METRIC-BASED FRAMEWORK FOR TESTING & EVALUATION OF SERVICE-ORIENTED SYSTEM

Salisu Garba

Department of Mathematics & Computer Science, Sule Lamido University, Kafin Hausa. Jigawa State.

ABSTRACT

The increase in the significance of service orientation in system development is accelerating with an increase in demand for qualitative and cost-effective systems. Service-Oriented Architecture (SOA) is one of the established structural designs used for developing and implementing flexible, reusable, rapid and low-cost service-oriented systems. The established testing and evaluation methods don't work well for systems that are made-up of services (service-oriented system). As a result, several testing and evaluation metrics for service-oriented systems were proposed. However, these metrics were created based on preceding software development approaches that offer insufficient focus to service-oriented systems thereby lacking the efficiency to evaluate these systems. Furthermore, Lack of access to source code also frustrates classical mutation-testing approaches, which require seeding the code with errors. This paper discusses different testing and evaluation metrics available for SOS and proposed a theory-grounded framework for testing and evaluation of service-oriented systems with the aim of decreasing cost and increasing the quality of the SOS. Then, the proposed framework is validated theoretically to check its usability and applicability for testing and evaluation of SOS. The results show that the proposed framework is able to decrease cost and increasing the quality of the SOS.

KEYWORDS

Service Oriented systems, SOA, Metrics, testing, cost evaluation, quality evaluation

1. INTRODUCTION

The persistence storms of the Internet, TCP/IP, HTTP, and XML have created the circumstances for another incarnation of SOA again. Due to the universal support for those technologies, now SOA has the potential to have a wider, ever permanent encounter than beforehand. Service Oriented Architecture enables flexibility, adoptability, integrability, business adaptability and the ability to incrementally change the system, switching service providers, extending services, modifying service providers and consumers due to loose-controlled coupling.

Essentially, beyond the technical definition, SOA is a change of paradigm, a change in the way of thinking about information technology (IT), and the process of delivering IT (via services) from start to end in an easier, more flexible manner, more reusable and more responsive to business changes while providing cost efficiency as a major benefit. Service Oriented Architecture (SOA) is devised to standardize obtainable IT resources and transformed the heterogeneous collection of distributed, intricate systems and applications into a set-up of integrated, straightforward and flexible IT assets.

Prior to Service-oriented architecture, the Common Object Request Broker Architecture (CORBA) and the Distributed Component Object Model (DCOM) provide similar and related functionality. These existing approaches to service orientation, however, suffered from a few tricky problems such as tightly coupled scenarios according to [4]. It is significant to recognize that SOA is not a technology, but a method of software design that proposes a fundamental shift in how organizations implement systems. SOA marks the end of monolithic enterprise applications and marks the commencement of a more flexible and adoptable business process-centric application. SOA Applications are built based on services. Therefore, it is very important to understand the word service clearly. According to [1], a service is a software component that is well-defined, self-contained, and independent on the situation or status of other services. A service is an implementation of well-defined company functionality, consumed by clients in disparate applications or company procedures.

Services are connected together using Web Services. However, Web services are merely a step along a much longer road. Web Services are the composition of protocols by which Services can be published, discovered and utilized in a technology impartial, methodology neutral, platform neutral, and language-neutral standard form. Services in SOA concentrated on conceding a schema and message-based contact alongside an application across interfaces that are application scoped, and not constituent or object-based.

The established testing and evaluation methods don’t work well for systems that are made-up of services (service-oriented system). As a result, several testing and evaluation metrics for service-oriented systems were proposed. However, these metrics were created based on preceding software development approaches that offer insufficient focus to service-oriented systems thereby lacking the efficiency to evaluate these systems. Furthermore, Lack of access to source code also frustrates classical mutation-testing approaches, which require seeding the code with errors.
This paper proposed a metric based framework for testing & evaluation of service-oriented system so as to enhanced quality as well as the effectiveness of testing and evaluation of service-oriented systems. The rest of this paper is organized as follows; the SOA Rationale and SOS Design Principles are discussed in section 2. The proposed framework is illustrated and discussed in section 3. The conclusion and future work are discussed in section 4.

2. SOA RATIONALE AND SOS DESIGN PRINCIPLES

A considerable amount of literature has been published on the SOA rationales and design principles. Enterprise architects regard SOA as an architectural evolution rather than revolution as it captures many of the excellent features of previous software architectures. Services are the building blocks of any software architecture, which is the implementation of well-defined business functionality, consumed by clients in different applications or business processes. Nowadays SOA has removed one more barrier by permitting application to interconnect in an object-model-neutral method. For example, employing a simple XML-based messaging scheme, Java requests can implore Microsoft .NET requests or CORBA-compliant, or even COBOL, applications.

[4] states that the intrinsic property of many modern computing paradigms (e.g. peer-to-peer systems, distributed systems, and smart environments) is the distribution of services and control among multiple entities (or agents), be it software, human or a mix of both. Service Oriented Architecture enables flexibility, adoptability, integrability, business adaptability and the ability to switch service providers, extend services; modify service due to loosely coupling.

Previous integration models such as point to point and spoke and the wheel had certain limitations. The complexity of application integration for a point to point model rises substantially with every new application that needs to communicate and share data with it. The Enterprise

Service Bus is an improvement over these two architectures and plays a critical role in connecting heterogeneous applications and services in a Service-Oriented Architecture.

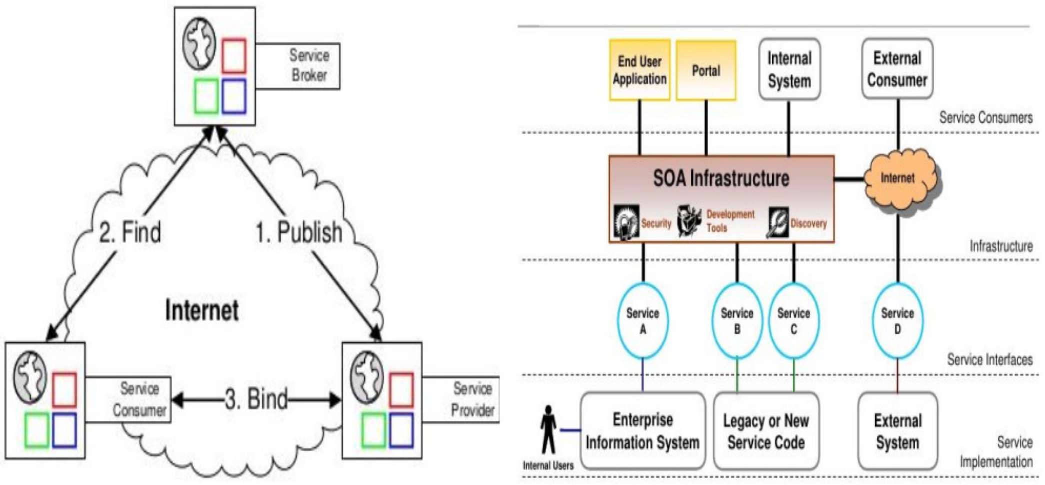


Figure 1. The basic concept of SOA and the components of SOS

The principle of service orientation includes loose coupling, reusability, statelessness, abstraction, autonomy, composability, discoverability. Therefore, the fundamental aim of SOA is to align enterprise IT competence with company goals, and to facilitate enterprise IT to respond with better agility toward business requirements, allowing employees, trading partners, and customers to respond extra quickly and become accustomed to shifting business demands.

A considerable amount of literature has been published on SOS design principles. While other authors such as [1], [17], [4] take account of Service normalization, Service optimization, Service relevance, Service encapsulation, Service location transparency as principles of designing SOS. The table below shows the ground rules that must be followed in designing SOS.

SOA principles that promote loose coupling, standards-based technologies, and coarse-grain service design enable the creation of reusable services repository that can be pooled into higher-level services and the composite system as new business needs arise. These lower the cost development, testing, and maintenance.

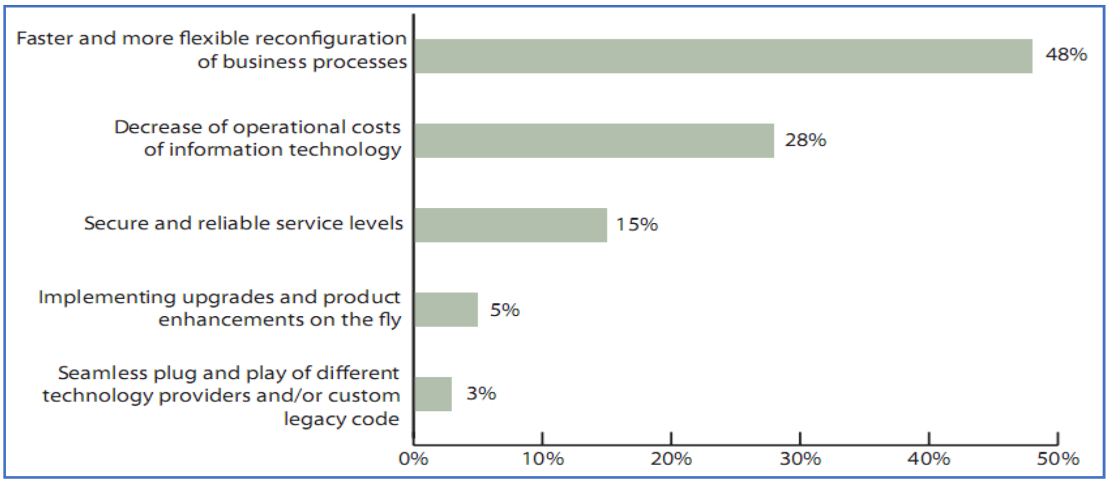


Figure 2. Expected Benefits of SOA (Adopted from: [7])

3. THE PROPOSED SOS TESTING AND EVALUATION FRAMEWORK

The proposed metric-based framework for testing & evaluation of the service-oriented system is shown in Figure 3 below. The detail activities and rationales in each part of the framework are discussed below.

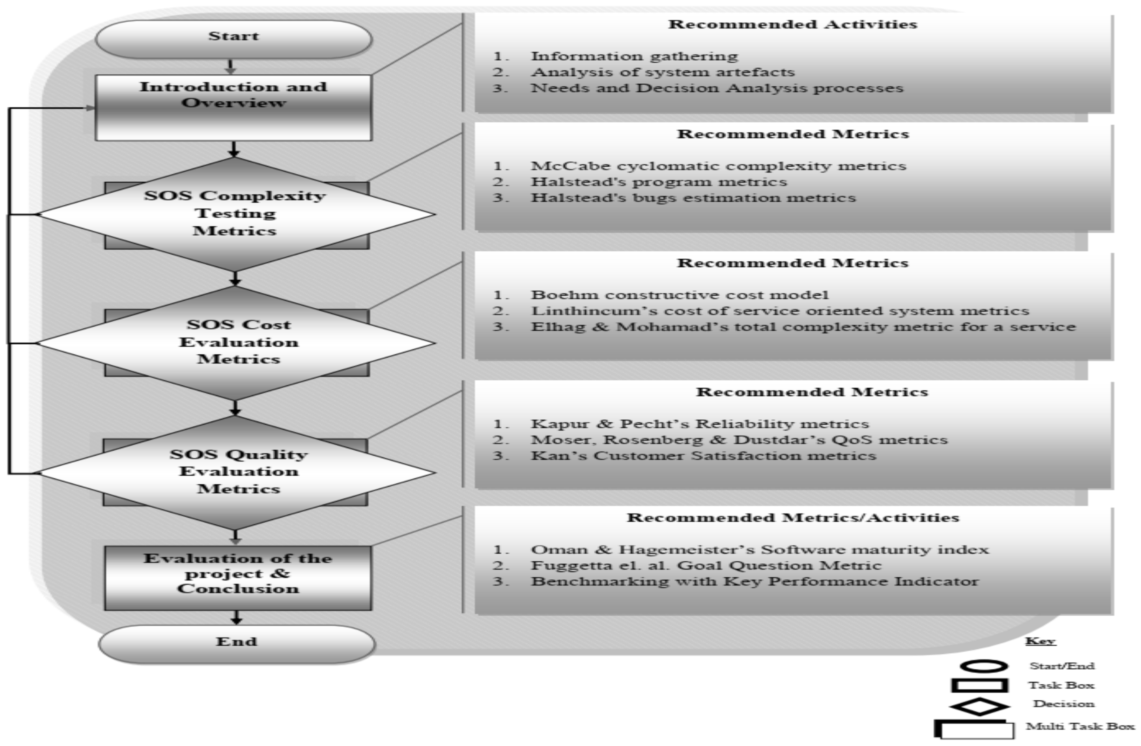

Start
Introduction and Overview
SOS Complexity Testing Metrics
SOS Cost Evaluation Metrics
SOS Quality Evaluation Metrics
Evaluation of the project & Conclusion
End
Recommended Activities
1. Information gathering
2. Analysis of system artefacts
3. Needs and Decision Analysis processes
Recommended Metrics
1. McCabe cyclomatic complexity metrics
2. Halstead's program metrics
3. Halstead's bugs estimation metrics
Recommended Metrics
1. Boehm constructive cost model
2. Linthincum's cost of service oriented system metrics
3. Elhag & Mohamad's total complexity metric for a service
Recommended Metrics
1. Kapur & Pecht's Reliability metrics
2. Moser, Rosenberg & Dustdar's QoS metrics
3. Kan's Customer Satisfaction metrics
Recommended Metrics/Activities
1. Oman & Hagemeister's Software maturity index
2. Fuggetta el. al. Goal Question Metric
3. Benchmarking with Key Performance Indicator
Key
Start/End
Task Box
Decision
Multi Task Box

Figure 3. The proposed SOS testing and evaluation framework

3.1. Testing Metrics

According to [8], it's next to impossible to control what cannot be measured. By his saying, it is very clear how important software measures are. The metrics we are about to discuss aim at getting empirical laws that relate SO program size to the expected number of bugs, the expected a number of tests required to find bugs, testing technique effectiveness. Linguistic Metrics that are based on measuring properties of SO program text without interpreting what the text means such as a line of codes (LOC) is highly inaccurate when used to predict costs, resources, and schedules. However, Structural Metrics that are based on structural relations between the objects in a SO program such as the number of nodes and links in a control flow-graph should only be used as a rule of thumb at best.

Cyclomatic Complexity is a software metric (measurement), used to indicate the complexity of a program. [13], states that if G is the control flowgraph of the program (P) and G has edges (E) and nodes (N), then the cyclomatic complexity of program (P) can be established using the following metrics.

$$V(G) = E - N + 2$$

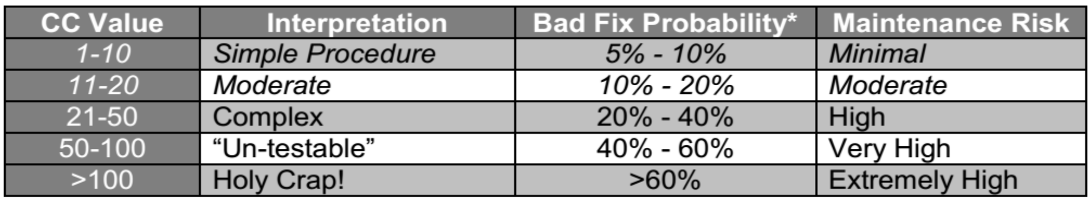
$$V(G) = 16 - 13 + 2, V(G) = 5$$

Alternatively, the cyclomatic complexity can also be determined by identifying the number of linearly independent path in the control flowgraph of program (P) or simply by determining the number of decision nodes in G. The metrics below shows how the cyclomatic complexity of the program (P) can be established using the decision nodes (D) in G
.

$$V(G) = D + 1$$

$$V(G) = 4 + 1, V(G) = 5$$

Table 1. Cyclomatic complexity interpretation



CC Value	Interpretation	Bad Fix Probability*	Maintenance Risk
1-10	*Simple Procedure*	*5% - 10%*	*Minimal*
11-20	*Moderate*	*10% - 20%*	*Moderate*
21-50	Complex	20% - 40%	High
50-100	“Un-testable”	40% - 60%	Very High
>100	Holy Crap!	>60%	Extremely High

According to [6], establishing an empirical science of software development is very essential for the maturity of the discipline. The objective was to identify quantifiable attributes of software, and the relations between them, thereby evolving philosophical discussions to quantification. This is comparable to the discovery of quantifiable attributes of matter (such as volume and mass) and the relationships between them (corresponding to the gas equation). Therefore, Halstead's metrics are really more than just complexity metrics. It states that the vocabulary of a program (η) can be determined by summing the number of distinct operators (keywords) and the number of distinct operands (data objects) as shown in the equation below;

$$\text{Vocabulary of the Program} : \eta = \eta_1 + \eta_2$$

While the length of the program (N) can be determined by summing the total number of operators (keywords) and the total number of operands (data objects) as shown in the equation below. However, the length of the program (N) should not be confused with the line of codes, therefore $\text{N} \neq \text{LOC}$

$$\text{Length of the Program} : N = N_1 + N_2$$

$$or$$

$$\text{Length of the Program} : N = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$$

The Volume of the program (V), the difficulty or complexity of the program (D), the amount of effort required (E) and the time needed to program the service-oriented system to can be determined using the metrics below;

$$\text{Volume of the Program} : V = N * \log_2 \eta$$

$$\text{Difficulty of the Program}: D = \frac{\eta_1}{2} * \frac{N_2}{\eta_2}$$

$$\text{The Effort Required}: E = D * V$$

$$\text{The Required to program}: T = \frac{E}{18}$$

Software Engineers are still counting lines of code due to its popularity. However, the number of delivered bugs (estimated number of errors in the implementation of SOS) can be determined by dividing the volume of the program by a Halstead's constant of 3000.

$$\text{The number of delivered bugs}: B = \frac{V}{3000}$$

Authors in [3] compared actual to predicted bug counts to within 8% over a range of program sizes from 300 to 12,000 volume of statements. The validity of the metric has been confirmed experimentally many times, independently, over a wide range of programs and languages.

3.2. Cost Evaluation

For the majority of organizations, the initial stride of the SOS project is to outline the cost. So that budget can be estimated to get the funding. The predicament is that the cost estimation of entire SOS components is so complex and necessitate a clear understanding of the work that has to be done.

Authors in [2] introduced an empirical effort estimation model that is still referenced by the software engineering community. The constructive cost Model (COCOMO II) is the most widely used software estimation model in the world which predicts the effort and duration of a project based on inputs relating to the size of the resulting systems and a number of factors (cost drives) that influence software projects.

The complexity of the model can be determined by the number of factors (cost drives) that are taken into account to influence software projects thereby given a more accurate estimate. The development model is the most important factor that contributes to the cost and duration of the software project. This can be organic, semi-detached or embedded based on the complexity of the project.

The intermediate and advanced COCOMO models incorporate 15 'cost drivers'. These 'drivers' multiply the effort derived from the basic COCOMO model. The importance of each driver is assessed and the corresponding value multiplied into the COCOMO equation, which becomes:

$$Effort : E = a(S)^b * product(\mathrm{cos}\, t\; drivers)$$

Where: E represents effort in person-months, S is the size of the software development in KLOC (1000LOC), while a and b are constant values dependent on the development mode, this is multiplied by the product of cost drivers of the project which varies from very to extra high based on the importance of a particular cost driver to the project.

Table 2. Different modes of COCOMO II

Mode	a	b	c	d
Organic	3.2	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

$$Development\,Time : DT = c(E)^{d}$$

SOS Development Time can be computed using the above metrics; Where: DT represents development time in months, E represents an effort in person-months, while c and d are constant values dependent on the development mode.

$$Number\,of\,Personnel : NP = \frac{E}{DT}$$

The number of personnel for SOS Development can be computed using the above metrics; where: NP represents the number of personnel (people), E represents an effort in person-months, while DT represents development time in months.

The author in [11] proposed a formula to figure out how much an SOA project will cost as shown in the metric below. Where: C (SOS) is the Cost of SOS, CDC is the Cost of Data Complexity, CSC is the Cost of Service Complexity, CPC is the Cost of Process Complexity and ETS is the Enabling Technology Solution.

$$C(SOS) = CDC + CSC + CPC + ETS$$

Upon arrival at the Cost of SOS, [11] advises figuring in "10 to 20 percent variations in cost for the simple reason that this is a new approach to calculating the cost of the service-oriented system. However, Complexity measures the difficulty of understanding the interaction and relationships between the services and services operations, therefore, the total complexity of the service-oriented system can only be determined through the following equation.

$$TCM(s) = \frac{C(s) + NS(s) + NO(s)}{CM}$$

Where: TCM is the is the total complexity metric for a service, C is the coupling which can either be direct or indirect, NS is the number of services, NO is the number of operations and CM is the cohesion metrics. This is because coupling and cohesion are used to estimate the degree to which the components of the service-oriented system belong together and the strength of the relationships between operations in service [3].

3.3. Quality Evaluation Metric

In order to help us categorize software quality factors, McCall proposes a categorization which focuses on three important aspects of a software product (product revision, product transition, product operation). However, the de facto definition of software quality consists of two levels:

intrinsic product quality and customer satisfaction. Intrinsic product quality is usually measured by the number of "bugs" (functional defects) in the software or by how long the software can run before encountering a "crash."

Authors in [9] define software reliability as the probability of failure-free operation of a program in a specified environment for a specified time. Reliability metric is an indicator of how broken a program is. Metrics are best weighted by the severity of errors. A minor error every hour is better than a catastrophe every month. Mean Time Between Failure (MTBF) which measures how long a program is likely to run before it does something bad like a crash, where MTTF and MTTR are mean time to failure and mean time to repair respectively as shown in the metrics below.

$$\text{Re}\, liability\ of\ SOS = \frac{MTTF}{(MTTF + MTTR)} * 100\%$$

$$\text{Re}\, liability\ of\ SOS = \frac{MTTF}{MTBF} * 100\%$$

Good practice in software quality engineering, however, also needs to consider the customer's perspective. From the customer's point of view, the defect rate is not as relevant as the total number of defects that might affect their business. Therefore, a good defect rate target should lead to a release-to-release reduction in the total number of defects, regardless of size.

According to [14], [16], dealing with the problem of runtime adaptation of composite services that implement mission-critical business processes requires a combination of domain-agnostic and domain-specific quality of service attributes such as response time, throughput, availability, and accuracy.

Table 3. Quality of service metrics [14], [16]

QoS Attribute	Formula	Unit
Response Time	$\frac{1}{\#requests} \sum_{i=1}^{n} rt_i$	Milliseconds
Throughput	$\frac{\#requests}{second}$	Requests per Second
Availability	1 - $\frac{downtime}{uptime}$	Percent
Accuracy	1 - $\frac{\#failedrequests}{\#totalrequests}$	Percent

The author in [8] states that customer satisfaction metric consists of the use of a five-point scale survey to measure the level of customer satisfaction. Different organizations employ different parameter in determining the satisfaction level of a customer. One of the most widely used parameters of customer satisfaction in software quality is CUPRIMDSO (capability, functionality, usability, performance, reliability, installability, maintainability, documentation/information, service, and overall). However, some organizations prefer FURPS (functionality, usability, reliability, performance, and service) for simplicity.

Table 4. Five-Point scale customer satisfaction

Completely satisfied	Satisfied	Neutral	Dissatisfied	Completely dissatisfied
5	4	3	2	1

A number of metrics can be created based on the five-point-scale data, so as to analyze the customer's satisfaction level of the SOS. For instance:

(1) Percent of completely satisfied customers
(2) Percent of satisfied customers (satisfied and completely satisfied)
(3) Percent of dissatisfied customers (dissatisfied and completely dissatisfied)
(4) Percent of non-satisfied (neutral, dissatisfied, and completely dissatisfied)

Furthermore, the weighted index approach can be used to determine the Customer satisfaction level of the SOS. For example, some organizations use the net satisfaction index (NSI) which has the following weighting factors:

- Completely satisfied = 100%
- Satisfied = 75%
- Neutral = 50%
- Dissatisfied = 25%
- Completely dissatisfied = 0%

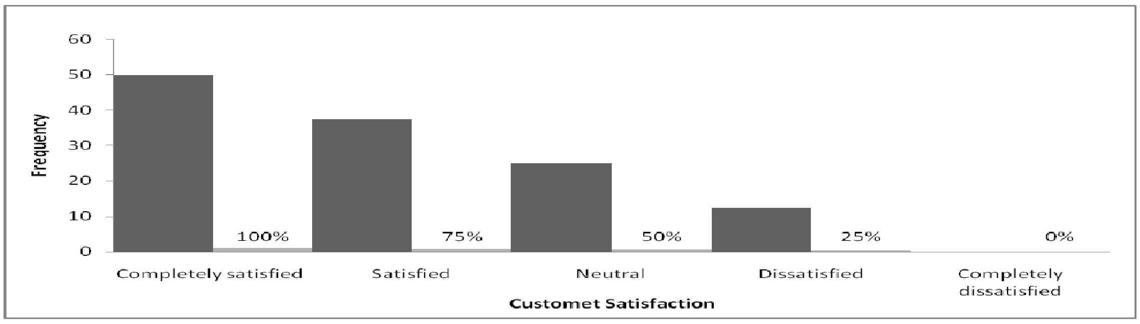


Figure 4. NSI customer satisfaction analysis

The range of the NSI starts from 0% (all customers are completely dissatisfied) to 100% (all customers are completely satisfied). If all customers are satisfied (but not completely satisfied), NSI will have a value of 75%. This weighting approach, however, may be camouflaging the satisfaction profile of one's customer set. For example, if half of the customers are completely satisfied and half are neutral, NSI's value is also 75%, which is equivalent to the scenario that all customers are satisfied.

If satisfaction is a good indicator of product loyalty, then half completely satisfied and half neutral is certainly less positive than all satisfied. Furthermore, we are not sure of the rationale behind giving a 25% weight to those who are dissatisfied. Therefore, this example of NSI is not a good metric for determining the customer's level of satisfaction with SOS; it is inferior to the simple approach of calculating the percentage of specific categories. If the entire satisfaction profile is desired, one can simply show the percent distribution of all categories via a histogram. A weighted index is for data summary when multiple indicators are too cumbersome to be shown.

For example, if customers' purchase decisions can be expressed as a function of their satisfaction with specific dimensions of a product, then a purchase decision index could be useful. In contrast, if simple indicators can do the job, then the weighted index approach should be avoided.

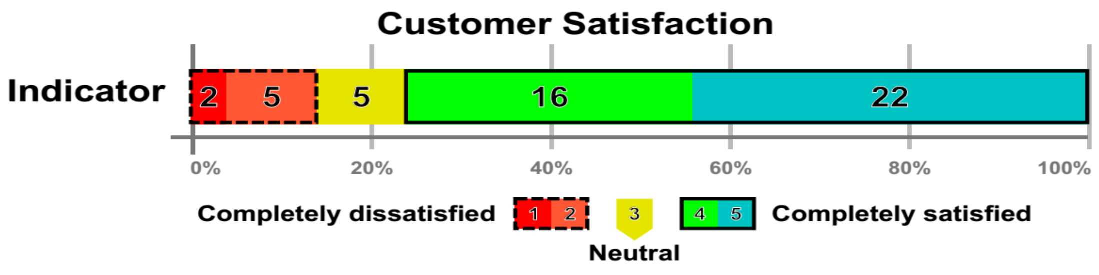


Figure 5. Customer Satisfaction indicator

System maintenance is any activity intended to eliminate faults or to keep programs in satisfactory working conditions. The author in [15] suggests a software maturity index (SMI) that provides an indication of the stability of a software product (based on changes that occur for each release of the product). The software maturity index is then computed in the following manner:

$$SMI = \frac{[M_T - (F_a + F_c + F_d)]}{M_T}$$

Table 5. Summary of testing & evaluation metrics with the assumptions, pros, and cons

Proposed Metrics	Assumptions	Pros	Cons
1. McCabe cyclomatic complexity metrics	Some upper limit numbers (6, 8 or 10 etc.) must be defined	It is about the detailed implementation and code execution.	Not suitable for large program with high complexity
2. Halstead's program metrics	This is supposed to satisfy the typical sizing aspects of SOA-based software.	Provides open range scales to take into account possibly high complexity functions	Wider set of guidelines for practical application
3. Halstead's bugs estimation metrics	The value helps in identifying bug causes and intervened early.	Essential for better testing and faster maintainability	Predicted bug counts fall to within 8% over a range of program sizes.
4. Boehm constructive cost model	COCOMO II considers two types of reused components black-box and white-box	COCOMO II model has a large number of coefficients such as effort multipliers	COCOMO II is generally inadequate to accommodate the cost estimation needs for SOS development.
5. Linthincum's cost of service oriented system metrics	10 to 20 percent variations in cost are expected.	Provide Simple range matrices for software cost evaluation	Some aspects of the calculation follow similar means without clarifying essential matters
6. Elhag & Mohamad's total complexity metric for a service	The complexity metrics is nonnegative	The complexity metrics provide null value when there are no interactions between the system components.	The complexity metrics are design to the quality of composite service design only
7. Kapur & Pecht's Reliability metrics	The environment is stable for a specified time.	Metrics identify the severity of errors or damage caused.	Reliability metric only indicate how broken a program is.
8. Moser, Rosenberg & Dustdar's QoS metrics	Include both domain-agnostic and domain-specific quality of service attributes	Deal with the problem of runtime adaptation of composite services for mission-critical business processes	Too many quality of service attributes such as response time, throughput, availability and accuracy.
9. Kan's Customer Satisfaction metrics	FURPS (functionality, usability, reliability, performance, and service) parameter is used for simplicity.	A number of metrics can be created based on the five-point-scale data, so as to analyse the customer's satisfaction level	The use of five-point scale survey to measure the level of customer satisfaction.
10. Oman & Hagemeister's Software maturity index	There is more than one version of the system	Provides an indication of the stability of a software product.	Everything is based on changes that occur for each release of the product.
11. Fuggetta el. al. Goal Question Metric	Assume that it is possible to identify certain SOA project types and certain context factors	SOA follows different goals on different levels of EA abstraction	The identification of relevant project types and context factors are not clear.

4. CONCLUSION

This paper has argued that testing and evaluation of cost and quality plays a vital role in system development, particularly service-oriented systems. However, the established testing and evaluation methods don't work well for systems that are made-up of services (service-oriented system) due to the fact that these metrics were created based on preceding software development approaches that offer insufficient focus to service-oriented systems thereby lacking the efficiency to evaluate these systems. Furthermore, Lack of access to source code also frustrates classical mutation-testing approaches, which require seeding the code with errors. Therefore, many metrics are proposed to test and evaluate the SOS. In this paper, a set of basic metrics is proposed and used for proposing derived metrics to evaluate the complexity, cost, quality, reliability and maintainability of SOS. Subsequently, the result is used to create a Metric based framework for Testing & Evaluation of Service Oriented System. The framework adds a new contribution is assessing the complexity and quality of SOS. The findings of this investigation complement those of earlier studies. The generalisability of these results is subject to certain limitations. For instance, the metrics do not pay too much consideration to the service that is built from other services (composite services) and only consider the operations as building blocks for the service-oriented system. Further investigation and experimentation in using the proposed framework is strongly recommended.

REFERENCES

[1] Basu, V., & Lederer, A. L. (2011). Agency theory and consultant management in enterprise resource planning systems implementation. ACM SIGMIS Database, 42(3), 10-33.

[2] Boehm, B. W., Madachy, R., & Steece, B. (2000). Software cost estimation with Cocomo II with Cdrom. Prentice Hall PTR.

[3] Elhag, A. A. M., & Mohamad, R. (2014, September). Metrics for evaluating the quality of service-oriented design. In Software Engineering Conference (MySEC), 2014 8th Malaysian (pp. 154-159). IEEE.

[4] Erl, T., Merson, P., & Stoffers, R. (2017). Service-oriented Architecture: Analysis and Design for Services and Microservices. Prentice Hall PTR.

[5] Fuggetta, A., Lavazza, L., Morasca, S., Cinti, S., Oldano, G., & Orazi, E. (1998). Applying GQM in an industrial software factory. ACM Transactions on Software Engineering and Methodology (TOSEM), 7(4), 411-448.

[6] Halstead, M. H. (1977). Elements of software science (Vol. 7, p. 127). New York: Elsevier.

[7] Kai, J., Miao, H., & Gao, H. (2016). A Survey of Quality Prediction Methods of Service-oriented Systems. International Journal of Hybrid Information Technology, 9(4), 183-198.

[8] Kan, S. H. (2002). Software quality metrics overview. Metrics and Models in Software Quality Engineering, 85-120.

[9] Kapur, K. C., & Pecht, M. (2014). Reliability engineering. John Wiley & Sons.

[10] Jensen, H. A., & Vairavan, K. (1985). An experimental study of software metrics for real-time software. IEEE Transactions on Software Engineering, (2), 231-234.

[11] Linthicum, D. (2007). How much will your SOA cost?. SOAInstitute. org, Mar.

[12] Li, H. F., & Cheung, W. K. (1987). An empirical study of software metrics. IEEE Transactions on Software Engineering, (6), 697-708.

[13] McCabe, T. J. (1976). A complexity measure. IEEE Transactions on software Engineering, (4), 308-320.

[14] Moser, O., Rosenberg, F., & Dustdar, S. (2012). Domain-specific service selection for composite services. IEEE Transactions on Software Engineering, 38(4), 828-843.

[15] Oman, P., & Hagemeister, J. (1992, November). Metrics for assessing a software system's maintainability. In Software Maintenance, 1992. Proceerdings., Conference on (pp. 337-344). IEEE.

[16] Rosenberg, L. H., & Sheppard, S. B. (1994, October). Metrics in software process assessment, quality assurance and risk assessment. In Software Metrics Symposium, 1994., Proceedings of the Second International (pp. 10-16). IEEE.

[17] Seth, A., Agarwal, H., & Singla, A. R. (2011). Designing a SOA based model. ACM SIGSOFT Software Engineering Notes, 36(5), 1-7.

[18] Seth, A., Agrawal, H., & Singla, A. R. (2014). Techniques for evaluating service oriented systems: A Comparative Study.