CSIT (March 2013) 1(1):51–66
DOI 10.1007/s40012-012-0003-z

ORIGINAL RESEARCH

An empirical approach for early estimation of software testing effort using SRS document

Ashish Sharma · Dharmender Singh Kushwaha

Received: 12 May 2012 / Accepted: 7 October 2012 / Published online: 6 November 2012
© CSI Publications 2012

Abstract Software testing is one of the most important and critical activity of software development life cycle, which ensures software quality and directly influences the development cost and success of the software. This paper empirically proposes a test metric for the estimation of the software testing effort, using IEEE-Software Requirement Specification (SRS) document, which aims to avoid budget overshoot, schedule escalation etc., at very early stage of software development. The effort required to test the software depends on the complexity of the proposed software. Hence, the paper first proposes to estimate the requirement based complexity of the proposed software on the basis of SRS document and further the estimation of software testing effort is carried out on the basis of requirement based complexity. The proposal is also compared with various leading and prevalent techniques for software complexity estimation and test effort estimation as proposed in the past. The result obtained validates the claim that the proposed measures are comprehensive one and compares well with various established measures. It shall act as early warning system for staffing and delivery schedules.

Keywords SRS · Requirement based complexity · Requirement based test function point · Requirement based testing effort · Technical and environmental factors

A. Sharma (✉) · D. S. Kushwaha
Department of Computer Science & Engineering, Motilal Nehru National Institute of Technology, Allahabad, India
e-mail: ashish.sharma@gla.ac.in

D. S. Kushwaha
e-mail: dsk@mnnit.ac.in

1 Introduction

There has been a continuous effort to estimate the critical activities associated with the development of software, but previous researches reveal that very few proposals exist that can estimate the software development activities from software requirements itself. The requirement of the software is the capability, that a system must supply or a quality that a system must possess in order to solve a problem of poor estimation and to achieve an objective within the system's conceptual domain [1]. Defining the user requirements is arguably one of the most difficult and challenging task for the development of complex systems [2]. Thus, before designing a software product, it is extremely necessary to understand the precise requirements of the customer and document them according to IEEE 830:1998 recommendations [3]. There are obligations that should be fulfilled for software to be developed along with other general and optional requirements. This paper considers the following issues that have not received much attention in majority of the research proposals that exist, till now, such as:

- First, early estimation of requirement based complexity of software to be developed on the basis of software requirements written as per IEEE 830: 1998 recommendations, and,
- A metric for early estimation of software testing effort from the requirement based complexity of software to be developed.

In order to address these issues, the proposed approach consists of the following two modules:

- A measure for early estimation of software complexity on the basis of SRS is proposed for software to be developed, so that the design and code decisions

pertaining to the software development can be made in advance.

- Finally, the paper proposes a measure for the estimation of software testing effort, as derived from requirement based complexity. This shall enable the developer, tester and practitioner to plan the testing and its associated attributes like test cases generation, test team productivity, computation of function point etc. in advance.

This research work aims to shift the conventional practices of software engineering to the requirement based software engineering paradigm by estimating the software development activities at a much early phase of software development life cycle.

The rest of the paper is organized as follows. Section 2 discusses a survey of leading researches carried out so far in various areas related to proposed approaches. Section 3 discusses the broad structure and major constituents of SRS for documentation of elicited software requirement in IEEE 830:1998 format. It further proposes an improved requirement based complexity (IRBC) measure for early estimation of software complexity on the basis of attributes extracted from SRS of the proposed software. This measure is also evaluated against Weyuker's properties for validation purposes. Section 4 proposes a measure for estimation of software testing effort based on IEEE SRS and IRBC measure. In Sect. 5, the proposed test effort estimation measure is compared with various prevalent development effort measures used by practitioners as proposed in the past. The final section concludes the work followed by references.

2 Related works

This section carries out a survey of leading papers describing the work carried out so far related to software complexity computation and software testing effort estimation.

2.1 Literature review for estimation of software complexity

Various research proposals have been put forward in establishing the code based complexities with different dimensions and parameters. The software complexity measures proposed in the past are broadly classified into code, cognitive and object oriented (OO), and requirement based complexity measures. These measures use source lines of code (LOC) for the estimation of software complexity. For the sake of completeness, a study and survey of leading papers for the estimation of software complexity are presented here. Mc Cabe [4] uses a fundamental assumption, that the software complexity is related to the number of control paths generated by the code. It describes a graph based complexity measure for control flow programs, that is independent of physical size of a program and depends on decision structure. Also there is no penalty for embedded loops and a series of single loops. Chaudhary and Shastrabuddhe [5] discuss about the study of the effect of meaningfulness of a program and its understanding and consider difficulty or psychological complexity of program. Hamer [6] describes the test made for the validity of the relationships and interpretations that form the foundation of software science. The measure depends on occurrence of operators and operands. Weyuker [7] discusses about the various axiomatic approaches for the evaluation of complexity measure using Weyuker properties. Davis and LeBlanc [8] discuss about how a complexity measure can serve as a good predictor based on the components, type of elements and structural relationship. Zweben and Gourlay [9] discusses about the fundamental nature of property and precision for the structural and mutant adequacy of a complexity measure. Chariniavsky and Smith [10] proposes complexity metric for mapping of program, tokens, variables and constants to estimates final complexity measure based on a recursive function. Halstead [11] proposes a measure for the estimation of difficulty based on the principle of count of operators and operands and their respective occurrences in the code for the length, vocabulary, volume, potential volume, estimated program length, difficulty and finally the estimation of effort and time. Yu and Zhou [12] provides a comprehensive survey of the metric for software complexity using some classic and efficient software complexity metrics, such as LOCs, halstead and cyclomatic complexity metric. Morozoff [13] discusses about the calculation of code rework based on code re-use. It considers various factors for rework such as—initial size, code added, code modified, total final size, final versus initial comparison, percentage of re-work for the calculation of effective LOC based on a tool.

Harrision [14] gives complexity measure based on information content of program and discusses that basic control structure (BCS), input, output and operands, have direct bearing on the cognitive complexity of the code. Tian and Zelkowitz [15] describes various axioms for defining the boundaries where complexity feasibility can fit and develops a technique for general complexity measure on the basis of formal model of complexity. Shao and Wang [16] uses cognitive weight as degree of difficulty for software using BCS. Gray et al. [17] describes the minimization of cognitive workload for interactive system and also defines architecture of cognition. Kushwaha and Misra [18] gives computation of cognitive information complexity measure (CICM) that uses identifiers, operands and LOC based on BCS. Further the measure is also evaluated through Weyuker's properties. Auprasert and Limpiyakorn [19] considers various cognitive complexity measures and proposes structured complexity measure by decomposing

the program into granular hierarchial structure and further computes structured cognitive information measure based on various definitions.

Aggrawal et al. [20] considers OO programs for complexity analysis on the basis of errors and exception handling. It proposes two new metrics i.e. number of catch block per class (NCBC) and exception handling routine (EHF). These metrics are also evaluated against Weyuker's Properties. Chhabra and Gupta [21] evaluates OO spatial complexity measures using formal evaluation frameworks as proposed by Weyuker.

Baccarini [22] describes complexity with reference to project dimensions at organizational, technological and managerial levels. The project complexity is measured in terms of differentiation and interdependencies among levels. Klemola and Rilling [23] propose a kind of line of code identifier density (KLCID) based cognitive complexity measure. It defines identifiers as programmers' defined labels (variable name, class name and object name) in order to define the identifier density (ID). Noble and Letsky [24] describes transactive memory model as collection of metric based on individual and team based structure for various collaboration models like team-work collaboration and task-work collaboration. Singh et al. [25] describes estimation of software complexity on the basis of requirements. It also provides a method based on strength of the individual and group, followed by dependency matrix. Din [26] defines a set of metrics to indicate the quality of software requirements based on the goodness property criteria and uses chunks as a basic language parsing unit for noun phrase count for coupling and cohesion. Boer and Vliet [27] gives arguments for notion of relationship between requirement problem description and software architecture. Kanjilal et al. [28] define Requirement Complexity Measurement Metric for measuring the complexity of the requirements and calculate the complexity factor based on sequence diagram. Fitsilis [29] presents a study and models the software complexity based on study carried out by Project Management Body of Knowledge. It also considers various patterns for complexity based on faith, fact and interaction to identify the uncertainties in project.

2.2 Literature review for estimation of software testing effort

During the last few decades, various models, methods and techniques have been developed to estimate the test effort for software to be developed. This section presents a survey of prevalent testing practices which are categorized into code based, requirement based and complexity & model based methods for the estimation of software testing effort.

Tai [30] proposes the exploration of testing complexity for several classes of programs, based on testing path that is obtained on the basis of test data. Ryser et al. [31] discusses about a comprehensive survey of requirement based validation and test of software. Aurum et al. [32] discusses various inspection methods with Fagan's Inspection. Holden and Dalton [33] uses cumulative test analysis for test selection and produces an objective measure upon identifying and assigning impact of risk for test effectiveness. Bertilino [34] discusses testing roadmap for the achievements, challenges in software testing and discusses four dreams as, efficacy maximized test engineering, 100 % automatic testing, test based modelling and universal test theory, for the testing of any software. Aranha and Borba [35] discusses test execution and a test automation effort estimation model for the test case selection on the basis of a controlled natural language and uses a manual coverage and automated test case generation technique for effort estimation. Deckkers [36] provides a method called test point analysis, which uses function points for the estimation of final result. It considers three elements to compute test effort i.e. software size, strategy of testing, and the productivity level of test teams.

Whalen et al. [37] describes structural coverage metrics obtained directly from high-level formal software requirements. These metrics provides objective, implementation, and independent measures, of how well a black-box test suite exercises a set of requirements. Rajan et al. [38] proposed an approach to automate the generation of requirements based tests for the model validation, in order to formalize the requirement using linear temporal logic properties for the test case generation. Sneed [39] uses a test strategy and automatically performs requirement analysis by identifying keywords in requirement analysis phase and generates the test cases. Uusitalo et al. [40] provides a set of practices that can be applied to link the requirements with testing based on interdependencies and linking the people with requirement documentation. Yi et al. [41] presents an empirical study on early test execution effort estimation based on test case number prediction from use case and estimating the test effort using test execution complexity. Ramchandran [42] proposes a model for test process and investigates the possibility of deriving the test cases from system models and requirement analysis techniques. Nageshwaran [43], discuss a use-case based approach for the estimation of test effort based on use case weight, use case points (UCPs) and complexity factors. de Almeida et al. [44] describes a method for test effort based on the information contained in use case. It also considers the various parameters like actor ranking and technical environment factor in order to finally arrive at test effort estimation.

Vaysburg et al. [45] discusses an approach for the reduction of requirement based test suites using Extended Finite State Machine (EFSM) dependence analysis. The technique

supports test case generation from EFSM system models. Kim and Sheldon [46] uses formal methods and language Z for the specification of testing that is to be carried out using finite state machine and data item based approach. Aranha et al. [47] uses a tool to convert test specification into natural language for the estimation of the test effort. It also finds test size and test execution complexity measure. Cabral and Tamai [48] uses a tool "Controlled Natural Language (CNL)" to specify requirement and support in testing and execution of the test cases with their implementation. It also uses formal methods for the purpose of verification. Guerreiro e Silva et al. [49] considers effort estimation model based on data analysis, hypothesis formulation, evaluation, accumulated efficiency and finally models the test effort. Lokan and Mendes [50] considers two types of chronological splits i.e. project by project split and date-base split. Further both splits are compared with each other in order to check which of the either leads to better prediction accuracy. Kushwaha and Misra [51] presents an approach of cognitive information based complexity measure (CICM) and models the software test effort based on component off the shelf and component based software engineering. It further correlates CICM with cyclomatic number. The approach is able to demonstrate that the cyclomatic number increases with increase in software complexity. Boehm [52] aims at estimating the development effort from COCOMO-I, for small to medium sized software projects. Further, Boehm et al. [53] discusses the revised version of COCOMO-I as COCOMO-II. It uses some fixed values for one constant and the other constant depends on the scaling factors.

Periyasamy and Liu [54] proposed a set of metrics for object-oriented programs that has been targeted towards estimating the testing efforts. Their metric set is divided into metrics based on inheritance hierarchy, polymorphism and interaction among objects. The metric value only provides the subjective estimation of testing effort.

3 Estimation of software complexity using SRS

This section aims at proposing a measure for early estimation of software complexity on the basis of the SRS of the proposed software.

SRS is an unambiguous and complete software specification document for software. IEEE 830:1998 [3] elaborates the recommended practices for documenting the software requirements to generate a Software Requirement Specification (SRS). The software development starts only after documenting the software requirements using SRS. The contemplation is that, if the entire software is implemented based on this document, can we not estimate the various software development activities too from this SRS?

Since complexity analysis has an extremely high payoff for the investment, hence it is necessary to carry out a precise and accurate estimation of software complexity for software to be developed.

The process starts by extracting the attributes from SRS of the proposed software and later the extracted attributes are arranged in order to obtain various contributing complexity measures for the estimation of requirement based software complexity. For more precise and accurate estimation, external references [52, 53] are also considered. Further, these contributing complexity measures are consolidated to build an IRBC measure.

3.1 Computation procedure

The computation method for the proposed complexity measure is explained in the following sections.

3.1.1 Input output complexity (IOC)

IOC is a combination of input complexity (IC), output complexity (OC) and storage complexity (SC). IC refers to the complexity of the information entering into the system. Similarly, OC refers to the complexity of the information received from the system. Both the IC and OC depend on two major parameters i.e. source and type of information. These two parameters are considered to differentiate various types of information either entering or leaving the system with different sources or locations for passing or receiving the input–output information. Tables 1 and 2 illustrate the details of various types and sources for information along with their relative weights. These weights are identified by considering the varying degree of difficulty at implementation level.

Since, types and sources of input and output information have dependency and relationship on each other as illustrated in Table 3. Hence, we propose IC and OC as product of source and type of information (entering or leaving the system).

The IC is calculated on the basis of information entering into the system and is expressed as:

Table 1 Sources for input and output to system

Parameters	Source	Weight
Input/output source	External input/output through devices	1
	Input/output: files, database, process, software, hardware etc.	2
	Input/output: outside system i.e. network, internet etc.	3

Table 2 Types of input and output

Parameters	Type of input	Weight
Input/output type	Text, character, integer, float etc.	1
	Image, picture, graphics, animation etc.	2
	Audio, video etc.	3

Table 3 Dependency and relationship between types and sources for IC and OC

Input/output domain	Text based	Graphics/image based	Audio/video based
Through console	√	√	√
Through files, database	√	√	√
Through network or internet	√	√	√

Table 4 Types of data storage and relative count

Parameters	Type	Count
Data storage	Local data storage	1
	Remote data storage	2

$$Input\ complexity = \sum_{i=1}^{3}\sum_{j=1}^{3} N_{ij} \times (Source)_i \times (Type)_j,$$

where "N" is the number of external inputs for the proposed software.

Similarly OC is computed on the basis of the output produced by the system and is expressed as:

$$Output\ complexity = \sum_{i=1}^{3}\sum_{j=1}^{3} O_{ij} \times (Source)_i \times (Type)_j,$$

where "O" is the number of external outputs for the proposed software.

Further, in order to obtain IOC, the data storage requirement for the proposed software is also to be considered using SC. The SC uses two major types of storage requirements i.e. local (data storage within the system) and remote (data storage outside the system) for the quantification of storage requirements. Table 4 shows the various types of data storage and their relative weights based on the varying levels of requirement.

The SC is obtained by identifying the number of storage type and their corresponding count for the proposed system. A sum of product for number of storage and their associated count is taken for the computation of SC, this is expressed as:

$$Storage\ complexity = \sum_{i=1}^{n} (Storage\ requirement)_i \times \sum_{j=1}^{2} (Type\ count)_{ij}.$$

Finally, the IOC is expressed as a combination of all three sub-complexities and expressed as

$$IOC = Input\ complexity + output\ complexity + Storage\ complexity.$$

3.1.2 *Functional requirement (FR)*

A FR describes the functionality to be performed by the proposed software and/or its associated components. It also describes the fundamental action that is to be performed in order to accept and process the inputs and generate the subsequent output. It is appropriate to partition the FR into sub-functions or sub-processes in order to get complete functionality coverage. Hence FR with its decomposition is mathematically expressed as:

$$Functional\ requirement = \sum_{i=1}^{n} (functionality)_i \times \sum_{j=1}^{m} (SPF)_{ij},$$

where "SPF" is sub-processes functionality as obtained on decomposing the overall functionality of proposed software.

3.1.3 *Non functional requirement (NFR)*

NFRs include constraints and quality related attributes. Quality is the property or characteristics of the system that its stakeholders expect about and that shall affect their degree of satisfaction with the system. Constraints are not subject to negotiation and, unlike qualities, are a compulsory part of design. The NFRs, in contrast to the FRs, specifies the criteria to be used in order to judge the overall usability of the proposed system, rather than any specific behavior. There are various models available for the identification of quality related requirements, but we consider the famous ISO-9126 model [55] as shown in Fig. 1. The model uses six core quality attributes i.e. functionality, reliability, usability, efficiency, maintainability and portability and twenty sub-attributes. The basis for deriving the value of NFR is illustrated in Table 5. The need and applicability of these attributes, for the proposed software, is determined first and marked with "R" for required and "NR" for not required. Similarly, the status of corresponding sub attributes is marked with "Y" for yes and "N" for no. Finally, the value of NFR is obtained based on

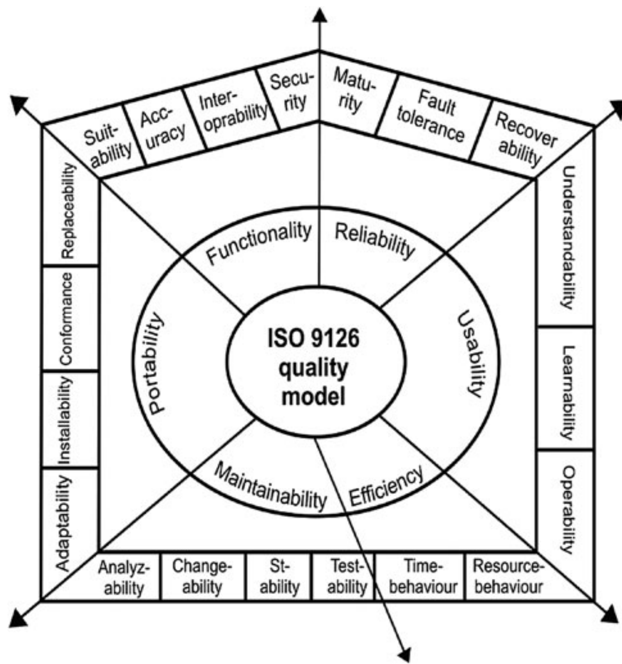


Fig. 1 ISO-9126 model for quality related requirements

Table 5 Dependency and relationship between attributes and sub-attributes based on their existence

Attributes	Status		Sub-attributes	Status	
	Required	Not required		Yes	No
Functionality	R	NR	Suitability	Y	N
	R	NR	Accuracy	Y	N
	R	NR	Interoperability	Y	N
	R	NR	Security	Y	N
Reliability	R	NR	Maturity	Y	N
	R	NR	Fault tolerance	Y	N
	R	NR	Recoverability	Y	N
Usability	R	NR	Understandability	Y	N
	R	NR	Learn ability	Y	N
	R	NR	Operability	Y	N
Efficiency	R	NR	Resource behavior	Y	N
	R	NR	Time behavior	Y	N
Maintainability	R	NR	Analyzability	Y	N
	R	NR	Changeability	Y	N
	R	NR	Stability	Y	N
	R	NR	Testability	Y	N
Portability	R	NR	Adaptability	Y	N
	R	NR	Install ability	Y	N
	R	NR	Conformance	Y	N
	R	NR	Replace ability	Y	N

sum of product of attributes with corresponding sub-attributes.

In order to compute the value of NFR for the proposed software, the suitable attributes and their corresponding sub-attributes are to be identified from ISO-9126 model and a sum of product of these attributes is taken to compute the NFR. It is mathematically expressed as:

$$NFR = \begin{cases} \sum_{i=1}^{6} (Attribute)_i \sum_{j=1}^{m} (Subattribute)_{ij} \\ 1 \; otherwise \; for \; inbuilt \; quality \; attributes \end{cases},$$

where “m” is number of sub attribute for corresponding quality attribute.

If NFR for the proposed software is not explicitly stated, then the minimum value of NFR will be 1, as built in quality attributes are also to be considered.

3.1.4 Requirement complexity (RC)

This complexity describes about the overall requirements of the proposed software. RC is defined as combination of functional and NFRs. The requirements (i.e. FR and NFR) must be explicitly specified, and if the value of NFR is unity, then the total RC of the proposed software will be equal to its FR. Hence the FR is added to the NFR in order to obtain the RC of the proposed software that is expressed as:

$$RC = FR + NFR.$$

3.1.5 Product complexity (PC)

The PC for software to be developed is expressed as a product of RC with IOC. Both the contributing complexity i.e. RC (FR and NFR) and IOC (IC, OC and SC) comprises of the sub complexities that are sufficient to provide an estimate about end PC. Hence, the recursive call and addition of the sub complexities creates dependency between various contributing complexities i.e. RC and IOC, on the basis of overall requirements. Hence IOC is multiplied with RC in order to obtain PC and it is expressed as:

$$PC = IOC \times RC.$$

3.1.6 Personal complexity attributes (PCA)

It refers to technical expertise of developer(s) involved in the development of proposed software based on their personal capabilities. This has been referenced through constructive cost model (COCOMO) of intermediate category [52, 53]. The details of PCA along with their associated ratings are illustrated in Table 6.

Table 6 Personal complexity attributes and their associated values

Attribute	Rating				
	Very low	Low	Nominal	High	Very high
Analyst capability	1.46	1.19	1.00	0.86	0.71
Application exp.	1.29	1.13	1.00	0.91	0.82
Programmer cap.	1.42	1.17	1.00	0.90	–
Virtual machine exp.	1.21	1.10	1.00	0.90	–
Programming lang exp.	1.14	1.07	1.00	0.95	–

The COCOMO describes four categories like project, product, hardware and personal as effort multipliers for cost drivers. The PCA includes five sub-attributes which range from very low to very high as illustrated in Table 6. The inclusion of these personal capability attributes provides a better estimate of the development effort. Hence, in order to carry out an effective software development and to provide better and accurate estimate for complexity of software to be developed, the PCA is considered for inclusion for estimation of software complexity. The PCA is obtained by applying sum of product of the personal attributes with their corresponding rating as:

$$PCA = \prod_{i=j=1}^{5} (AttributeValue)_{ij}.$$

3.1.7 Design constraints imposed (DCI)

DCI describes the constraint imposed on the software development process. It considers various types of design constraints for the development of software. These design constraints are stated by any statuary body/internal-external agencies etc. These design constraints could be regulatory constraint, hardware constraint, software constraints, communication and network constraint, database constraint and other constraints associated to the software. Hence, DCI is mathematically expressed as:

$$\mathrm{DCI} = \sum_{\mathrm{i}=0}^{\mathrm{m}} (\text{Type of constraint})_{\mathrm{i}} \times \sum_{\mathrm{j}=0}^{\mathrm{n}} (\text{Constraints count})_{\mathrm{ij}}.$$

Further, in case of blind/no constraints, the value of DCI will be equal to zero.

3.1.8 Interface complexity (IFC)

This complexity describes the external integration/interfaces that are associated with the proposed software. These integrations could be hardware, software, network and communication based for the deployment of the proposed software. IFC is expressed as:

$$IFC = \sum_{i=0}^{n} (External\ interface)_i \times \sum_{j=0}^{m} (Externjal\ interface\ count)_{ij}.$$

3.1.9 Improved requirement based complexity

Finally, the IRBC is calculated by combining all the sub complexities on the basis of their relevance and relative contributions for the computation of overall complexity for any physical system or software to be developed. This is mathematically expressed as:

$$\mathbf{IRBC} = ((\mathbf{PC} \times \mathbf{PCA}) + \mathbf{DCI} + \mathbf{IFC}).$$

The IRBC of any physical system primarily depends on the functionality that is derived from customer's requirement and depends on developer's capability. Due to this reason, the PC is multiplied with PCA in order to get a basic estimate of overall complexity. Further, in order to provide better estimate, there are also other contributing complexity measures like DCI and IFC that also have a very specific and significant contribution towards the computation of final complexity measure. These contributing complexity measures are dependent on the development process, environment and external integrations.

The next section illustrates the proposed IRBC measure with the help of a case study and also compared with other established complexity measures.

3.2 Results

The proposed IRBC measure is derived from the software requirement document written in IEEE-830:1998 standard for the generation of SRS. This section discusses the result of applying IRBC on 15 SRS's and generating their source code too. Further the result of the proposed IRBC measure is compared with other established code, cognitive value and OO metric based measures. The result of applying various complexity measures on fifteen problems is illustrated in Tables 7 and 8.

Figure 2 shows the plot between IRBC and other established program based complexity measures i.e. code based (like Halstead and Mc Cabe) and cognitive complexity based (like KLCID, CFS and CICM). It can be observed from the plot that the proposed IRBC measure values, follow the similar trend in increasing and decreasing order with other existing approaches. KLCID has very small program complexity value due to the fact that, it is based on only number of

identifiers and LOC. Higher values of CFS and CICM are there due to consideration of control structures and LOC.

However, for the rest of the cases, the trend is exactly matching with the code as well as cognitive based complexity measures. On the other hand, the IRBC values are higher for the programs, that have higher functionality to be performed and more quality attributes to be retained. For more prominent statistical analysis, IRBC is also individually compared with a group of code and cognitive complexity value based measures.

To move a step closer, the proposed IRBC measure is also compared with OO metric [20] i.e. NCBC and exception handling functions (EHF) as shown in the Table 8.

Since these metrics consider the code to be written in OO language. Hence, for the sake of comparison five programs written in OO language are considered and their SRS's generated. Further, it is observed that IRBC follows exact trend with these measure too.

3.3 Comparison IRBC and other complexity measures based on Weyuker's properties

This section presents a comparison between proposed IRBC measure and other established complexity measures in terms of all nine Weyuker's properties [7] as shown in Table 9. Weyuker's [37] properties play an important role in evaluating and validating software complexity measure as well as its robustness. Weyuker proposed nine properties to evaluate any software complexity measure.

Property # 1 $(\exists P)\ (\exists Q)\ (|P| \neq |Q|)$ *Where P and Q are program body*

This property states that a measure should not rank all programs as equally complex.

Property # 2 *Let c be a non-negative number. Then there are only finitely many programs of complexity c.*

This property states that the measure should give complexity value by a non-negative number.

Property # 3 *There are distinct programs P and Q such that* $|P| = |Q|$.

This property states that the measure should neither rate too many programs as being of equal complexity nor assign every program a unique complexity.

Property # 4 $(\exists P)\ (\exists Q)\ (P \equiv Q \text{ and } |P| \neq |Q|)$

This property states that the measure should not consider the function being computed by a program for complexity calculations, but instead it should consider the implementation details i.e. a measure should be implementation dependent.

Table 8 Comparison between IRBC and object oriented complexity measure

S. no.	Program	EHF	NCBC	IRBC
1	Program to sort an array	33.3	33.3	21.06
2	Program for temperature conversion	50	50	24.57
3	Program for number format exception	100	100	28.08
4	To open and read a file	42.5	42.5	23.4
5	Demonstrate exception handling in C++	50	50	24.57

Table 7 Comparison between IRBC and other established complexity measure

S. no.	Programs	Halstead	Mc Cabe	CICM	CFS	KLCID	IRBC
1	Bit stuffing	57.73	9	329.03	183	1.43	35.1
2	Addition of matrix	65.45	9	58.20	365	.34	35.1
3	Add and Mul. of matrix	108.5	19	624.9	1630	.65	46.8
4	FCFS	29.33	3	17.11	28	.84	29.25
5	Fibonacci	36.25	5	195.6	50	1.17	24.57
6	LZW	67.09	6	643.26	412	.63	28.08
7	Round robin	102	8	232.26	360	.56	35.1
8	Sum and average	21.87	3	14.53	12	.8	28.08
9	Prime no.	33	5	79.45	50	.95	28.08
10	Semaphore	77.33	6	217.44	294	1.027	42.12
11	Bubble sort	62	6	260.48	120	.68	28.08
12	Palindrome	33.25	3	17.95	12	.809	21.06
13	Calculator	49.33	10	65.93	44	.965	70.2
14	RSA	85.76	8	123.96	306	.65	77.22
15	Linear Search	46.44	4	41.99	60	.66	56.16

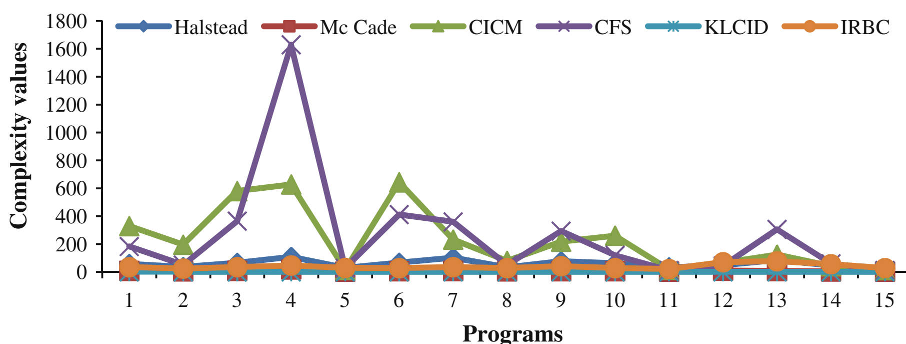


Fig. 2 IRBC versus other established measures

Table 9 Comparison of IRBC with established measures in terms of Weyuker's Property

Prop. no.	SC	CN	EM	DC	CCM	CICM	IRBC
1	Y	Y	Y	Y	Y	Y	Y
2	Y	N	Y	N	Y	Y	Y
3	Y	Y	Y	Y	Y	Y	Y
4	Y	Y	Y	Y	Y	Y	Y
5	Y	Y	N	N	Y	Y	Y
6	N	N	Y	Y	N	Y	Y
7	N	N	N	Y	Y	Y	NA
8	Y	Y	Y	Y	Y	Y	Y
9	N	N	Y	Y	Y	Y	Y

SC statement count, *CN* cyclomatic number, *EM* effort measure, *DC* dataflow complexity, *CCM* cognitive complexity measure, *CICM* cognitive information complexity measure, *Y* yes, *N* no, *NA* not applicable

Property # 5 $(\forall P)(\forall Q)(|P| \leq |P;Q|$ *and* $|Q| \leq |P;Q|)$

This property states that the measure should consider the interaction among the modules of a program after their concatenation.

Property # 6-a $(\exists P)$ $(\exists Q)(\exists R)(|P| = |Q|)$ *and* $(|P;R| \neq |Q;R|)$

This property states that the measure should not rate the components of a program as more complex than the program itself.

Property # 6-b $(\exists P)$ $(\exists Q)(\exists R)(|P| = |Q|)$ *and* $(|R;P| \neq |R:Q|)$

This property states that the measure should not rate the components of a program as more complex than the program itself.

Property # 7 *There are program bodies P and Q such that Q is formed by permuting the order of the statement of P and* $(|P| \neq |Q|)$.

This property states that the measure should be responsive to the order of the statements and hence the potential interaction among the statements.

Property # 8 *If P is renaming of Q, then* $|P| = |Q|$.

This property states that changing the program or variable name will not affect the functionality of the proposed software.

Property # 9 $(\exists P)$ $(\exists Q)(|P| + |Q|) < (|P;Q|)$ *OR* $(\exists P)$ $(\exists Q)$ $(\exists R)(|P| + |Q| + |R|) < (|P;Q;R|)$

This property states that the measure should not rate the complexity of a program as less than the sum of the complexities of its components.

These properties also evaluate weaknesses of a measure in a concrete way and in turn lead to the definition of really good notion of software complexity. With the help of these properties, one is able to determine the most suitable measure among the different available complexity measures.

As can be seen from the Table 9, IRBC satisfies 8 out of 9 Weyuker's properties.

However 7th property is not applicable to the proposed measure because of dependency on code. Hence, Weyuker's 7th property is not applicable to IRBC.

4 Estimation of software testing effort using requirement based complexity

This section discusses the application of IRBC for the estimation of software testing effort for software to be developed. Since practitioners are generally short of time and resources, they tend to evade systematic testing as this is not considered to be so very lucrative job. This affects the whole of the software development because every type of testing technique demands adequate test case generation, modeling and documentation. Though many software testing measures have been proposed in the past research, but still it is far from being matured. If we go by the quality standards, then the estimation of the software testing effort has to be done before the tester(s) start writing the test cases. The proposed test effort measure empirically estimates the software testing effort for software to be developed from IRBC which has been obtained from software requirements in order to

properly plan the testing process, reduce the testing cost, and computation of software testing effort in early phases of SDLC.

4.1 Computation procedure

In order to clearly understand the approach, the following paragraphs discuss about the estimation of software testing effort from IRBC measure.

4.1.1 Requirement based test function points (RBTFP)

IRBC comprises of all the attributes that are sufficient to obtain function point analysis (FPA) [37] for any software. The function point measure [37] consists of five attributes i.e. external input, external output, interfaces, file and enquiry. However IRBC encompasses an exhaustive set of quantified parameters like IOC consisting of IC, OC for the quantification of input output information, entering or leaving the proposed system and SC for the quantification of local and remote storage for the proposed system. IRBC uses IFC for computation of external interfaces and User Class Complexity (UCC) for quantification of user classes. In addition IRBC uses FR, NFR, PC, RC, DCI and SFC for the computation of RBTFP. These parameters serve a basis for the estimation of RBTFP that have not been considered in FPA. However, IRBC makes use of an exhaustive set of quantified parameters for its computation in order to compute the Requirement Based Function Point (RBTFP) measure. Therefore, in order to estimate RBTFP, we require technical and environmental factors (TEFs) [43, 44] pertaining to the testing activity. The details of nine different TEFs are illustrated in Table 10 along with their assigned weights.

These parameters have significant contribution to the software testing activity in order to obtain RBTFP of the proposed software. The need and applicability of these factors are calculated on the basis of the value of perceived impact or degree of influence (DI) of the above nine factors that ranges from zero (harmless) to four (essential). TEF is computed by summing the score of nine different factors on the basis of DI value that ranges from zero to four. Harmless = 0, Needless = 1, Desirable = 2, Necessary = 3, Essential = 4.

Table 10 Technical and environmental factors

Factors	Description	Assigned value
1	Test tools	5
2	Documented inputs	5
3	Development environment	2
4	Test environment	3
5	Test suite reuse	3
6	Distributed system	4
7	Performance objectives	2
8	Security features	4
9	Complex interfaces	5

Mathematically, TEF is represented [43, 44] as:

$$\mathrm{TEF} = 0.65 + 0.01 \times \sum \mathrm{Fi}.$$

IRBC has bearing on two basic parameters i.e. functionality and input. These parameters are sufficient to compute the test case requirement for the proposed software in both black box and white box scenarios. The functionality and input parameters are also supplemented by nine TEF, which is specifically meant for the testing purpose only. We propose the RBTFP as a product of IRBC and TEF that is expressed as:

$$\mathrm{RBTFP} = \mathrm{IRBC} \times \mathrm{TEF}.$$

RBTFP acts as basis for finding out the optimal number of test cases, which is required to carry out an exhaustive testing for software to be developed. The next sub section discusses the details of computation of number of requirement based test cases (NRBTC).

4.1.2 Number of requirement based test cases

NRBTC is a function of RBTFP. Because numbers of function point dictate the number of test cases that is to be designed [41]. This is expressed as:

$$\mathrm{NRBTC} = (\mathrm{RBTFP})^{1.2}.$$

The estimation of test case requirement for the proposed software serves as basis for the estimation of test effort in man hours.

4.1.3 Requirement based test team productivity (RBTTP)

Test team productivity depends on the number of staff and personnel skill available to test the software. A model [56] for the estimation of tester rank with two different dimensions i.e. experience in testing and knowledge of target application is illustrated in Fig. 3. Tester rank helps in understanding tester behavior for test execution. Higher the rank of test team, lower is the need of number of tester. Hence, in order to derive the RBTTP, the number of testers and their relative rank is considered which can be expressed as:

$$RBTTP = \sum_{i=1}^{n} T_i \times \sum_{j=1}^{4} R_{ij},$$

where T is total number of testers and R is the respective rank of the tester from tester rank model.

Having obtained the value of number of requirement based test cases (NRBTC) and requirement based test team

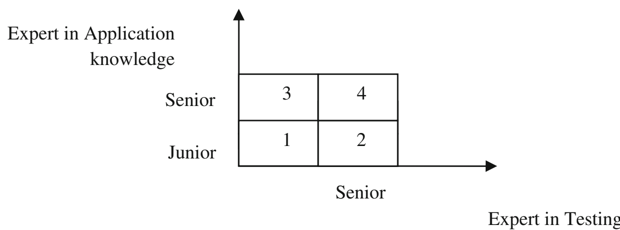


Fig. 3 Tester rank model

productivity (RBTTP), we compute the requirement based test effort estimation (RBTEE) in man-hours in next section.

4.1.4 Requirement based test effort estimation

In order to compute software testing effort for the proposed software, it is necessary to know the two significant parameters, first, the prior knowledge of number of test cases and second, the productivity of the test team. In this direction, we have already derived the contributing measures i.e. NRBTC, for the computation of number of test case and RBTTP, for the estimation of test team productivity in previous sub sections. Since higher productivity of test team(s) reduces the amount of required testing effort hence, a ratio of NRBTC and RBTTP is taken in order to get the final requirement based test effort estimate (RBTEE) in man-hours for the proposed software, this is expressed as:

$$\text{RBTEE} = \text{NRBTC}/\text{RBTTP} \quad (\text{man-hours}).$$

Early estimation of software testing effort using IRBC will save tremendous amount of time, cost and man power for software to be developed.

4.2 Results and validation

This section categorically compares the proposed RBTEE measure with various other established test effort estimation measures proposed in the past. To analyze the validity of the proposed measure, fifteen SRS's of various problem statements have been considered. In order to compare the proposed test effort measure with code based measures, the source code of all the fifteen problems is also generated. The comparison strictly considers various categories of test effort estimation measures like use case based, complexity value based and code & execution points based. Use case based test effort estimation methods are further classified into two broad categories i.e. UCP based method, and scenario based method. The measure based on use case include the parameters like actor weight, use case weight, conversion factors etc., for the estimation of test effort. The most important contributing factors such as input, output, interfaces, storage, FR and most importantly NFRs are not taken into consideration in the established testing measures. However, there are two other categories also for test effort estimation i.e. code based and complexity value based. Code based test effort estimation measures uses the LOC, computes the execution points and generates the test cases using a constant to compute test effort measure. However the complexity value based test effort estimation measures use LOC, control structure and finally correlates the test effort with complexity of the code. These two categories of approaches can work only when the code of the software is available.

Table 11 shows the results of various test effort estimation measures applied on fifteen problems.

Table 11 Comparison between proposed test effort estimation v/s other established test effort estimation

S. no.	Program	Use case (man-h)	Test spec (man-h)	Scenario based (man-h)	Test exe effort (man-h)	Exp. based approach	CICM (comp. unit)	Test effort (man-h)	RBTEE (man-h)
1	Bit stuffing	277.4	182	23.5	598	74.6	329.034	44.99	67.24
2	Matrix addition	273.6	161	22.2	529	72.02	580.20	43.68	67.24
3	Matrix add multiply	410.4	273	33.66	897	156.4	624.925	78.11	94.96
4	FCFS	241.4	133	15.98	437	171.5	17.11	90.04	54.02
5	Fibonacci	244.8	119	18.20	391	61.67	195.6	38.39	43.82
6	LZW	241.4	140	15.98	460	186.74	643.26	96.66	59.33
7	Round robin	273.6	161	22.64	529	243.28	232.26	120.4	67.24
8	Sum & average	244.8	147	18.20	483	25.74	14.5	18.53	51.44
9	Prime number	269.8	154	19.18	506	88.01	79.45	51.64	51.44
10	Semaphore	326.6	217	25.75	713	115.74	217.44	64.88	83.6
11	Bubble sort	265.2	154	18.25	506	72.02	260.48	43.67	51.44
12	Palindrome	277.4	175	23.53	575	61.17	17.91	38.38	36.42
13	Calculator	281.8	189	25.58	828	66.81	65.93	41.04	154.48
14	RSA	334.4	217	26.31	713	162.46	123.96	86.06	199.77
15	Linear search	296.4	175	25.27	575	96.91	41.99	55.61	118.19

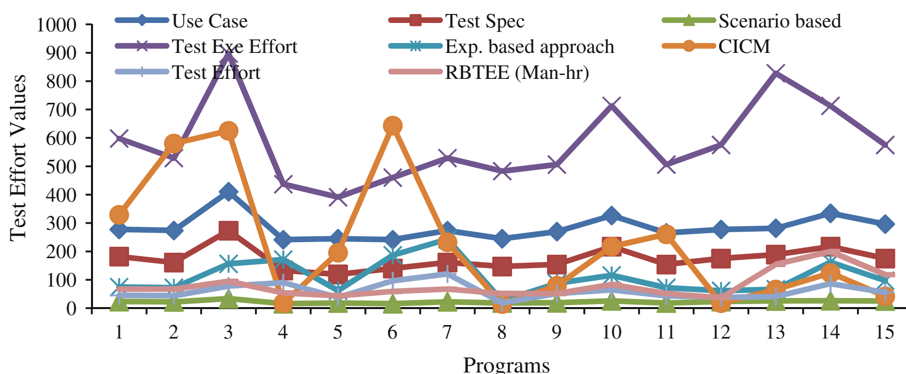


Fig. 4 Plot between RBTEE v/s other test effort estimation measures

Figure 4 shows the comparison between various categories of test effort estimation measures with the proposed measure. As it is seen from the plot of Fig. 4 that, program based complexity measures carries higher value of test effort than other approaches, because of the difference in coding style by different programmers. The calculation of these measures also depends on the number of statements in the program, their subsequent control structures and weighted count which makes the value of test effort higher in comparison of other approaches.

Further, code and execution point based approaches carry higher values than the proposed measure and lower values than the complexity based measures because of the fact that, these approaches are based on the execution points. The execution points depend on the variables used in the program. Higher number of variables carries higher execution points, and hence higher value of test effort (in man hours) obtained.

The use case based measures have lower values of test effort (in man-hours) than code and execution point based measures because the values of test effort for these approaches are derived upon modeling the use case at requirement level. Also the variations observed in use case based measures are due to the difference in conversion factor i.e. 20 for UCP and 3 for scenario based UCP. The co-relation that is observed with use case based approaches and the proposed approach illustrate that:

- The UCP based approach is close to five times of proposed measure.
- The Scenario based approach is close to one-third of proposed measure.

4.3 Result analysis for larger programs

Having established the proposed measure for average size of programs, verification for its validity was carried out for larger programs approximating to about 1–2 KLOC. The results are illustrated in Table 12. Again, it is seen that the proposed measure is able to generate realistic estimates comparable with FPA, UCP and Scenario based UCP measures.

4.4 Result analysis for real life software industry projects

In order to prove the applicability of proposed RBTEE measure in software development industry, four large SRS of software projects of different areas has been used to compare and reaffirm the results. The SRS's of these projects resulted in software with more than 10 KLOC as shown in Table 13. Based on IEEE SRS document of these projects, we have computed IRBC and NRBTC and RBTEE for all these four projects. It is observed that the value obtained from the proposed NRBTC measure is very

Table 12 Tabular comparison for larger programs with proposed RBDEE measure

S. no.	Program name	FPA	Use case	Scenario	IRBC	RBTEE
1	Tower of Hanoi	13.43	190	8.55	10.53	15.85
2	Insert B-tree	15.8	304	12.54	18.72	31.63
3	Prim's algorithm	26.86	323	15.39	46.8	94.96
4	Kruskal's algorithm	33.97	418	18.24	58.5	124.12
5	Threaded binary tree	27.65	513	26.22	74.88	166.91
6	8-Queens	24.49	513	28.5	81.9	185.86
7	B-tree	21.33	513	33.63	84.24	186.79
8	Knapsack problem	35.55	608	36.48	90.09	88.39
9	AVL tree	37.13	608	37.62	102.96	244.6
10	Tic tac toe	24.49	608	38.19	114.66	278.33

Table 13 Details of lines of code for industry projects

Industry project	KLOC
1	1.05
2	10.16
3	12.58
4	13.24

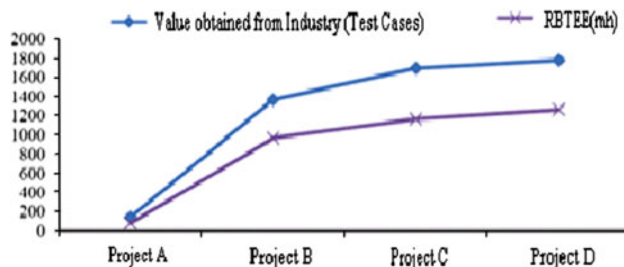


Fig. 5 Plot between NRBTC and industry values for test cases

close to the value of test cases obtained by Software Company for all these projects. The details of the value obtained from NRBTC and value of test cases received from Software Company is illustrated Fig. 5.

5 Strength of the proposed measure

FPA considers only five attributes i.e. external input, external output, internal logical file, external interface, and external inquiry at abstract level. It is performed after creation of design specifications. Also the method is quite time-consuming thus might result costly and the evaluation is basically subjective. It makes it difficult to be applied coherently and repeatedly. Further, it is not appropriate to sizeable software of other types, including real-time, scientific and embedded software. It needs subjective evaluations with a lot of expert judgment involved. However the proposed measure is drawn from the requirement based complexity that uses an exhaustive set of objective attributes along with their precise quantification. IRBC is computed immediately after freezing the requirements in requirement analysis phase of SDLC. Even the naïve developer can estimate the development and testing effort from IRBC unambiguously.

Further the Use Case Based Measures depends on the prudence of system analyst and it requires expert's intervention. However the proposed measure is drawn from SRS and considers IRBC measure for precise estimation. Use Case based measures are categorized into simple, average and complex with multiplicative values of 5, 10, 15 respectively. But, these values for a particular software type are not properly quantified and, next, these values don't converge with the size of the proposed software. Also, the actor classification is done as simple, average and complex with values 1, 2 and 3 respectively. But the type of user with privileges is not mentioned. However, proposed measure is quantified for every parameter and considers UCC for this consideration. Also, Use Case based measures computes UCPs only and estimates the testing effort on the basis of a conversion factor in man hour. However the proposed measures empirically estimates all the parameters required to compute the software testing effort in person - hour for the proposed software. Use cases are not well suited to capturing non-interaction based requirements of a system (such as algorithm or mathematical requirements) or NFRs (such as platform, performance, timing, or safety–critical aspects). However these requirements are taken into consideration in the proposed testing effort measure using ISO-9126 model. UCP does not automatically ensure clarity. Use cases are complex to write and to understand, for both end users and developers. The proposed measure is clear and understandable due to its derivation from SRS based IRBC. As there are no fully standard definitions of use cases, each project must form its own interpretation. Use case developers often find it difficult to determine the level of user interface dependency to incorporate in a use case. The proposed measure quantifies external interfaces on the basis of IFC.

6 Conclusion

The work presented in this paper makes an attempt to establish the metrics for estimation of software development activities like computation of software complexity and estimation of software testing effort using SRS of the proposed software.

At the onset, a measure for early estimation of software complexity based on SRS is proposed. The measure computes the complexity of software to be developed from the SRS document, because an SRS contains all the characteristics, content and functionality of the software to be developed. The results obtained are compared with control flow based, code based, cognitive complexity and OO metric based measures. It is seen that the proposed IRBC measure is comparable and closely follows the trend of other established complexity based measures. Since, IRBC computes the software complexity at a very early stage of the software development; hence it is able outperforms other comparable measures. The robustness of the proposed measure is evaluated and established against Weyuker properties.

Later, the paper proposes RBTEE metric on the basis of IRBC of proposed software. From the result and validation, it is observed that the proposed test effort measure follows the trend set by other established measures in a comprehensive fashion. Also, the values obtained through RBTEE are a approximate mean of use case based measures. This provides

an aid to the developer and practitioner in reducing rework by delivering maximum coverage with minimum number of test cases for improving the test effectiveness. The approach presented here is also validated with realistic results pertaining to large projects of software development organization to ascertain validity of the proposed measures as compared with the conventional code based approaches. This will enable the developers and analysts to carry out effective, planned and systematic software development in requirement analysis phase of software development life cycle.

Appendix

1. *Introduction*

Scheduling is central to operating system design. Different algorithms are there to decide which process in ready queue is to be allocated to CPU. CPU scheduling algorithms can be categorized based on pre-emptive and non pre-emptive scheduling algorithms. One of the CPU scheduling algorithms is First Come First Serve (FCFS) scheduling algorithm with this scheme, the process that requests the CPU first is allocated the CPU first.

2. *Purpose*

The main purpose of FCFS algorithm is to increase the CPU utilization, increase throughput, reduce waiting time and response time. By this algorithm we can achieve fairness which can be reflected by treating all the processes same. We can reduce overheads to improve the overall performance of the system.

3. *Scope*

The major scope of FCFS algorithm is in the batch systems. Waiting time can be large if short request wait behind the long once therefore FCFS is used in the case where the burst time of the processes are comparatively less and are in ascending order.

4. *Definitions*

FCFS can be defined as a scheme in which the process that requests the CPU first is allocated the CPU first. When a process enters the ready queue it's process control block (PCB) is linked on to the tail of the queue when the CPU is free it is allocated in the process at the head of the queue. The running process is then removed from the queue.

Throughput is defined as the number of processes that are completed per unit time.

Turnaround Time is the interval from the time of submission of the processes to the time of its completion.

Waiting time can be defined as the sum of periods spent waiting in the ready queue.

Response Time is the amount of time, it takes to start responding, but not the time that it takes to output that response.

5. *References*

Peter Baer Galvin, Abraham Silberschatz, Introduction to Operating System, Fourth Edition.

6. *Overview*

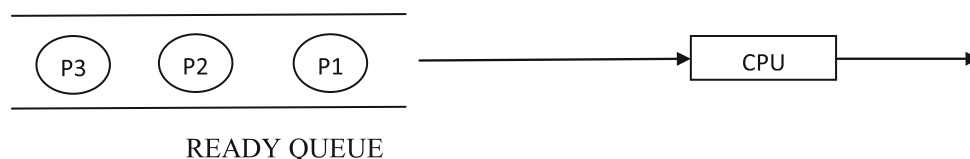
By far the simplest CPU scheduling algorithm is the FCFS scheduling algorithm. This algorithm executes the requests that arrives first. The implementation of the FCFS policy is easily managed with the FIFO queue. The average waiting time under this policy however is quite long. FCFS Scheduling algorithm is non-preemptive. It would be disastrous to allow one process to keep the CPU for an extended period.

7. *Overall description*

- Process that requests the CPU first shall be allocated the CPU first.
- Implementation of FCFS is managed through a FIFO queue.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue.
- When the CPU is free it is allocated to the process at the head of the queue.
- On execution of the running process it is then removed from the queue.
- The average waiting time under the FCFS policy is often quite long.
- FCFS scheduling is non-preemptive.
- Once the CPU has been given to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting Input Output.
- The FCFS algorithm is particularly troublesome for time sharing systems.
- Each user needs to get a share of CPU at regular intervals.

8. *Product perspective*

A diagram describing the overall FCFS perspective is shown below



9. *Product functions*

Inputs

- Number of Processes
- Burst Time of Processes

Computation

- Calculation of Waiting Time of every Process
- Calculation of Turn Around Time of every Process

Outputs

- Display of Waiting Time
- Display of Turn Around Time

10. *User characteristics*

For Casual end users to provide input and output

11. *Constraints*

- FCFS is non-preemptive in nature.
- FCFS is particularly troublesome for time-sharing system.
- No process should be allowed to keep the CPU for an extended period.
- Proper mix of jobs is needed to have good results from FCFS Scheduling.

References

1. Costello RJ, Liu DB (1995) Metrics for requirement. Eng J Syst Softw 29:39–63. Elsevier Science, North Holland, pp 39–63
2. The Standish group research for staggering bugs and effort, http://standishgroup.com
3. IEEE Computer Society (1998) IEEE recommended practice for software requirement specifications. IEEE Inc, NY, pp 1–37
4. Mc Cabe TJ (1976) A complexity measure. IEEE Trans Softw Eng SE-2(4), pp 308–320
5. Chaudhary BD, Sahasrabuddne HV (1980) Meaningfulness as a factor of program complexity ACM transaction ACM – 0-89791-028-1/80/1000/0457, pp 457–466
6. Hamer PG, Frewln GD (1982) M.H. Halstead's software science - a critical examination. IEEE Trans Softw Eng 0270-5257/82/0197, pp 197–206
7. Weyuker EJ (1988) Evaluating software complexity measure. IEEE Trans Softw Eng 14(9):1357–1365
8. Davis JS, LeBlanc RJ (1988) A study of applicability of complexity measures. IEEE Trans Softw Eng 14(9):1366–1372
9. Zweben S, Gourlay J (1989) On the adequacy of Weyuker's test data adequacy axioms. IEEE Trans Softw Eng 15(4):496–500
10. Chariniavsky JC, Smith CH (1991) On Weyuker's axioms for software complexity measures. IEEE Trans Softw Eng 17(6):636–638
11. Halstead MH (1977) Elements of software science. Elsevier, New York
12. Yu S, Zhou S (2010) A survey on metric of software complexity. In: Proceeding of 2nd IEEE international conference on information management and engineering (ICIME), pp 352–356
13. Morozoff EP (2010) Using a line of code metric to understand software rework IEEE-software. IEEE Comput Soc 27(1):72–77
14. Harrision W (1992) An entropy based measure of software complexity. IEEE Trans Softw Eng 18(11):1025–1029
15. Tian J, Zelkowitz M (1995) Complexity measure evaluation and selection. IEEE Trans Softw Eng 21(8):641–650
16. Shao J, Wang Y (2003) A new measure of software complexity based on cognitive weights. Can J Comput Eng 28(2):69–74
17. Gray WD, Scholes MJ, Sims C (2005) Cognitive metrics profiling. J Human Factor Ergonomics 5:1144–1148
18. Kushwaha DS, Misra AK (2006) Evaluating cognitive information complexity measure. In: Proceedings of 13th annual IEEE international symposium and workshop on engineering of computer based System (ECBS'06), pp 502–504
19. Auprasert B, Limpiyakorn Y (2009) Towards structured software cognitive complexity measurement with granular computing strategy. In: Proceeding of 8th IEEE international on cognitive informatics (ICCI-09), pp 365–370
20. Aggarwal KK, Singh Y, Kaur A, Melhorta R (2006) Software design metric for object oriented software. J Object Technol 6(1):121–138
21. Chhabra JK, Gupta V (2009) Evaluation of object-oriented spatial complexity measures. ACM Sig Soft Softw Eng 34(3):1–5
22. Baccarini D (1996) The concept of project complexity. Int J Project Manag 14(4):201–204
23. Klemola T, Rilling J (2003) A Cognitive Complexity Metric Based on Category Learning. IEEE International Conference on Cognitive Informatics, pp 106–112
24. Noble D, Letsky M (2003) Cognitive based metrics to evaluate collaboration effectiveness RTO-symposium, pp 1–14
25. Singh Y, Sabharwal S, Sood M (2004) A systematic approach to measure the problem complexity of SRS of an Information System. Int J Inf Manag Sci 15(1):69–90
26. Din CY (2007) Requirement content goodness and complexity measurement based on NP chunks. J Syst Cybern Inf 15(1):69–90
27. de Boer RC, van Vliet H (2009) On the similarity between requirements and architecture. J Syst Softw 82(3):544–550
28. Kanjilal A, Sengupta S, Bhattacharya S (2009) Analysis of complexity of requirements: a metric based approach, ACM-ISEC'09, February 23–26, 2009, India, pp 131–137
29. Fitsilis P (2009) Measuring the complexity of software projects. In: 2009 World congress on computer science and information engineering, pp 644–646
30. Tai KC (1980) Program testing complexity and test criteria. IEEE Trans Softw Eng SE-6(6), pp 531–536
31. Ryser J, Bernaer S, Glinz M (1999) On the state of art in requirements-based validation and test of software. University of Zurich, Zurich
32. Aurum A, Peterssson H, Wohlin C (2002) State-of-the-art: software inspection after 25 years. J Softw Test Verification Reliab 12(3):133–154
33. Holden I, Dalton D (2006) Improving test efficiency using cumulative test analysis. In: Proceedings of the testing: academic and industrial conference—practice and research techniques (TAIC-PART'06), pp 152–158
34. Bertilino A (2007) Software testing research: achievements, challenges and dreams. IEEE Future Softw Eng FOSE, pp 85–103
35. Aranha E, Borba P (2007) Test effort estimation model based on test specifications. In: Testing: academic and industrial conference—practice and research techniques, IEEE Computer Society, 2007, pp 67–71
36. Veenendal E, Deckkers T In: Kusters R, Cowderoy A, Heemstra e Erik van F (eds) Test point analysis: a method for test estimation in project control for software quality. Shaker publishing.
37. Whalen MW, Rajan A, Heimdahal MPE, Miller SP (2006) Coverage metrics for requirements-based testing, ISTA 06, July 17–20, 2006 Portland Maine, USA, pp 161–170

38. Rajan A, Whalen MW, Heimdahl MPE (2007) Model validation using automatically generated requirements-based test. In: 10th IEEE-high assurance systems engineering symposium, pp 95–104
39. Sneed HM (2007) Testing against natural language requirements. In: 7th international conference on quality software (QSIC-2007) CSIT, pp 380–387
40. Uusitalo EJ, Komssi M, Kauppinen M, Davis AM (2008) Linking requirement and testing in practice. In: 16th IEEE international requirement engineering conference, IEEE-Computer Society, pp 265–270
41. Yi Q, Bo Z, Xiaochun Z (2008) Early estimate the size of test suites from use cases. In: 15th Asia-Pacific software engineering conference, IEEE Computer Society, pp 487–492
42. Ramachandran M (1996) Requirements-driven software test: a process oriented approach. ACM Sigsoft Softw Eng Notes 21(4): 66–70
43. Nageshwaran S (2001) Test effort estimation using USE CASE points, Quality Week 2001, San Francisco, California USA, 2001, pp 122–126
44. de Almeida ERC, de Abreu BT, Moraes R (2009) An alternative approach to test effort estimation based on use case. In: IEEE-international conference on software testing verification and validation, pp 279–288
45. Vaysburg B, Tahat LH, Korel B (2002) Dependence analysis in reduction of requirement based test suites. ACM ISSTA–02, pp 107–111
46. Kim HY, Sheldon FT (2004) Testing software requirement with Z and statecharts applied to an embedded control system. Softw Qual J 12:231–264
47. Aranha E, de Almeida F, Diniz T, Fontes V, Borba P (2007) Automated test execution effort estimation based on functional test specification. In: Proceedings of testing: academic and industrial conference practice and research techniques, MUTATION 07, pp 67–71
48. Cabral G, Tamai T (2008) Requirement based testing through formal methods, In: Proceedings of Testing of Communicating Systems (TESTCOM) and Formal approaches to testing of Software (FATES), Tokyo, 10–13 June, 2008
49. Guerreiro e Silva D, de Abreu BT, Jino M (2009) A simple approach for estimation of execution of function test case. In: IEEE-international conference on software testing verification and validation, pp 289–298
50. Lokan C, Mendes E (2009) Investigating the use of chronological split for software effort estimation. IET Softw 3(5):422–434
51. Kushwaha DS, Misra AK (2008) Software test effort estimation. ACM Sigsoft Softw Eng Notes 33(3):1–5
52. Boehm BW (1981) Software engineering economics. Prentice Hall, Englewood Cliffs
53. Boehm B, Horowitz E, Madachy R, Clark B, Westland C, Selby R (1995) Cost models for future software lifecycle process: COCOMO 2.0, IEEE computer society, pp 1–31
54. Periyasamy K, Liu X (1999) A new metric set for evaluating testing efforts for object oriented programs. In: Proceedings of technology of object oriented languages and systems 1999, pp 84–93
55. Dromey RG (1995) A model for software product quality. IEEE Trans Softw Eng 21:146–162
56. Xiaochun Z, Bo Z, Fan W, Yi Q, Lu C (2008) Estimate test execution effort at an early stage: an empirical study. In: International conference on cyber world, IEEE Computer Society, pp 195–200