as follows:

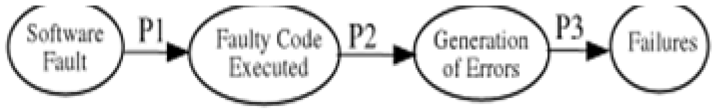


Figure 7.1: Events generate system failure [98]

The probability of software faults resulting into a failure is heavily dependent on the operational profile. Assuming a fault exists, the probability of a faulty code to be executed is p1. If a faulty code is executed, the probability of error generation is p2. If errors are generated, the probability of these errors resulting into failure is p3. Another factor should be considered to reflect whether the resulting failure is violating the specification (P4). Thus, the probability of a software fault resulting into a failure is product of P1, P2, p3 and P4. The following section describes these factors:

7.3.1 Estimating probability of executing faulty statements:

Due to high complexity and constraints involved in the software development process, it is difficult to develop and produce software without faults. So, the aim of this factor is to estimate the probability of executing statements that contain faults. This study makes use of two metrics here, fault density and software size as follows, probability of executing faulty statements= 1(1-fault density)^N. where N represents software size measured in Lines of code.

7.3.2 Probability of sensitization

Executing statements that contain faults may result in generating errors. The probability of sensitization tends to measure, to what extent executed faults can cause errors, errors represent deviations from an expected state to another erroneous state. Both state redundancy and non-injectivity are contributors to this factor as shown below:

$$\text{Probability of sensitization} = 1-2^{(\text{Nj} - \text{initial state redundancy})}$$
$$= 1-2^{(\text{Nj}- \text{K}(б))}$$

7.3.3 Probability of error propagation

Errors can propagate to change the final state to be erroneous, non-injectivity metric used to measure this,

$$\text{Probability of error propagation} = (1- 2)^{\text{NJ} - \text{H}(б))}$$

Where NJ is non-injectivity, H(s) represents the entropy of inputs.

7.3.4 Probability of specification violation:

A program may fail to compute its intended function and yet still behave according to the specification it is intended to satisfy [8]. This part tends to measure the probability that the resulting erroneous final state is also violating the specification. Non-determinacy metric is used here for this purpose.

$$\text{Probability of specification violation (intolerance)} = (1- 2)^{\text{ND} - \text{H}(б\ \text{f})}$$

Where ND is non-determinacy, H($^{\sigma}$ f) represents the entropy of outputs. Figure7.2. Shows data flow diagram for main factors of system failure

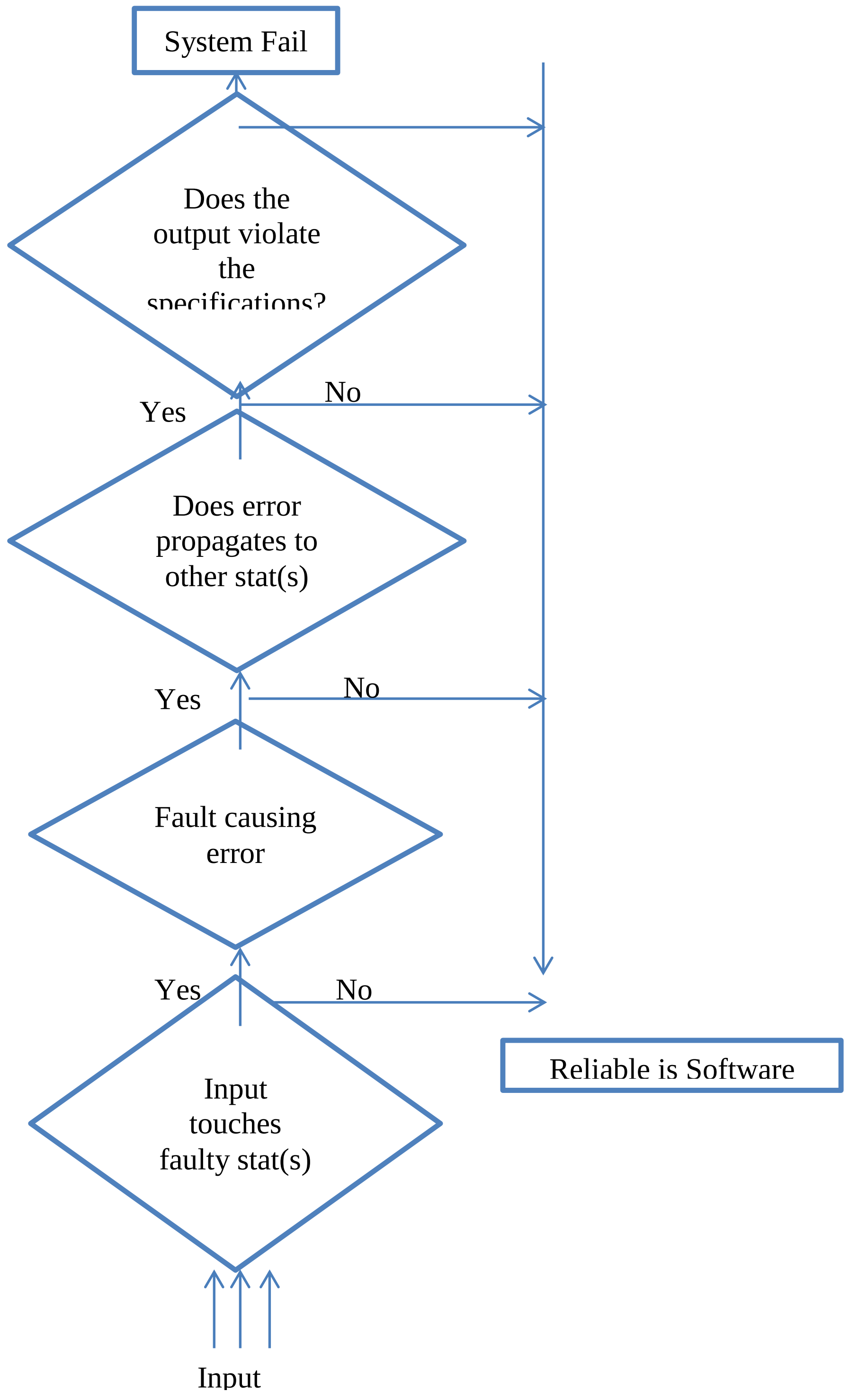

System Fail
Does the output violate the specifications?
Yes
No
Does error propagates to other stat(s)
Yes
No
Fault causing error
Yes
No
Reliable is Software
Input touches faulty stat(s)
Input

Figure 7.2: Flow diagram including factors of system failure

7.4 Results

To predict failure probability as discussed in section (7.3) the product of the above factors are measured. Table 7.6 shows failure probability for sample programs. The probability of failure =P1 x P2 xP3 x P4.

TABLE 7.6: FAILURE PROBABILITY FOR SELECTED PROGRAMS

P. name	P(ex. Faults)	P(sensitization)	P(propagation)	P(violation)	P(Failure)
Tcas	0.8243	1	1	0.9	0.7416
Schedule2	0.312	1	1	0.9	0.2808
Schedule	0.663	1	0.9	0.99	0.59073
Replace	0.432	1	1	0.9	0.3888
Space	1	1	1	1	1
Sorting	0.635	1	1	1	0.635
Totinfo	0.9966	1	1	0.594	0.59198

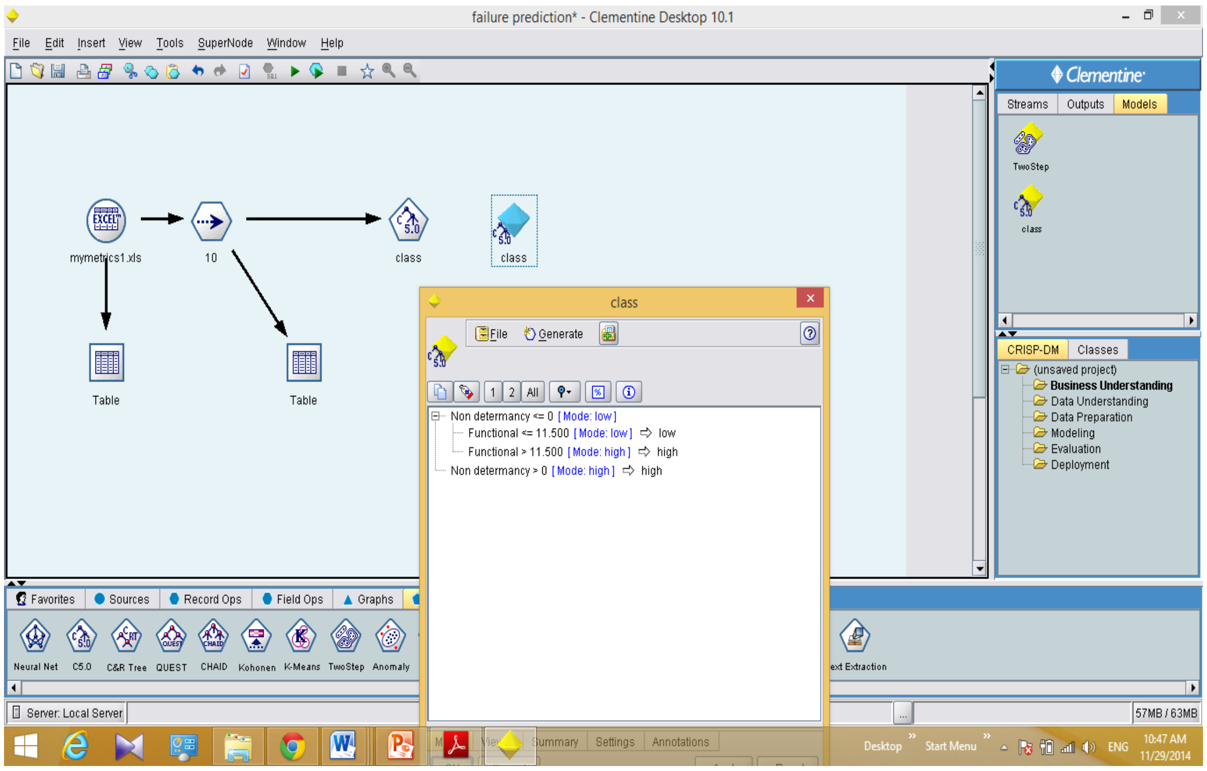
7.6 Using failure classification Model

To build a classification model data mining is used. The C5.0 algorithm was selected to find a relationship between our four semantic metrics and P(failure), in the sense that some of these metrics have more of an effect on system failure. The reason of using C5.0 algorithm is its ability to classify a set of data based on training data and generates a set of rules according to that. C5 is an algorithm developed by Ross Quinlan and is used to generate a decision tree [99]. The decision trees generated by C5.0 can be used for classification; therefore, it is often referred to as a statistical classifier. C5.0 has a number of features such as:

- Speed - C5.0 is significantly faster than other algorithms such as C4.5
- Support for boosting - Boosting improves the trees and gives them more accuracy.
- Weighting - C5.0 allows weighting different cases and misclassification types.

7.6.1 Building classification model

To build a classification model, data minor software was used - clementine software. The sample data that was fed to the model consisted of 7 programs where all factors were measured and P(failure) was further estimated. In order for the model to be accurate the sample was increased by duplicating the existing programs data. The size of the total sample was 24 records, 5 from the total are used as training sample and the rest are testing data fed to perform classification. Each record from the training set was further classified manually to either high or low, the study suggested that the probability of failure will be high if it is more than 0.5 and low if it less. Figure 7.3, shows the classification model.



```
Non determancy <= 0 [Mode: low]
    Functional <= 11.500 [Mode: low] ⇨ low
    Functional > 11.500 [Mode: high] ⇨ high
Non determancy > 0 [Mode: high] ⇨ high
```

Figure 7.3: Classifiction Model

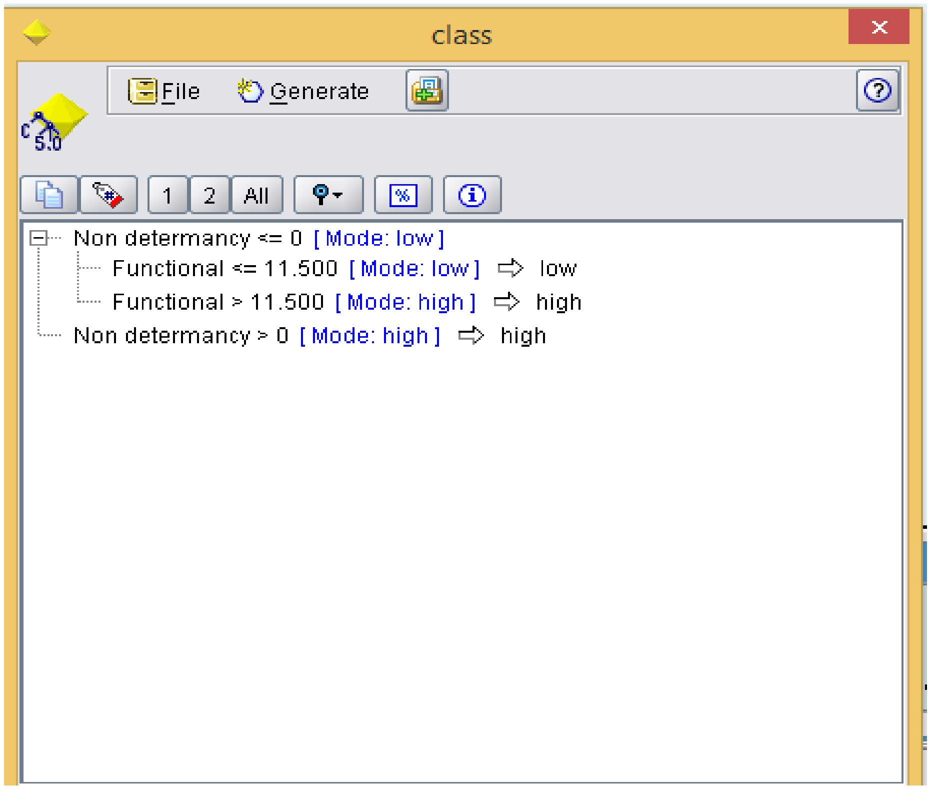
7.6.2 Classification result rules

After applying the classification model, 2 classification rules were detected by a classifier, rules confirmed that there was a relationship between failure probability and both non-determinacy and functional redundancy. So we can predict failure probability class (bigger than 0.5 or less than that) according to these rules. The rules are shown below:

1- If functional redundancy <= 11.5 and non-determinacy <=0 Failure probability will be low(less than 0.5).

2- If functional redundancy > 11.5 and non-determinacy <=0 Failure probability will be high (bigger than 0.5)

3- If Non-determinacy >0 , failure probability will be high.



```
Non determancy <= 0 [ Mode: low ]
    Functional <= 11.500 [ Mode: low ]  ⇨  low
    Functional > 11.500 [ Mode: high ]  ⇨  high
Non determancy > 0 [ Mode: high ]  ⇨  high
```

Figure 7.4: Classication Rules

Based on these rules we can predict the failure probability for any new program where metrics are calculated. Test are done on the sample program printtokins2 which appeared according to these rules to be in high class (failure probability is greater than 0.5) and after real estimation of failure probability the result confirms the observation (P (failure) = 0.98).

7.6.3 Classification Model Limitations:

Building a classification model requires a high, accurate, complete data set for both training and testing purposes. The main difficulty that faced the prediction model is the lack of software specification required to estimate non – determinacy and consequently the estimation of the fourth factor that affects failure probability.

7.7 Chapter Summary

This chapter inspects the validation process along with results. Validation is divided into two parts: empirical and analytical research. Empirical research was used to find correlation between semantic and syntactic metrics, where analytical used to estimate probability of failure statistically based on the main factors derived from the software failure lifecycle.

8.1 Conclusion

At a time when software systems grow increasingly large and complex, it becomes increasingly tenuous/unrealistic to obsess about fault avoidance and fault removal [8].

At the very least, the goal of fault-free software, by whatever means it is achieved, was to be combined with the goal of ensuring that the program is adequately equipped to prevent residual faults from causing failure. The study presented a new set of semantic metrics based on entropy to measure program ability to be fault tolerant with respect to its specification. The proposed metrics are: state redundancy, functional redundancy, non-injectivity and non-determinacy metric. Both empirical and analytical validations were done to assess their fitness to the goal. Empirical validation was used to find correlation between proposed metrics and other syntactic metrics such as: McCabe, Halstead and Fault density. Analytical validation used measure probability of failure by using semantic metrics. The reached results confirmed the ability of these metrics to predict software reliability. Finally, data mining based classification model were done to find out which of the measured metrics has more effect on measuring probability of failure. The main obstacle here is the number of programs that are used as a input sample to classification model. The reason is that lack of available documented programs specifications that can be used to measure non-determinacy metric. Results of this section show that 2 metrics could be used to give indicator to probability of failure, metrics are: function redundancy and non-determinacy.

8.2 Future Work

A number of extensions to the current work could be done in future work:

1. Consolidate the analytical validation by applying estimation techniques to more sample product, for what we have:
    - Source code.
    - Usable specifications.

        - An estimated reliability.

2. Consolidate the empirical validation by extending/ broadening the software for more samples.
3. Explore the possibility of automating the calculation of some semantic metrics by analyzing source code.
4. Study the concept of bandwidth of assertions related to our semantic metrics.

References

[1] Linda Westfall,12Steps to Useful Software Metrics,2005 - available online at floors-utlet.com/specs/spec-t-1-20111117171200.pdf. Access date: 28/3/2014.

[2] N Leveson-2009, software metrics available online: www.core.org.cn/NR/rdonlyres/.../cnotes7. pdf access date 1/4/2014.

[3] Fenton, Norman, and James Bieman. Software metrics: a rigorous and practical approach. CRC Press, 2014.

[4] Yu, Sheng, and Shijie Zhou. "A survey on metric of software complexity". Information Management and Engineering (ICIME), 2010 the 2nd IEEE International Conference on. IEEE, 2010.

[5] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lips Software Metrics SEI Curriculum Module, SEI-CM-12-1.1, December 1988.

[6] Fenton, Norman E., and Martin Neil. "A critique of software defect prediction models." Software Engineering, IEEE Transactions on 25.5 (1999): 675-689.

[7] Leveson, Nancy. "A new accident model for engineering safer systems."Safety science 42.4 (2004): 237-270.

[8] Mili, A., A. Jaoua, M. Frias, and Rasha Gaffer Mohamed Helali. "Semantic metrics for software products." Innovations in Systems and Software Engineering 10, no. 3 (2014): 203-217.

[9] Rawat, Mrinal Singh, Arpita Mittal, and Sanjay Kumar Dubey. "Survey on impact of software metrics on software quality". (IJACSA) International Journal of Advanced Computer Science and Applications 3.1 (2012).

[10] A Whitepaper on Metrics, Andreas Rau, Steinbeis Transferzentrum Softwaretechnik, 1998, 1999, 2001.Last Change: 2001-08-06.

[11] Boehm, Barry W., Brown, J. R, and Lipow, M.: Quantitative evaluation of software quality, International Conference on Software Engineering, Proceedings of the 2nd international conference on Software engineering, 1976.

[12] Kitchenham, B. and Pfleeger, S. L., "Software quality: the elusive target [special issues section]", IEEE Software, no. 1, pp. 12-21, 1996.

[13] Hyatt, Lawrence E. and Rosenberg, Linda H.: A Software Quality Model and Metrics for Identifying Project Risks and Assessing Software Quality, European

Space Agency Software Assurance Symposium and the 8th Annual Software Technology Conference, 1996.

[14] Software Quality Models and Philosophies, online at: https://www.bth.se/com/besq.nsf/(WebFiles)/CF1C3230DB425EDCC125706900317C44/$FILE/chapter_1.pdf.

[15] Adams, E, “optimizing preventive service of software products,” IBM journal of research and development,28(1),pp. 2-14, 1984.

[16] Pressman, R.S, Software engineering: A practitioner Approach , 3rd edn, McGraw-Hill, New York 1992.

[17] Jelinski, Z. and Moranda,PB., “ Software reliability research,” in statistical computer performance evaluation (ed.

[18] Miller D.R, “exponential order statistic models of software reliability growth”, IEEE transactions on software engineering, SE-12(1),PP.12-24.1986.

[19] Shanmugam, Latha, and Lilly Florence. "A comparison of parameter best estimation method for software reliability models." International Journal of Software Engineering & Applications 3.5 (2012): 91-102.

[20] Albrecht, A.J. and J.E. Gaffney, Jr. Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation, IEEE Transactions on Software Engineering SE-9,6 p639-648, Nov. 1983, A comparison between FP, software science and LOC with a detailed appendix on applying FP.

[21] Halstead, M.H. Elements of Software Science, New York: Elsevier North Holland, 1977, the original book by Halstead on his software science.

[22] A. Rau, Steinbeis Transferzentrum Softwaretechnik,. A Whitepaper on Metrics. 1998, 1999, 2001. avilable online at: http://www.it.fhtesslingen.de/~rau/forschung/metrics.htm.

[23] Thomas J McCabe, “A Complexity Measure”, IEEE Transactions On Software Engineering, IEEE, Washington, Oct 1976, pp. 308-320.

[24] Morell, Larry J., and Jeffrey M. Voas. "A framework for defining semantic metrics" Journal of Systems and Software 20.3 (1993): 245-251.

[25] Adrian COSTEA1 PhD, ON MEASURING SOFTWARE COMPLEXITY., University Lecturer Academy of Economic Studies, Bucharest, Romania journal of applied quantive methods. June 2007

[26] Torn, A., T. Andersson and K. Enholm, 1999. “A complexity metrics model for software”. South Afr. Comput. J., 24: 40-48.

[27] Victor R. Basili, and Barry T. Perricone, “Software errors and complexity: an empirical investigation”, Communications of the ACM, ACM, New York, Jan. 1984, pp. 42 – 52

[28] Tu Honglei ; Coll. of Inf. Technol., Beijing Normal Univ., Zhuhai, China ; Sun Wei ; Zhang Yanan, The Research on Software Metrics and Software Complexity Metrics, Computer Science-Technology and Applications, 2009. IFCSTA '09. International Forum on (Volume:1). 25-27 Dec. 2009 , 131 - 136 .

[29] J. Li, J. Chen, and P. Chen, “Modeling web application architecture with UML,” in Proceedings of Technology of Object-Oriented Languages and Systems, 2000, pp. 265-274.

[30] E. Ghosheh, J. Qaddour, M. Kuofie, and S. Black, “A comparative analysis of maintainability approaches for web applications,” in Proceedings of IEEE International Conference on Computer Systems and Applications, 2006, pp. 1155-1158.

[31] Y. Zhang, H. Zhu, and S. Greenwood, “Website complexity metrics for measuring navigability,” in Proceedings of International Conference on Quality Software, 2004, pp. 172-179.

[32] E. Mendes, N. Mosley, and S. Counsell, “Comparison of web size measures for predicting
web design and authoring effort,” IEEE Proceedings Software, Vol. 149, 2002, pp. 86-92.

[33] E. Mendes, S. Counsell, and N. Mosley, “Web metrics − Estimating design and authoring
effort,” IEEE Multimedia, Vol. 8, 2001, pp. 50-57.

[34] W. Jung, E. Lee, K. Kim, and C. Wu, “A complexity metric for web applications based on the entropy theory,” in Proceedings of Asia-Pacific Software Engineering Conference, 2008, pp. 511-518.

[35] W. A. Barrett and J. C. Couch "Compiler construction: Theory and Practice", Science Research Associates, Inc., 1979.

[36] N. F. Schneidewind and H. M. Hoffmann "An Experiment in Software Error Data Collection and Analysis", IEEE Trans. on Software Eng., vol. SE-5, no. 3, pp.256 -286 1979.

[37] M. Lipow "Number of faults per line of code", IEEE Trans. on Software, vol. 8, 1982

[38] Giger, E. ; Pinzger, M. ; Gall, H.C., Can we predict types of code changes? An empirical analysis. Mining Software Repositories (MSR), 2012 9th IEEE Working Conference. June 2012, Zurich,217-226

[39] Zeeshan Ahmed and Saman Majeed, Measurement, Analysis with Visualization for Better Reliability

[40] Software Quality Analysis System : a New Approach, Nadine MESKENS. 0-7803-2775-6/96 $4.00 0 1996 IEEE

[41] H. Basson and J.-C. Derniame "Towards an Evolutive Kernel of Measurements on Ada Sources Developed on an Integrated Software Engineering Environment", ACM Proceedings of 7th Washington Ada Symposium, 1990.

[42] H. Basson and J.-C. Derniame "Quality Tree Extensions and Partial Instantiation for Ada Objects", ACM Proceedings of 8th Washington Ada Symposium, 199.

[43] Je_rey M. Voas, Keith W. Miller, Je_ery E. Payne, Designing Programs That Are Less Likely To Hide Faults_1993. Volume 20, Number 1, January 1993.

[44] M. J. Ordo˜nez and H. M. Haddad. The state of metrics in software industry. In Proc. of the Fifth International Conference on Information Technology: New Generations, (ITNG 2008), pages 453–458, LasVegas, Nevada, USA, Apr. 7-8 2008. IEEE Computer Society.

[45] Mertik, M., Lenic, M., Stiglic, G., and Kokol, P, “Estimating Software Quality with Advanced Data Mining Techniques”, International Conference on Software Engineering Advances, IEEE, Tahiti, Oct. 2006, pp. 19-19

[46] Hartson, H. Rex and Smith, Eric C. and Henry, Sallie M. and Selig, Calvin (1987) Design Metrics Which Predict Source Code Quality. Technical Report TR-87-32 ...

[47] T.M. Khoshgoftaar and N. Seliya, "Fault Prediction Modeling for Software Quality Estimation: Comparing Commonly Used Techniques," Empirical Software Eng., vol. 8, no. 3, pp. 255-283, 2003.

[48] Salwa K, A Metrics-Based Data Mining Approach for Software Clone Detection. Pages 35-41 , COMPSAC '12 Proceedings of the 2012 IEEE 36th Annual Computer Software and Applications Conference IEEE Computer Society Washington, DC, USA ©2012.

[49] Menzies, T., Greenwald, J., and Frank, A., "Data Mining Static Code Attributes to Learn Defect Predictors", IEEE Transactions on Software Engineering, IEEE, San Francisco, Jan. 2007, pp. 2-13.

[50] W. Tang and T.M. Khoshgoftaar, "Noise Identification with the KMeans Algorithm," Proc. Int'l Conf. Tools with Artificial Intelligence (ICTAI), pp. 373-378, 2004.

[51] Larry J. Morel, Jeffrey M. Voas, A framework for defining semantic metrics.

[52] Gall, C. S. Inf. Technol. & Syst. Center, Univ. of Alabama in Huntsville, Huntsville, AL Lukins, Stacy K.; Etzkorn, Letha H.; Gholston, Sampson; Farrington, Phillip A.; Utley, Dawn R.; Fortune, J.; Virani, Shamsnaz, Semantic software metrics computed from natural language design specification. Volume: 2, Issue: 1 Page(s): 17 – 26.2008.

[53] Voas, J. M. and K. W. Miller (1993). Semantic metrics for software testability. Journal of Systems and Software 20(3), 207–216

[54] Semantic Metrics, Conceptual Metrics, and Ontology Metrics: Letha H. Etzkorn

[55] Bo Hu, Yannis Kalfoglou, Harith Alani, David Dupplaw, Paul Lewis, Nigel Shadbolt Managing Knowledge in a World of Networks Lecture Notes in Computer ScienceVolume 4248, 2006, pp 166-181. Semantic Metrics.

[56] Zschaler, S.: Towards a Semantic Framework for Non-Functional Specifications of Component-Based Systems. In: IEEE (ed.) Proceedings of the 30th EUROMICRO Conference 2004, Rennes, France, 31 August - 3 September 2004, pp. 92–99. IEEE Computer Society Press, Los Alamitos (2004).

[57] Bilong Wen, Li Zhang, An arithmetic of mapping enterprise process metrics to information model based on semantics, April 2010, Volume 15, Issue 2, pp 121-126

[58] Gabriele Bavota · Andrea De Lucia · Andrian Marcus · Rocco Oliveto, Using structural and semantic measures to improve software modularization, published online: 14 September 2012. © Springer Science+Business Media, LLC 2012

[59] Maletic JI, Marcus A (2001) Supporting program comprehension using semantic and structural information. In: Proceedings of 23rd international conference on software engineering. IEEE CS Press, Toronto, Ontario, Canada, pp 103–112

[60] Giger, Emanuel, Martin Pinzger, and Harald C. Gall. "Comparing fine-grained source code changes and code churn for bug prediction." *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 2011.

[61] M. M. Lehman and L. A. Belady, Eds., Program evolution: Processes of software change. San Diego, CA, USA: Academic Press Professional, Inc., 1985.

[62] M. W. Godfrey and Q. Tu, "Evolution in Open Source software: A case study," in Proceedings of the International Conference on Software Maintenance, San Jose, California, 2000, pp. 131-142.

[63] Selim Kebir, Abdelhak-Djamel Seriai, Sylvain Chardigny, Allaoua Chaoui, Quality-Centric Approach for Software Component Identification from Object-Oriented Code, WICSA-ECSA '12 Proceedings of the 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture. Pages 181-190

[64] S. Allier, H. A. Sahraoui, S. Sadou, and S. Vaucher, "Restructuring object-oriented applications into component-oriented applications by using consistency with execution traces," in Proceedings of the 13th international conference on Component-Based Software Engineering, ser. CBSE'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 216-231.

[65] D. Birkmeier and S. Overhage, "On component identification approaches - classification, state of the art, and comparison," in Proceedings of the 12th International Symposium on Component-Based Software Engineering, ser. CBSE '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 1-18.

[66] S. D. Kim and S. H. Chang, "A systematic method to identify software components," in Proceedings of the 11th Asia-Pacific Software Engineering

Conference, ser. APSEC '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 538-545. [Online]. Available: http://dx.doi.org/10.1109/APSEC.2004.11

[67] S. K. Mishra, D. S. Kushwaha, and A. K. Misra, "Creating reusable software component from object-oriented legacy system through reverse engineering," Journal of Object Technology, vol. 8, no. 5, pp. 133-152, 2009.

[68] D. Melamed, "Measuring Semantic Entropy," Proceedings of the SIGLEX Workshop on Tagging Text with Lexical Semantics, Washington, DC, 1997.

[69] P. F. Brown, S. Della Pietra, V. Della Pietra, R. Mercer, "Word Sense Disarnbiguation using Statistical Methods", Proceedings of the ~9th Annual Meeting of the Association for Computational Linguistics, Berkeley, Ca., 1991.

[70] D. Yarowsky, "One Sense Per Collocation," DARPA Workshop on Human Language Technology, Princeton, N J, 1993.

[71] Abd-El-Hafiz, Salwa K. An information theory approach to studying software evolution [J]. AEJ - Alexandria Engineering Journal, v 43, n 2, March, 2004, p 275-284.

[72] Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems Software Engineering, IEEE Transactions on Date of Publication: March-April 2008, Marcus, Andrian. Wayne State Univ., Detroit. Poshyvanyk, Denys; Ferenc, Rudolf
Volume: 34, Issue: 2 . Page(s): 287 – 300.

[73] Chidamber, Shyam and Chris Kemerer, "A Metrics Suite for Object-Oriented Design," IEEE Transactions on Software Engineering, pp. 476- 492, June 1994.

[74] B. Neate, W. Irwin, and N. Churcher. Coderank: A new family of software metrics. In J. Han and M. Staples, editors, ASWEC2006: Australian Software Engineering Conference, pages 369-378, Sydney, Apr. 2006. IEEE.

[75] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Component rank: Relative significance rank for software component search. In Proc. of the 25th International Conference on Software Engineering, (ICSE 2003), pages 14–24, Portland, Oregon, USA, May 3-10 2003. IEEE Computer Society.

[76] Zhou Y., Research on Software Measurement [Ph.D. Thesis], Department of Computer Science and Engineering, Southeast University, Nanjing, P.R. of Chaina, 2002.

[77] Yi. Shin ,K. Kim and C. Wu, “Complexity measures for object-oriented program based on the entropy,” in Proceedings of Asia-Pacific Software Engineering Conference, 1995, pp. 127-136.

[78] M. Lorenz and J. Kidd. Object-oriented software metrics: a practical guide. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[79] Y Gil, Maayan Goldstein, Dany Moshkovich, How Much Information Do Software Metrics Contain, PLATEAU '11 Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools, Pages 57-64 ACM New York, NY, USA ©2011

[80] G. Lajios. Software metrics suites for project landscapes. In Proc. Of the 2009 European Conference on Software Maintenance and Reengineering., volume 0, pages 317–318, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[81] E. B. Allen, “Measuring graph abstractions of software: An information-theory approach,”
in Proceedings of IEEE Symposium on Software Metrics, 2002, pp. 182- 193.

[82] Yi T. and Wu F., Empirical Analysis of Entropy Distance Metric for UML ClassDiagrams, ACM SIGSOFT Software Engineering Notes, 2004.

[83] Zhou Y. and Xu B., Dependence structure analysis-based approach for measuring importance of classes, Journal of Southeast University (Natural Science Edition), 2008; 38(3): 380-384.

[84] Matinee Kiewkanya* and Pornsiri Muenchaisri, Constructing Modifiability Metrics by Considering Different Relationships. 2011, Chiang Mai J. Sci. 2011; 38 (Special Issue) : 82-98 www.science.cmu.ac.th/journal-science/josci.html Contributed Paper

[85] Kang D., Xu B., Lu J. and Chu W.C., A Complexity Measure for Ontology Based on UML. Proceedings of the 10th IEEE International Workshop on Future Trends of Distrubuted Computing Systems, Suzhou, Chaina, May, 2004; 222-228.

[86] Panchenko, O. ; Hasso Plattner Inst. for Software Syst. Eng., Potsdam, Germany ; Mueller, S.H. ; Zeier, A. Measuring the quality of interfaces using source code entropy, Industrial Engineering and Engineering Management, 2009. IE&EM '09. 16th International Conference, Oct. 2009, Page(s): 1108 - 1111

[87] N. Chapin. “An entropy metric for software maintainability”. In Proceedings of the 22nd Annual Hawaii International Conference on System Sciences, Software Track, pages 522{523. IEEE, October 1989.

[88] L. H. Etzkorn, S. Gholston, and W. E. Hughes, Jr. “Semantic entropy metric. Journal of Software Maintenance: Research and Practice”, 14(4):293

[89] Research Methodology, online at: http://www.is.cityu.edu.hk/staff/isrobert/phd/ch3.pdf.

[90] Northrop L, Feiler P, Gabriel RP, Goodenough J, Linger R, Longstaff T, Kazman R, KleinM, Schmidt D, Sullivan K,Wallnau K (2006) Ultra large scale systems: the software challenge of the future. Software Engineering Institute, July 2006.

[91] Patterson D, Fox A (2005) Recovery oriented computing—an overview. Technical report, University of California at Berkeley. http://roc.cs.berkeley.edu/roc_overview.html.

[92] Sullivan, Mark, and Ram Chillarege. "Software defects and their impact on system availability: A study of field failures in operating systems." FTCS. 1991.

[93] Csiszar I,Koerner J (2011) Information theory: coding theorems for discrete memory less systems. Cambridge University Press, Cambridge,UK

[94] Brink C, Kahl W, Schmidt G (1997) Relational mathematics in computer science. Advances in computer science. Springer, Berlin

[95] Mathematics for Computer Science. http://www.pling.org.uk/cs/mcs.html

[96] Software-artifact Infrastructure Repository, online: http://sir.unl.edu/portal/bios/tcas.php. Access date: 2/2014.

[97] Lindley, D.V. (1987). "Regression and correlation analysis," New Palgrave: A Dictionary of Economics, v. 4, pp. 120–23.

[98] Madeira, Henrique, João Durães, and Marco Vieira. "Emulation of software faults: Representativeness and usefulness." *Dependable Computing*. Springer Berlin Heidelberg, 2003. 137-159.

[99] Rokach, Lior; Maimon, O. (2008). *Data mining with decision trees: theory and applications*. World Scientific Pub Co Inc. ISBN 978-9812771711.