World Academy of Science, Engineering and Technology
International Journal of Computer and Systems Engineering
Vol:10, No:4, 2016

Suitability of Black Box Approaches for the Reliability Assessment of Component-Based Software

Anjushi Verma, Tirthankar Gayen

***Abstract*—**Although, reliability is an important attribute of quality, especially for mission critical systems, yet, there does not exist any versatile model even today for the reliability assessment of component-based software. The existing Black Box models are found to make various assumptions which may not always be realistic and may be quite contrary to the actual behaviour of software. They focus on observing the manner in which the system behaves without considering the structure of the system, the components composing the system, their interconnections, dependencies, usage frequencies, etc. As a result, the entropy (uncertainty) in assessment using these models is much high. Though, there are some models based on operation profile yet sometimes it becomes extremely difficult to obtain the exact operation profile concerned with a given operation. This paper discusses the drawbacks, deficiencies and limitations of Black Box approaches from the perspective of various authors and finally proposes a conceptual model for the reliability assessment of software.

***Keywords*—**Black Box, faults, failure, software reliability.

I. Introduction

ERRORS may cause fault and faults may cause the failure of the system. Yet, it is not necessary that all errors may cause faults and all faults may lead to failures. There may be some errors which may not cause fault and some faults which may never cause a failure (as shown in Fig. 1). Reliability may be viewed a user-oriented factor [8]. If the user hardly experiences failure, then the system is taken to be more reliable than the system failing more often. According to some dictionaries, a software usually corresponds to the routines, programs, symbolic language, or data which control the functioning and directs the operation of hardware. Since, it is known that software cannot be touched, has no weight, no material, or energy but when it is executed properly it performs a specific task. Many books define software as *a group of programs which perform a specific task*. The program is usually defined as the *set of instructions to perform a specific task*. Hence, software is considered to be an *abstract-ware*. Hence, a software process is purely a logical process. Therefore, predicting the reliability of software is predicting the reliability of this *abstract-ware*. In accordance with ANSI, software reliability corresponds to "*the probability of failure-free software operation for a specified period of time in a specified environment*." [9], [10]. According to Hoang Pham [45], [46], in software, failures in software are caused by incorrect input data, incorrect logic or incorrect statements (*without considering the hardware or operating environment dependencies*) and in hardware, the failures are caused by environmental factors, design errors, misuse, random failures and material deterioration.

A Verma is with the JNU, India (e-mail: avbinni56@rediffmail.com).

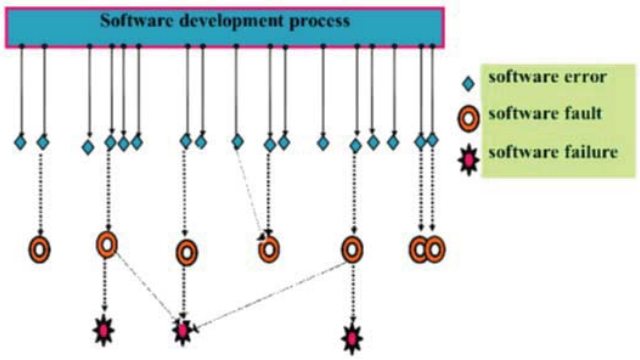

Fig. 1 Software errors, faults and failures

II. Issues from the Literature

Various models like time between failure models, failure count model, etc. focus on observing the way in which the system behaves irrespective of the structure of the system. These models treat the software as a black box without considering the architecture, the components (composing the system), their interconnections, their dependencies and their usage frequencies. These block box models may not be suitable to find out the set of all executable paths to estimate the reliability since they do not consider the inner structure of the system and hence may not be suitable for the reliability estimation of large and complex component based software systems. The failure data used in many of these black box models are inconsistent with today's multitasking operating environment with varying system loads. Therefore, architecture based models which consider the behavior of the system based upon the usage of various portions of the code, relating the system or application reliability to the reliabilities of individual components and system structure are needed. Software reliability assessment is an action of estimating the reliability of software. In accordance with [4], it can be estimated by two ways as follows:

- ***System level reliability:*** In it, the whole software system is considered as a single unit and then reliability is calculated. Here unit testing is used. But, it may not be very suitable for component based system as it ignores the compositional properties of components.
- ***Component level reliability:*** In it, the reliability of individual component is calculated and then these individual components' reliability is used to calculate the whole system reliability. Here, integration testing is used

and may be more suitable for component based software systems.

Littlewood [26], [27] expressed that certain kind of assumptions which have been made seemed to be very naive, like for example when the software failure rate or crash rate is proportional to the remaining number of errors in the software. He stated that he could not find the validity of this assumption in the programs which he had seen. Since, one can easily imagine a scenario in which a program with two bugs in little exercised portions of code is considered to be more reliable than a program containing only one and frequently encountered bug. He queried the reason for which the failure times of a program cannot be predicted exactly. If one knows the way in which the program behaved for every conceivable input, and could predict future inputs, then one can suppose that it would be possible to predict the next failure epoch. Unfortunately, one never has such a total knowledge. Most authors are of the opinion that the failure process is random. But, what are the sources of this randomness? In this case the conceptual model of software which is most widely used is the input-program-output model as shown in Fig. 2. According to this model, some (random) mechanism selects points from the input space to be processed by the program and produces points of the output space. The program can thus be seen as a mapping of I into O, that is p: I→O. One would observe failures in the output whenever the program received input from the subset I_F: this subset being encountered "randomly", and thus the failures in the output space occurring randomly. Thus, if one knows the properties of the program totally, it might be reasonable to assume that the failure process would be purely random, reflecting the fluctuations of the input data stream. Usually, the reasoning at this point about this model stops and inputs are seen as the only source of randomness. If one imagines that two programming teams have been set perform the same task, each to write a program for the same specification. The resulting programs pl, p2 then operate in identical environments (i.e. have the same input space, I) and their outputs are compared by a comparator (quality engineer, customer, etc.). From the program specification the comparator will define a single set of correct outputs along with a single set of incorrect outputs (failures), O_F, which is taken together to form the total output set, O. Since the input set is the same for both programs, the difference between them is revealed as a difference between two mappings from I to O, with the same subset O_F defining failures in each case as shown in Fig. 3. In other words, the two programs will differ in the way they partition the input space into a region I_F, which will produce failures, and its complement. Our uncertainty about the programs, then, can be regarded as uncertainty about the nature of I_F. If the program is changed, in a bug-fixing attempt, the partitioning of I changes: say I_F becoming I_F'. Although, the intention is to improve reliability by removing sources of failure, this cannot be guaranteed. Indeed, the bug fixing itself is a new source of uncertainty.

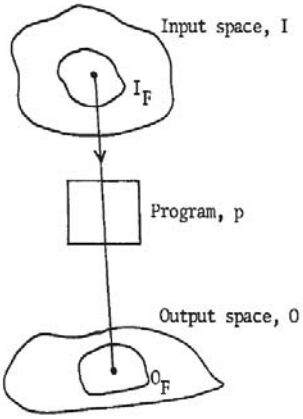

Fig. 2 Input-program-output model of Littlewood

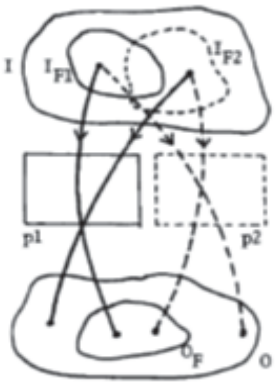

Fig. 3 The model with two programs by Littlewood

Cheung [1], in 1980 introduced a model which measures reliability of a service to users depends on the reliability of the components for a user profile. The reliability value was measured with respect to a user environment. Yet, sometimes for a given operation, it becomes too difficult to get the exact operation profile. In accordance with [2], although several software reliability growth models exist like, Failure count model, Time between failure model, etc. yet these models may not be versatile enough with respect to various types of systems because of their various assumptions. Many models assume that there is equal contribution of all the faults (present in the program) to reliability of software. This may not always be true since different types of faults may have different occurrence frequencies. Further, some models assume that failures are independent, bugs can be eliminated in negligible time and perfect debugging. Again, all these assumptions may not always be realistic for various software systems.

In 1988, Weiss et al. [3] provided a model for obtaining the probability value for logical correctness by using α-discrepancy between the observed and expected output. Again, there may be occurrences of operational failures caused by operational errors (e.g. overflow, etc.) even when the program is logically correct. But, this was not focused on and moreover no methods were specified to select the test cases. In accordance with [3], most of the time-dependent definitions expresses that the reliability of a program is the probability

that a software error which causes a discrepancy from the specified requirements, by more than specified tolerances, in a specified environment, does not lead to a failure during a specified exposure period [15]-[21]. Domain-based definitions usually allow the "exposure period" be a run of the program thereby, attempting to capture the idea that the reliability of a program corresponds to the probability that an arbitrary run of the program will produce the specified results [21], [22]. Time-dependent definitions provide a straight forward approach to measuring reliability; the indexes most commonly derived in such an approach include the mean time to failure, number of remaining errors, and probability of failure in a time interval [0, *t*]. Various models which use time as the unit of exposure are distinguished by the different assumptions that they make concerned with questions like the existence of faults or the occurrence of failures. There has been considerable debate as to which among the time-based models is the most reasonable [23], [24]**.** From the history, one finds that these models have been judged in part on statistical analysis, empirical results, and ease of use.

Iannino et al. [25] proposed a set of criteria to judge software reliability models, particularly the time-dependent models, and recommended that these criteria be used to compare and assess these models. For the two principal domain-based models like Nelson model [22] and Brown and Lipow's model [23], the value of the reliability function does not depend on time. The reliability function ***R,*** in these models, remains constant, but the estimates of its value change as more information becomes available. It is in contrast to the time-based models, where the value of the reliability function *R (t)* changes as more information becomes available. Yacoub et al. [4] introduced a path based approach considering various execution scenarios. It obtained the reliability of a component based system using the reliability values of the components, their interface and link reliabilities. But, there were no guidelines for obtaining the reliability values of used components. Gayen [5] analyzed the shortcomings in the conventional failure rate based models and proposed a new model for estimating minimum reliability of software. In the useful life of a software, when there are no upgrades, various authors have contradicting opinions. According to [5], if a software is not modified, the reliability value should not change with time. However, the reliability can change based on the changes in operating environment. *Error seeding* can be used to predict the number of residual errors in a system, provided that the seeded errors match closely with the kind of defects that actually exist. Although, it is difficult to predict the different types of errors that might exist in software, yet to some extent, the different categories of errors that may remain can be estimated to a first approximation by analyzing the historical data of similar software. But, this may not always prove to be useful since there may be more than one errors causing a failure.

For the reliability assessment of reliability of a modular software system, [6] developed an adaptive framework using path testing. A sensitivity analysis was carried out to find out the important components. But, again they have used a black box approach for obtaining the reliability values of individual components with various assumptions which may not always be realistic. Mirandola et al. [11] stressed that the reliability of a software system depends not only on the reliability of the individual components, their interactions, and their operating environment but also on the way in which the system is used (usage/operational profile). The impact of faults on reliability differs based on the way in which the system is used, (i.e. how often the faulty part of the system gets executed). When the usage profiles are unknown, the analysis of various ways and frequencies for executing the system is a big challenge for reliability prediction.

Fouadben Nasr Omri [12] stressed that statistical usage-based test cases are generated and commonly used for assessing the reliability of a software. But in most of the cases it is impracticable, as because to reach a target reliability value even for a small software the number of test cases executions required is too large [13], [14]. In accordance with [7], [28] considerable amount of research effort in the past were focused on modeling the reliability growth during the debugging phase. These black-box models treat the software as a monolithic whole, considering only its interactions with external environment, without an attempt to model internal structure. Usually, in these models no information other than the failure data is used.

The common feature of black-box models is the stochastic modeling of the failure process, assuming some parametric model of cumulative number of failures over a finite time interval or of the time between failures. Failure data obtained while the application is tested are then used to estimate model parameters or to calibrate the model. With the increasing emphasis on reuse, many organizations develop and use software as components which are parts of larger application. Use of Black-box models in this case may be quite inappropriate for modelling these component-based softwares. Hence, there arises a need for a model for analyzing software components and their interconnections. In this context, the white-box approach may be useful since it aims to estimate system reliability of the component based software by considering the information of its architecture.

The following are some issues motivating the use of architecture-based approaches for software systems:

i) Techniques are to be developed for the analysis of reliability/performance issues concerned with applications which are built from reusable and third party software components.
ii) The way in which component reliabilities/performance and component interactions affect the system reliability/ performance are to be understood.
iii) Guidance is to be provided to the process of identifying critical components and interfaces.
iv) The sensitivity of the reliability of the application to the reliabilities of various components used along with their interfaces are to be studied.
v) Techniques are to be developed for quantitative analysis which one is able to apply throughout the life cycle of software.

An exception to this may be considered in the case of software written from scratch. Since, the modern software engineering practice corresponds to making evolutions from existing software; as a result, there is an enhancement to the predictions performed for a product using field data with respect to previous and similar software products. But, when the architecture, in the absence of source code is only available, then one can proceed to build a simulation model for conducting studies similar to those designed for investigating the impact of components on the system reliability/performance for the specified architecture. The managers may use this information for rethinking their system's architecture, so as to proceed for planning their resource allocation. These projections and decisions might be constantly updated with the availability of new testing data. Considering the software architecture along with information about the components along with their interactions, calculations can be performed in what–if analysis. For example, what–if a certain component was more or less reliable? In what way would it affect system reliability? Architecture-based approach may be used for allocating effort to components which are critical from the reliability/ performance perspective as it permits an insight considering the sensitivity of the entire system to each component.

For calculating the system reliability from components information or for stipulating system reliability and calculating an allocated reliability of components, architecture-based approach can be used. These approaches can show the scale as well as the scope of effort which are needed to illustrate the required levels of components reliabilities. Further, it can also focus on the reasonableness of building a system to the desired reliability levels based on the reuse of a specific collection of components. Most of the work concerned with architecture-based reliability assessment, usually assume that reliability values of the components are available, and do not focus on developing techniques for obtaining them. Again the use software reliability growth models are not always possible in order to estimate the individual component's reliability, since a difficulty may arise due to the scarcity of failure data. Hence, predictions which are based on failure data are of little interest to users unless sufficient data is available for it to be statistically representative.

Considering the underlying assumptions of software reliability growth models (like, random testing performed in accordance with the operational profile where the independence of successive testing runs may be seriously violated in the unit testing phase) difficulties may arise. These models may be used for making reliability assessment of software considering the results of testing which is performed during its validation phase. Usually in this phase, hardly any changes are introduced in the software. In accordance with this context, testing do not correspond to a development activity concerned with discovering and fixing faults, but rather it is considered to be an independent assessment of the execution of software in its representative operational environment. But, the problem is that the number of executions required for estimating the levels of reliability values commensurate with some reasonable expectations. For, the probabilistic software reliability assessment for levels (as required for safety-critical applications), like failure rate $10^{-9}h^{-1}$ or 10^{-5} h^{-1} failure probability per demand, it is presently not reachable [29]. It is because to achieve a reasonable statistical confidence concerned with the reliability estimate random testing might require decades of testing. Hence, a practical current limit which is generally agreed upon is in the range of 10^{-2} to 10^{-4} h^{-1} [30], for the assessment of failure rate prior to operational use.

Several other techniques were also proposed for estimating component's reliability. Krishnamurthy and Mathur [31] considered the technique of seeding faults into software. Voas [32] determined the quality of COTS components by using a black-box testing method with the system-level fault injection. The fault-based techniques are considered to be only as powerful as the range of fault classes these techniques simulate [33]. But, it may not always be possible to capture the various classes of faults present in the software. Gokhale et al. [34] stressed that the software application as a whole is taken into consideration and only its interactions with the outside world are modelled (like a black box) in conventional approaches for analyzing the performance/reliability of the software applications.

III. The Problem

A major drawback of these approaches is that they ignore the internal structure of the application, and hence, the performance/reliability of the various parts of the application are not explicitly and individually captured [35], [36]. Even a moderate-sized software application is likely to be developed using a "divide and conquer" strategy and made up of several interacting parts. With the advent of component technologies and object-oriented programming people started realizing the vision of assembling software applications from systematically developed reusable software components [37]. As a result, a renewed interest was generated in "architecture-based analysis" which aims to characterize the performance/reliability behavior of software applications based on the behavior of the "components" and the "architecture" of the application [7], [31], [38]-[44]. However, many approaches assume that a fault/defect in software causes a failure, without considering the aspect that there could be more than one faults/defects leading to a failure. Even, there could be some faults/defects which may not lead to a failure.

From the survey, it can be inferred that in many approaches it is found that various assumptions are made in various black box models, like failures are independent, perfect debugging and failure can be corrected in a negligible time [8] etc. These assumptions may not be very realistic and many of the software do not behave in accordance with these assumptions. Since component/system execution time depends on various instance characteristics like input data, execution scenario, loops (which may be indefinite) etc. and hence it varies with various instances of execution of the program. Therefore, it becomes very difficult to obtain the exact usage ratio for component usage ratio. Dependencies among the components

have not been taken into consideration in many approaches. Even if the reliability bounds, delivers the range of reliability values irrespective of any operational profile, yet, sometimes it becomes necessary for practitioners to use exact values for a given operation (i.e. the reliability value when using a particular operation profile, for a given operation). But again, sometimes it becomes indeed a mammoth task in obtaining the exact operation profile concerned with an operation.

IV. The Proposed Conceptual Model

For the input taken from the input space I to generate the output in the output space O, various execution paths in the program module are followed, based on the input domain and other instance characteristics (concerned with various execution scenarios of the program module). It is basically a modification of the model provided by Littlewood [26] with the addition of various execution paths (which may depend on the input data and other data generated by the program during execution based on the instance characteristics) for various execution scenarios of a program module as shown in Fig. 4.

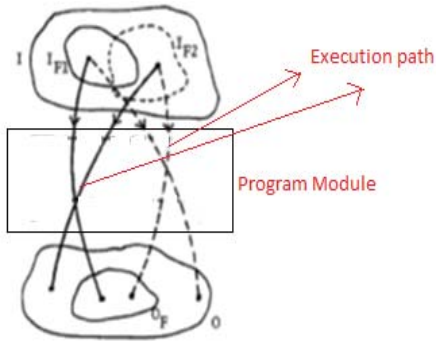

Fig. 4 The diagrammatic representation of the proposed conceptual model

Some examples where data is generated and used by the program based on the instance characteristics to follow various execution paths are as follows:

- In computing where *procedural generation* is the method of creating data algorithmically rather than manually. It is commonly used in computer graphics for creating textures in the context of video games it is also used for creating various other kinds of content such as items, quests or level geometry.
- Fractals, are examples of procedural generation, which express this concept, around which a whole body of mathematics (fractal geometry has developed). Commonplace procedural content includes textures and meshes.
- Sound, which is often procedurally generated as well, has applications in both speech synthesis as well as music.

It has been used to create several compositions in various genres of electronic music by artists like *Brian Eno* who popularized the term "generative music". Procedurally generated elements which have appeared in earlier video games is often used in film to rapidly create visually interesting and accurate spaces. One application which is known as "imperfect factory," is where artists can rapidly generate a large number of similar objects. It is because of the fact that, in real life, no two objects are ever exactly alike. For example, an artist could model a product for a grocery store shelf, and then create an imperfect factory which would generate a large number of similar objects to populate the shelf.

- It may also be used in intelligent software (concerned with Robotics, Automation and Testing) where Computational and Artificial Intelligence are used for designing, modelling and simulating various scenarios concerned with a process.
- In a multitasking operating system which uses signals to report exceptional situations to an executing program.

Some signals report errors such as references to invalid memory addresses, stack overflow, etc.; others report asynchronous events, such as disconnection of a phone line, etc. Some kinds of events make it inadvisable or impossible for the program to proceed as usual, and the corresponding signals normally abort the program. Hence, considering this conceptual model, the problem gets confined to determining the reliability of a program module/component by considering various execution paths (which may depend on the input data and other data generated by the program during execution, based on the instance characteristics) for various execution scenarios of a program module. The obtained reliability values of various modules/components are to be combined using a suitable methodology based on the dependencies of the components for various execution scenarios to obtain the reliability of the component based software. From, the study it has been found that a system/subsystem (comprising of software and hardware) may fail when

- ➢ The software fails
  - when there is a fault in the software
  - when there is a fault in the hardware or when hardware fails
  - when there is a change in hardware (incompatibility)
- ➢ The hardware fails
  - when there is a fault in the software or when software fails
  - when there is a fault in the hardware
  - when there is a change in software (incompatibility)
- ➢ When both software and hardware fails together.

Considering these and several other issues (discussed earlier), a suitable model needs to be developed for the reliability assessment of a software system. Since, the testing environment needs to capture the operating environment for good results. Therefore, the reliability assessment starting from the subsystem (considering the operating environment of the subsystem) to the system may help us in obtaining good results.

A. The Proposed View for a Software System

A software system can be broadly viewed as a system which may comprise of Application Software components, Operating System components and hardware components having interdependencies among themselves. To obtain the

reliability value of each component and not the system as a whole, each component needs to be tested in isolation (unit testing). This may be accomplished by testing the component in its ideal operating environment. It is because the intention is to detect the failures which are caused by the faults present only in this component and not contributed by the faults present in other components of the system. In general, a Software System may comprise of the Application Software with the underlying Operating System (which may even include hardware). Fig. 5 shows a common example of a 3-*tier* architecture for a Software System.

In accordance with this architecture the Software System may fail under the following conditions:

i) When the Application Software fails considering the Operating System and the Hardware to work properly
ii) When the Operating System fails considering the Application Software and the Hardware to work properly
iii) When the Hardware fails considering the Application Software and the Operating System to work properly
iv) When both the Application Software and the Operating System fails considering the Hardware to work properly
v) When both the Application Software and the Hardware fails considering the Operating System to work properly
vi) When both the Operating System and the Hardware fails considering the Application Software to work properly
vii) When Application Software, Operating System and the Hardware fails.

Besides the system also may also fail when there are incompatibilities between

- the Application Software and Operating System
- the Operating System and Hardware

as is evident from the history of Ariane 5, developed under European Space Agency. In June 4, 1996 Ariane 5 Flight 501 veered off its flight path, lost control and got exploded resulting in a financial loss of the cargo and rocket of around $500 million [8]. Although, much of the software that was used for Ariane 5 was the same as in Ariane 4. In Ariane 4, it was working perfectly fine but it resulted in a failure in Ariane 5. Table I, shows the various conditions under which this *3-tier* software system may fail.

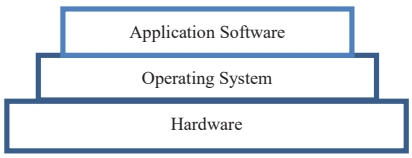

Fig. 5 A diagrammatic representation of the 3-tier architecture of a Software System

TABLE I
VARIOUS CONDITIONS UNDER WHICH THE CONSIDERED *3-TIER* SOFTWARE SYSTEM MAY FAIL

Application Software fails	Operating System software fails	Hardware fails	Incompatibilities between Application Software and Operating System software	Incompatibilities between Operating System software and hardware	System may fail
Yes	No	No	-	-	Yes
No	Yes	No	-	-	Yes
No	No	Yes	-	-	Yes
Yes	Yes	No	-	-	Yes
Yes	No	Yes	-	-	Yes
No	Yes	Yes	-	-	Yes
Yes	Yes	Yes	-	-	Yes
No	No	No	Yes	No	Yes
No	No	No	No	Yes	Yes
No	No	No	Yes	Yes	Yes
No	No	No	No	No	No

"-" denotes irrespective of Yes/No

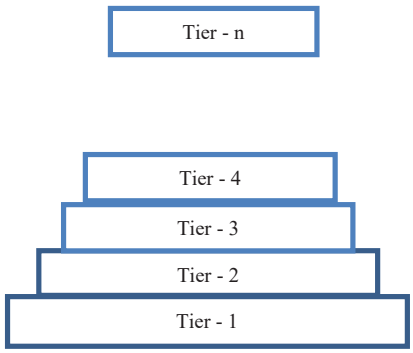

Fig. 6 Diagrammatic representation of an *n-tier* System

The reliability values of this *3-tier* software system can be expressed as:

Rel_{sys} = P(Proper execution of the system comprising of Application software, Operating System software and the hardware) = P(Proper execution of Application Software | The subsystem comprising the Operating System software and the hardware is ideal) * P(Proper execution of the subsystem comprising of the Operating System software and the hardware) * P(not encountering failure due to incompatibilities between Application Software and Operating System software)

P(Proper execution of the subsystem comprising the Operating System software and the hardware) = P(Proper execution of Operating System Software | The subsystem comprising only hardware is ideal) * P(Proper execution of the subsystem comprising only hardware) * P(not encountering failure due to incompatibilities between Operating System software and hardware)

P(Proper execution of the subsystem comprising only hardware) is the reliability value of the hardware component where P(x) corresponds to the probability of 'x', P(x|y) corresponds to the conditional probability of 'x' when 'y' takes place. On generalizing this for an *n-tier* system (as shown in Fig. 6), the following formula can be used for obtaining the reliability value of an *n-tier* system.

Rel_{sys} = P(Proper execution of the system/subsystem from *tier -1* to *tier - n*)
= P(Proper execution of the *tier - n* component | The subsystem from *tier - 1* to *tier – (n-1)* is ideal) *P(Proper execution of the subsystem from *tier - 1* to *tier – (n-1)*) * P(not encountering failure due to incompatibilities between *tier – n* and *tier – (n-1)* components) (1)

P(Proper execution of the subsystem from *tier - 1* to *tier – (n-1)*) can be *obtained* by replacing *n* by *n-1* in (1).
Hence, the reliability values of the other subsystems can be obtained by substituting suitable values for *n* in (1).

P(Proper execution of *tier -1* component) is the reliability value of the *tier-1* component.

Reliability values of each component used may be obtained (and not the system as a whole), by testing each component in isolation (unit testing). (This may be accomplished by testing the component in its ideal operating environment) in accordance with a suitable structural analysis based reliability assessment approach.

V. Conclusion

Although there are various software reliability models existing today, yet there is no suitable model which can be used for reliability assessment of software. Many of the models are black box models, which make certain assumptions like failures are independent, perfect debugging and failure can be corrected in a negligible time [8] etc. These assumptions may not be very realistic as many of the softwares do not behave in accordance with these assumptions. Since component/system execution time depends on various instance characteristics like input data, execution scenario, loops (which may be indefinite) etc. and hence it varies with various instances of execution of the program. Therefore, it becomes very difficult to obtain the exact usage ratio for component usage ratio. Dependencies among the components has not been taken into consideration in many approaches. The reliability bounds, although provides us the range of reliabilities values irrespective of any operational profile, yet for practitioners, it becomes sometimes necessary to use the exact reliability values for a specified operation (i.e. using a specific operation profile concerned with the given operation). Again, sometimes it becomes difficult to obtain the exact operation profile for a given operation. Various models like failure count model, time between failure models etc. only focus on observing the way in which the system behaves irrespective of the structure of the system. They treat the software as a black box and do not consider its structure and architecture of the system i.e. they do not consider about the components (composing the system), and their interconnections. These models are not suitable to find out the set of all execution paths for estimating reliability because they do not consider the internal details of the system.

Thus, these black box models may not be suitable for estimating the reliability of large or complex component based system. The entropy (the degree uncertainty) is much higher for black box models as compared to white box models. This paper will help the practitioners, developers, quality assurers and testers for reviewing the issues concerned with black box Software Reliability assessment in order to obtain a suitable model for the reliability assessment of their software.

Acknowledgment

The authors would like to thank the staffs for their support behind this work.

References

[1] R. C. Cheung. "A user-oriented software reliability model", *IEEE Transactions on Software Engineering*,vol.6, pp.118-125,1980.
[2] A. L. Goel. "An analysis of competing software reliability models", *IEEE Transactions on Software Engineering*, vol.6, pp.501-502,1980.
[3] S N. Weiss and E.J Weyuker "An Extended Domain-Based Model of Software Reliability", *IEEE Transactions on Software Engineering,* vol. 14, pp.1512-1524,1988.
[4] S. Yacoub, B. Cukic, H. H Ammar, "A scenario-based reliability analysis approach for component-based software", *IEEE Transactions on Reliability*, vol.53, pp. 22-31,2004.
[5] T. Gayen. "Analysis and proposition of error based model to predict the minimum reliability of software", *International Conference on Education Technology and Computer,* Singapore, pp.40-44*, 2009.*
[6] C. J. Hsu and C. Y. Huang. "An adaptive reliability analysis using path testing for complex component-based software systems", *IEEE Transactions on Reliability,* vol. 60, pp.158-170, 2011.
[7] G. Popstojanova, K. S. Trivedi, "Architecture-based approach to reliability assessment of software systems", *Performance Evaluation*, vol. 45, pp.179-204, 2001.
[8] S. Garnaik, T. Gayen., S. Mishra S, "Reliability Enhancement of Software by Minimizing the Overflow Errors", *International Journal of Systems Assurance Engineering and Management, Springer*, Vol. 5, No. 4, pp. 724-730, 2014.
[9] Michael R. Lyu *Handbook of Software Reliability Engineering.* McGraw-Hill publishing, 1995, ISBN 0-07-039400-8
[10] ANSI/IEEE, "*Standard Glossary of Software Engineering Terminology*", STD-729-1991, ANSI/IEEE, 1991
[11] R. Mirandola, P. Potena, E. Riccobene, P. Scandurra, "A reliability model for Service Component Architectures", Vol. 89, pp. 109–127, 2014.
[12] Fouad ben Nasr Omri, Ralf Reussner, "Towards Reliability Estimation of Large Systems-of-Systems with the Palladio Component Model", Trusted Cloud Computing, Springer, 2014.
[13] Whittaker, J. A., Poore, J.H., "Markov analysis of software specifications. ACM Transactions on Software Engineering Methodology", Vol. 2, Issue 1, 93–106, 1993.
[14] Poore, J., Mills, H., Mutchler, D, "Planning and certifying software system reliability", IEEE Software,Vol. 10, Issue 1, pp.88–99, 1993.
[15] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative evaluation of software quality," in Proceedings of Second International Conference on Software Engineering, San Francisco, CA, Oct. 1976, pp. 592-605.
[16] A. L. Goel, "A guidebook for software reliability assessment,'' Syracuse Univ., Syracuse, NY, Tech. Rep. No. 83-11, Apr. 1983.
[17] H. Hecht, "Mini-tutorial on software reliability," in *Proc.* IEEE *COMPSAC80,* Chicago, IL, pp. 383-385, 1980
[18] IEEE Standard *Dictionary of Electrical and Electronic Terms.*2nd ed., New York: IEEE Press, 1977.

[19] W. H. MacWilliams, "Reliability of large real-time control software systems," in *Rec. **1973** IEEE Symp. Computer Software Reliability,* New York, pp. 1-6, 1973.
[20] M. L. Shooman, *Sofrware Engineering.* New York: McGraw-Hill, 1983.
[21] T. A. Thayer, M. Lipow, and E. C. Nelson, *Software Reliability* (TRW Ser. Software Technol. 2). New York: North-Holland, 1978.
[22] J. R. Brown and M. Lipow, "Testing for software reliability," in Proceedings of International Conference on *Reliable Software,* Los Angeles, CA, pp.5 18-527, 1975.
[23] A. L Goel, "A summary of the discussion on 'An analysis of competing software reliability models" *IEEE Transactions Software Engineering***,** Vol. SE-6, pp. 501-502, Sept. 1980.
[24] G. J. Schick and R. W. Wolverton, "An analysis of competing software reliability models," *IEEE Transactions on Software Engineering,* vol. SE-4, pp.104-120, Mar. 1978.
[25] A. Iannino, I. D. Musa, K. Okumoto, and B. Littlewood. "Criteriafor software reliability model comparisons," *IEEE Tranactions Software Engineering,* vol. SE-IO, pp. 687-691, Nov. 1984.
[26] Bev Littlewood, "How to Measure Software Reliability and How Not To", *IEEE Transactions on Reliability*, Vol.R-28, Issue 2, pp. 103 – 110, 1979.
[27] Bev Littlewood, "Theories of Software Reliability: How Good Are They and How Can They Be Improved*", IEEE Transactions on Software Engineering*, Vol. SE-6, no. 5, 1980.
[28] G.Popstojanova, K S Trivedi, "Architecture-based approaches to software reliability prediction", *Computers & Mathematics with Applications, Elsevier*, Vol. 46, Issue 7, pp. 1023–1036, 2003.
[29] R. W. Butler, G. B. Finelli, The infeasibility of quantifying the reliability of life-critical real-time software, *IEEE Transactions on Software Engineering* Vol.19 Issue1, pp. 3–12, 1993.
[30] J. C. Laprie, Dependability of computer systems: concepts, limits, improvements, in: Proceedings of the *Sixth International Symposium on Software Reliability Engineering* (ISSRE'95), pp. 2–11, 1995.
[31] S. Krishnamurthy, A.P. Mathur, On the estimation of reliability of a software system using reliabilities of its components, in: Proceedings of *the Eighth International Symposium on Software Reliability Engineering* (ISSRE'97), pp. 146–155, 1997
[32] J. M. Voas, Certifying off-the-shelf software components, *IEEE Computer*, Vol. 31, Issue 6, pp. 53–59, 1998.
[33] J. M. Voas, C. C. Michael, K. W. Miller, confidently assessing a zero probability of software failure, *High Integrity Systems*, Vol. 1, Issue 3, pp. 269–275, 1995
[34] Swapna S. Gokhale, W. Eric Wong, J. R. Horgan, Kishor S. Trivedi, "An analytical approach to architecture-based software performance and reliability prediction", *Performance Evaluation Journal*, Elsevier, Vol. 58, Issue 4, pp. 391 – 412, 2004.
[35] D. Hamlet. "Are we testing for true reliability?". *IEEE Software*, 13(4):21–27, 1992.
[36] J. R. Horgan and A. P. Mathur, "Handbook of Software Reliability Engineering", M. R. Lyu, Editor, chapter "Software Testing and Reliability", McGraw-Hill, NewYork, NY, pp. 531–566, 1996.
[37] D. L. Parnas, P. C. Clements, and D. M. Weiss. "The modular structure of complex systems". *IEEE Transactions on Software Engineering*, SE-11(3):259–266,1985.
[38] W.W. Everett. "Software component reliability analysis". proc. of *Application Specific Software Engineering and Technology*, Dallas, TX, March 1999.
[39] S. Gokhale, M. R. Lyu, and K. S. Trivedi. "Reliability simulation of component-based software systems". In *Proc. of Ninth Intl. Symposium on Software Reliability Engineering (ISSRE 98)*, pages 192–201, Paderborn, Germany, 1998.
[40] S. Gokhale and K. S. Trivedi. "Structure–based software reliability prediction". In *Proc.of Fifth Intl. Conference on Advanced Computing (ADCOMP 97)*, pages 447–452, India, 1997
[41] D. Hamlet, D. Woit, and D. Mason. "Theory of software reliability based on components", In *Proc. of Intl. Conference on Software Engineering*, pages 361–370, Toronto, Canada, 2001.
[42] J. Ledoux and G. Rubino. "A counting model for software reliability analysis". *IASTED Journal on Simulation*, 1997.
[43] J. Ledoux and G. Rubino. "Simple formulas for counting processes in reliability models", *Theory of Applied Probability*, 1997.
[44] W. Wang, Y. Wu, and M. H. Chen. "An architecture-based software reliability model", In *Proc. of Pacific Rim Dependability Symposium*, Hong Kong, December 1999.
[45] Hoang Pham, "System Software Reliability", Springer, 2005.
[46] Hoang Pham, "Software Reliability", *Springer*, 2000.