Evaluation of a Suite of Metrics for Component Based Software Engineering (CBSE)

V. Lakshmi Narasimhan, P. T. Parthasarathy, and M. Das
Department of Computer Science, East Carolina University
Greenville, NC, USA

narasimhanl@ecu.edu

Abstract

Component-Based Software Engineering (CBSE) has shown significant prospects in rapid production of large software systems with enhanced quality, and emphasis on **decomposition** of the engineered systems into **functional** or **logical** components with well-defined **interfaces** used for communication across the components. In this paper, a series of metrics proposed by various researchers have been analyzed, evaluated and benchmarked using several large-scale publicly available software systems. A systematic analysis of the values for various metrics has been carried out and several key inferences have been drawn from them. A number of useful conclusions have been drawn from various metrics evaluations, which include inferences on complexity, reusability, testability, modularity and stability of the underlying components. The inferences are argued to be beneficial for CBSE-based software development, integration and maintenance.

Keywords: CBSE metrics, software integration, software reusability, software maintenance.

Introduction

Component-Based Software Engineering (CBSE) is a methodology that emphasizes the design and construction of computer–based systems using reusable software components. This principle embodies an element of "buy, don't build" that shifts the emphasis from programming software to composing software systems (Pressman, 2001). It is also an approach for developing software that relies on software reuse and it emerged from the failure of object-oriented development to support effective reuse. The behavior and the stability of an application cannot be assessed unless it is tested comprehensively. The quality of the application is high when it yields the expected results, is stable and adaptable and leads to reduce maintenance costs. If a change has been introduced in a component, which has been integrated in an application, the impact of the change on the whole application has to be determined by the developer to assess the stability of the application. Consequently, there is certainly a need to measure quality and assess the component's impact on the overall system. Metrics are needed to measure several types of quality issues. Metrics are also needed to study the characteristics of a given software system under different scenarios (Ali & Ghafoor 2001 Bertoa, Troya, & Vallecillo 2003 Lorenz & Kidd 1992). Most of the existing metrics are applicable to small programs or components (Kan 2002), while the objective of CBSE metrics is to evaluate the behavior and reliability of the component when integrated into a large software

Material published as part of this publication, either on-line or in print, is copyrighted by the Informing Science Institute. Permission to make digital or paper copy of part or all of these works for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage AND that copies 1) bear this notice in full and 2) give the full citation on the first page. It is permissible to abstract these works so long as credit is given. To copy in all other cases or to republish or to post on a server or to redistribute to lists requires specific permission and payment of a fee. Contact Publisher@InformingScience.org to request redistribution permission.

system. Consequently (Weyuker, 1998), the lack of appropriate mathematical properties fails quality metrics. Metrics that have a sound theoretical basis become applicable to real life organizations (Pfleeger & Fenton, 1998). Some of the metrics rely on parameters that could never be measured or are too difficult to measure in practice. Since a component's internal structure may not be available, there is a need for black box testing and a number of existing metrics may not be applicable directly.

A software component is a coherent package of software implementation that offers well-defined and published interfaces, is reusable and that can be independently developed and delivered; such components are put together to form an application. However, there are no good metrics available to validate their effectiveness, when components are integrated together to form a complete system. Due to the inherent differences in the development of component based and non-component based systems, the traditional software metrics prove to be inappropriate for component-based systems. The component metrics alone are not sufficient for an integrated environment, because there is a need to measure the stability and adaptability of each component when it is integrated with other components.

Narasimhan and Hendradjaya (2007) noticed the lack of metrics that aids in reducing the maintenance costs and defined metrics whose values are collected during the execution phase. Such metrics are useful for assessing the maintenance cost of individual components and that of the application in which the component is integrated. This paper supports and critiques the ideas of Narasimhan and Hendradjaya in providing metrics for the integration of software components. A component when executed may yield the expected results, but its behavior and functionality, when integrated with other components to make a complete application, may yield unexpected results. Therefore, there is a need for metrics to assess the functionality of each component when integrated with other components and functionality of the application on the whole. The paper provides a comparison of various metrics and observes several views on the traditional metrics and the metrics proposed by Narasimhan and Hendradjaya are useful in assessing the quality of components in an integrated application. Benchmarks software programs have been used as inputs to instrumentation programs and metric values have been collected. A systematic analysis of the values for various metrics (Chidamber & Kemerer, 1994; Cho, Kim, & Kim, 2001) has been carried out and several key inferences have been drawn from them. Inferences from this work certainly provide relative comparisons on complexity, reusability, testability, modularity and stability of the underlying components. Finally, we show that the inferences drawn from this work are beneficial for CBSE-based software development, integration and maintenance.

The rest of the paper is organized as follows: the next section provides related works on integration of software components, while the third section provides a comparison of suites of metrics proposed by three different researchers. The fourth section details the design of the metric evaluation system and the fifth section describes the instrumentation programs used to evaluate a CBSE software systems. The sixth section describes the nature of the benchmark suites selection, and inferences from the various suites of metrics analyzed. The final section concludes the paper and offers some pointers for further research in this area.

Metrics on the Integration of Software Components

Narasimhan and Hendradjaya (2007) classified their metrics into complexity, criticality, triangular and dynamic metrics. Since this paper is in-part an evaluation of their metrics and comparison with other metrics, the reader is encouraged to read the original papers that define the various metrics (see Chidamber & Kemerer, 1994; Cho, Kim, & Kim, 2001; Narasimhan & Hendradjaya, 2007).

Comparison of the Three Metric Suites

A comparison of metrics provides a basis for choice of selection of a particular type of metric and this has been made on such issues as, reusability, complexity, size, testing time and maintenance. The metric values are compared using benchmark software programs. While many authors have provided a comparison of metrics, their focus has been on collecting the metric values for a component considered as a stand-alone entity (Bertoa et al., 2003; Henderson-Sellers, 1996; Lorenz & Kidd, 1992). In this paper, we collected metric values for sub-components that make a system and evaluated the best suite of metrics that suit a given context/system. Three sets of metrics that are currently in use are tabulated in Table 1. The behavior of a metric is theoretically analyzed based on their definitions provided by the corresponding authors. A metric is considered as suitable for a given quality factor, if its value is significant to the particular factor (Boehm et al., 2000).

Table 1: Comparison of various metrics

Metrics	Author(s)	Strengths & Limitations
WMC, RFC, LCOM, CBO, DIT, NOC	Chidamber & Kemerer, 1994	Broad indicator, but lack specificity
CPC, CSC, CDC, CCC	Cho, Kim & Kim, 2001	Narrow indicator
CPD, CID, CIID, COID, CAID, $CRIT_{link}$, $CRIT_{bridge}$, $CRIT_{inheritance}$, $CRIT_{size}$, $CRIT_{all}$, Triangular metrics, ANAC, ACD, AACD.	Narasimhan & Hendradjaya, 2007	Covers a broad set of issues

Metrics behavior under the criteria reusability

The metrics mentioned in Table 2 measure reusability of a component. A high LCOM, NOC, and DIT implies that the corresponding components are highly reusable. A high CID, CPD, WMC, CSC, and CBO, implies that the corresponding components are less reusable. A low $CRIT_{Size}$, $CRIT_{Link}$ implies that the corresponding components are highly reusable. It is noted that a component is considered good, if it is highly reusable (Browne, Werth, & Lee, 1990; Washizaki, Yamamoto, & Fukazawa, 2003).

Table 2: Relative values for metrics ideal for Resusability Quality factor

Name of metric	Relative value of metric	Implication for Reusability
LCOM	increases	Increases
WMC	increases	Decreases
CBO	increases	Decreases
NOC	increases	Increases
DIT	increases	Increases
CSC	increases	Decreases
CPD	increases	Decreases
CID	increases	Decreases
CAID	increases	Decreases
$CRIT_{Size}$	decreases	Increases
$CRIT_{Link}$	decreases	Increases

Table 3: Relative values for metrics ideal for the quality factor - Complexity

Name of metric	Relative value of metric	Implication for Complexity
RFC	increases	Increases
CBO	increases	Increases
LCOM	decreases	Increases
DIT	increases	Increases
CPC	increases	Increases
CPD	increases	Increases
CID	increases	Increases
CAID	increases	Increases
CIID	increases	Increases
COID	increases	Increases

Metrics behavior under the criteria complexity

The metrics mentioned in Table 3 measure the complexity of a component. A high value for the metrics RFC, CBO, DIT, CPD, CIID, COID, and CPC, imply that the corresponding component

is considered to be highly complex. A low value for the metric LCOM implies that the corresponding component is considered highly complex. It is noted that a component is considered ideal if it is less complex and hence the values of the metrics like RFC, CPC, CIID and COID are to be very low.

Metrics behavior under the criteria testability

The metrics provided in Table 4 measure the testability of a component. A high value for the metrics NOC, CID, $CRIT_{Bridge}$, $CRIT_{Link}$ and RCC imply high testability. An ideal application made up of components should take a short time-to-test.

Table 4: Relative values for metrics ideal for the quality factor - Testability

Name of metric	Relative value of metric	Implication for Testability
NOC	Increases	Increases
RCC	Increases	Increases
CID	Increases	Increases
$CRIT_{Bridge}$	Increases	Increases
$CRIT_{Link}$	Increases	Increases

Table 5: Relative values for metrics ideal for the quality factor -Modularity

Name of metric	Relative value of metric	Implication for Modularity
WMC	increases	Increases
CPC	increases	Increases
$CRIT_{Inheritance}$	increases	Increases
$CRIT_{Size}$	increases	Decreases
AC	increases	Increases
NOC	increases	Decreases

Metrics behavior under the criteria maintenance

The list of metrics, whose values can be used to infer the maintenance effort required for a given application, is: ANAC, CCC, NC and ACD. The metrics values for these metrics are collected during run-time which implies that the development phase of the components has been completed and that, these metrics are being collected for maintenance purposes. Narasimhan and Hendradjaya (2007) have proposed a series of dynamic metrics for the purpose of maintenance.

Metrics behavior under the criteria modularity

The metrics mentioned in Table 5 measure several aspects on the size of a component. A high value for the metrics WMC, and CPC, implies a large component. If $CRIT_{Size}$ is high, it means the component has letter degree of modularity. A high value for the metrics $CRIT_{Inheritance}$ and NOC, imply that the corresponding component is considered to less reusable. It is noted that a component is considered good, if it is has an appropriate size so as to make it less complex and highly reusable.

Software Architecture of Metric Evaluation System

The software architecture of metric evaluation system is provided in Figure 2. The system has the following six major components:

- Benchmark suite that contains programs, whose source/object codes are used for metric generation
- Instrumentation program suite that facilitates collection metric values from the benchmark programs
- Compiler that takes the benchmark suite as input for the instrumentation program, compiles and executes

- Metric values generator, which is the output of the instrumentation program, that gives the metric values for the benchmark software
- Inferences engine is the place at which inferences are made from the various metric values
- Metrics visualization environment

Various benchmark software selected on the basis of some criteria is given as inputs to the instrumentation program which, when compiled and executed by the compiler, gives the metric values as outputs. Inferences are made based on the outputs and the theoretical analysis and the best matched metrics suite for a given context is concluded.

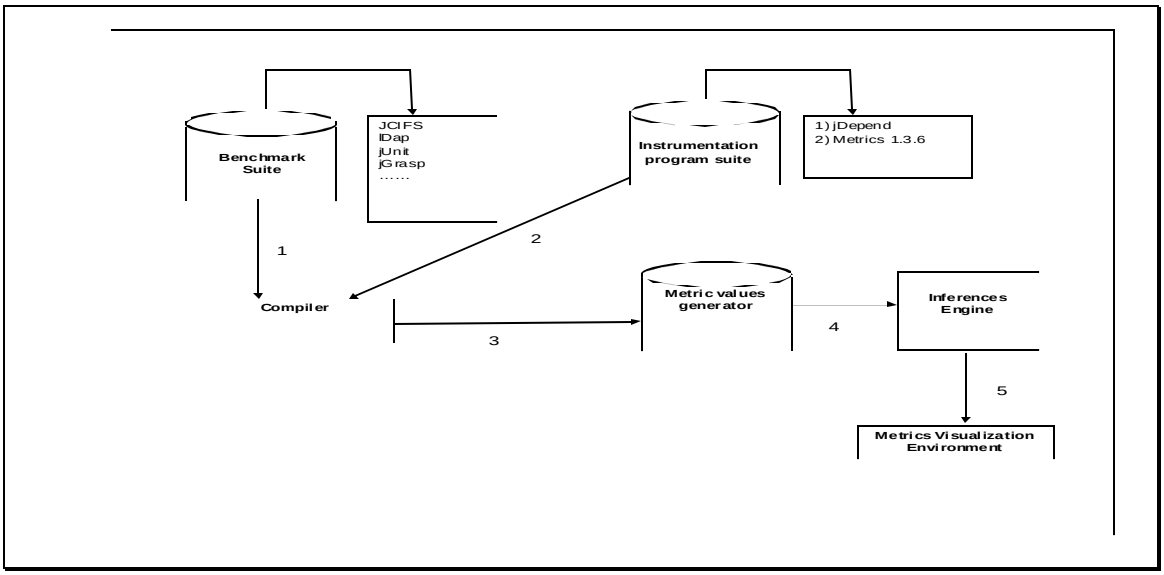

Figure 2: Software architecture of metric evaluation system

Instrumentation Programs

Instrumentation programs concern a set of programs or tools used for collecting metrics from various benchmark software systems. The instrumentation programs provide output in the form of data units, which might be a direct or indirect representation of some of the metrics of the three suites considered. If the data units are in indirect form, the required calculations/transformations are performed by the authors.

jDepend (jDepend, 2007) and Metrics 1.3.6 (Metrics 1.3.6 2007) are the instrumentation programs used to facilitate data collection. JDepend software is used to collect data for the following metrics: CRIT Inheritance, CRIT Size, COID, CAID, and CIID. In the developer's own words, "*JDepend traverses Java class file directories and generates design quality metrics for each Java package*". JDepend allows automatic measurement of the quality of a design in terms of its extensibility, reusability, and maintainability to manage package dependencies effectively. The output of the software is the following units.

Metrics 1.3.6 software is used to collect values for the following metrics directly or indirectly: NOC, WMC, DIT, CPD and LCOM. Metrics 1.3.6 provides metrics calculation and dependency analyzer plug-in for the Eclipse platform. It measures various metrics with average and standard deviation, detects cycles in package and type dependencies and provides a graphical visualization. This package is operating system independent developed for the Java programming language.

The following procedure has been adopted to collect metric values from the output of the instrumentation programs:

a) The outputs of the JDepend software provides values for the metrics: COID, CIID, $CRIT_{Inheritance}$, $CRIT_{size}$ over a (Java collection) data structure.

b) Values for the metric CAID, CID and COID are calculated manually based on the definition of the CAID. The indirect values for CIID and COID have been obtained from the software outputs.

c) From the output of Metrics 1.3.6, values for the metrics NOC, DIT, LCOM and WMC have been collected directly.

d) CPD is calculated by considering the mean value of the *Number of Classes* for each benchmark program.

Benchmarking the Metrics

This section provides an overview of the software packages which are used as source inputs for collecting values of the various metrics. A good benchmark has been empirically defined (based on observations of several software systems) as a software containing at least 50 classes, and 15,000 lines of code and further, the code has is to be available over any object-oriented language. In this paper, the emphasis is on preserving the properties of CBSE in an integrated environment such that the application yields the expected results. Therefore, if the value of a metric determines the component for which the metric value has been calculated as stable, reusable, more abstract and less complex, then that benchmark is considered stable. Otherwise, some of the components in the benchmark may need to be re-designed. Several benchmark packages of different sizes and varying modules have also been considered in this work. The following criteria have been used in the selection benchmark software:

(i) Code should be object–oriented: The package code is to be written in any object–oriented language.

(ii) The size of package: The size of the package should be large enough to depict a practical scenario, i.e., the packages with at least 20,000 lines of code are considered. The number of classes should be at least 50.

(iii) Transparency of source code: Packages for which the source code is not transparent are selected for black-box testing and reuse. By the definitions of CBSE, the complete source code of a component may not be available for any developer while reusing the component. Therefore, to depict the real-life scenario, packages with object codes have only been considered.

Inferences from the Results

Table 6 provides a snapshot of the characteristics of the chosen benchmark software programs. Table 7 provides the values of various metrics for the six benchmark programs chosen. Inferences are listed below based on the theoretical definitions and the metric values collected. The best suite of metrics that matches the context of measuring the integrated components has also been provided. Among the three suite of metrics used for comparison purpose, we chose the best ones matching the context of measuring the integrated components in order to measure various values.

Table 6: Characteristics of benchmark software programs

Benchmark programs	No. of classes	No. of sub-components
lDap	339	16
JCIFS	141	8
jGrasp	1265	18
jUnit	107	8

Table 7: Table of metric values

Metrics	Junit	Element	mouseGestures	Idap	JCIFS	jGrasp
CPD	13.375	19	4.5	6.25	0.5	70.7
CID	61	6	11	114	70	204
CIID(Ce)	315	6	10	89	51	159
COID(Ca)	91	1	1	25	19	45
CAID(CID/8)	7.625	6	5.5	7.125	8.75	11.34
CRIT Inheritance	93	19	8	91	4	1142
CRIT size	0	0	0	0	0	1
AC(=CID)	61	6	11	114	70	204
NOC	16	4	0	22	18	1
LCOM	0.91	0.855	0.778	0.627	0.753	0
DIT	6	4	6	8	7	1
WMC	822	407	46	763	539	37

- *Inferences from the CIID metric*: A high value for the CIID metric implies that the complexity is high. *lDap, JCIFS , mouseGestures* and *Element* packages have comparably low values.
- *Inferences from the COID metric*: A high value for the COID metric implies that the complexity of the component is relatively high. *Mouse Gestures, lDap, JCIFS* and *jGrasp* have low COID values, which implies that their complexity is less.
- *Inferences from the CAID metric*: A high value for the CAID metric implies that the reusability property of the component decreases. The complexity of the component is considered high thus increasing the effort of testability. CAID metric value is relatively small for all the benchmarks considered.
- *Inferences from the $CRIT_{Inheritance}$ metric*: A high value for the $CRIT_{Inheritance}$ metric implies a highly modular component; high modularity makes a component more reusable. *lDap, jUnit* and *jGrasp* are considered to be highly modular.
- *Inferences from the $CRIT_{Size}$ metric*: If this value is high, it means it is less modular and hence less reusable. For the benchmarks, the metric values are within the threshold value except for *jGrasp*.
- *Inferences from the AC metric*: A high value for the AC metric implies that the component is highly modular. *lDap* and *jGrasp* are considered highly modular.
- *Inferences from the NOC metric*: A high value for the NOC metric implies that the component is highly reusable, but testability effort is relatively high. *lDap, JCIFS,* and *jUnit* are relatively highly reusable.
- *Inferences from the CPD Metric*: From the theoretical analysis, if a high value for the CPD metric implies that the reusability decreases. Among the considered benchmarks, *lDap, JCIFS* and *mouseGestures* have low CPD values and hence highly reusable.
- *Inferences from the CID metric*: A high value for the CID metric implies that the reusability decreases. Further, the time taken for testing and component complexity is high. *MouseGestures* and *Element* packages have low CID values, thereby having high reusability and less complexity.

- *Inferences from the LCOM metric*: A high value for the LCOM metric implies that the reusability of that component is high and the component is relatively less complex. *jUnit, Element* and JCIFS have very high values. It is inferred that all the packages are relatively reusable except *jGrasp*.
- *Inferences from the DIT metric*: If the value of DIT is high, reusability is high and complexity is high (Chidambar & Kemerer, 1994). *lDap, JCIFS, mouseGestures* and *jUnit* are highly reusable and highly complex. This metric value and the inferences indicate that these kinds of metrics are not efficient at measuring the CBSE qualities, as they consider the value of the entire application as a single component.
- *Inferences from the WMC metric*: If the value of WMC is high, reusability is considered low. The packages *lDap, JCIFS, jUnit* are less reusable. This metric value and the inferences indicate that these kinds of metrics are not efficient at measuring the CBSE qualities, as they consider the value of the entire application as a single component.
- *Inferences on $CRIT_{Instability}$ Metric:* The values for $CRIT_{Instability}$ for the benchmarks, calculated using the formula as defined by Martin (1995), is given in Table 8.

Table 8: Values of $CRIT_{Instability}$ for different benchmark software programs

Benchmark	Value for the metric
lDap	2
jGrasp	7
JUnit	6
JCIFS	2
mouseGestures	0
Element	0

From the values quoted in Table 7, it is inferred that the some of the components of jGrasp and JUnit need to be redesigned.

Discussion and Conclusions

This paper provides a systematic comparison of three suites of metrics. The comparisons (Table 7 & 8) allow a user to choose the best applicable metric based on their particular requirements. The metric values provided are helpful to study the behavior of metrics under various quality factors. The metrics defined by Chidamber and Kemerer (1994) considered in this paper (WMC, NOC, DIT and LCOM) do not measure quality of integrated components. The metrics may be applicable to analyze reusability, complexity and size indirectly, but they are not sufficient to measure testing time and maintenance. The metrics defined by Cho et al. (2001) are CPC, CSC, CDC, and CCC and these metrics prove deficient for black- box testing. These metrics deal with the complexity of the code, which requires the availability of the entire source code. Since Cho metrics calculate the complexity of metrics by using the combination of the number of classes, and interfaces, the calculation of cyclomatic complexity with the sum of classes and interfaces needs information from the source code, which is a shortcoming. This proves successful only if the developer has access to the source code. Narasimhan and Hendradjaya (2007) metrics test to check if any incorrect operations are not inherited by the subcomponents. Dynamic metrics measure maintenance and testing issues as a consequence of execution of the code. The metrics measure several aspects, such as reusability, complexity, testing-time, size and maintenance. The formulas provided for each metric considers the average of the subcomponents rather than a single component, thus extending them to an integrated environment also. The suite of metrics proposed by Narasimhan and Hendradjaya prove to be efficient at measuring the quality of integrated components. However, there are some limitations that restrict the use of this suite of metrics, which are discussed below: The lack of threshold values restricts the suite of metrics theoretically hindering its use practically. Narasimhan and Hendradjya have intuitively defined the values using representative software, but an accurate threshold value calculated by quantifying and testing more empirical dataset is necessary. Criticality of the metrics means the limitations that stop the use of met-

rics for practical purposes and requires immediate solutions. Future research can be carried out in the following directions: 1) Collecting metric values for further benchmarks to study metric behavior, 2) Revising the formulas used for calculation of metrics for greater accuracy and 3) Setting appropriate Threshold values.

References

Ali, S. S., & Ghafoor, A. (2001). Metrics-guided quality management for component-based software systems. *P*roceedings of the 25th Annual International Computer Software and Applications Conference, pp.303-308.

Bertoa, M. F., Troya, J. M., & Vallecillo, A. (2003). A survey on the quality information provided by software component vendors. *Proceedings of the 7th ECOOP Workshop on Quantitative Approaches in Object – Oriented Software Engineering* , pp.25- 30.

Browne, J. C., Werth, J., & Lee, T. (1990). Experimental evaluation of a reusability-oriented parallel programming environment. *IEEE Transactions on Software Engineering,16*(2), 111-120.

Boehm, B.W. et al. (2000). *Software cost estimation with COCOMO II.* Prentice Hall Edition.

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object-oriented design. *IEEE Transaction on Software Engineering, 20*(6), 476-493.

Cho, E. S., Kim, M. S., & Kim, S. D. (2001). Component metrics to measure component quality. *The 8th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 419- 426.

Henderson-Sellers, B. (1996). *Object-oriented metrics: Measures of complexity*. Prentice Hall.

JDepend. (n.d.). Available 10 September, 2007 at http://clarkware.com/software/JDepend.html

Kan, S. H. (2002). *Metrics and models in software quality engineering*. Addison-Wesley Professional.

Lorenz, M., & Kidd, J. (1992). *Object–oriented software metrics: A practical guide*. Engle wood Cliffs, N.J: Prentice-Hall.

Martin, R. (1995). *Designing object-oriented C++ applications using the Booch method*. Englewood Cliffs, NJ: Prentice Hall.

Metrics 1.3.6. (n.d.). Available 10 September, 2007 at http://metrics.sourceforge.net/

Narasimhan, V. L., &.Hendradjaya, B. (2007). Some theoretical considerations for a suite of metrics for the integration of software components. *Information Sciences: An International Journal, 177*(3), 844-864.

Pfleeger, L. S., & Fenton, N. E. (1998). *Software metrics: A rigorous and practical approach*. Brooks/Cole Publications.

Pressman, R. S. (2001). *Software engineering: A practitioner's approach*. McGraw Hill.

Washizaki, H., Yamamoto, H., & Fukazawa, Y. (2003). A metrics suite for measuring reusability of software components. *Proceedings of the International***,** pp. 211- 223,.

Weyuker, E. J. (1998). Testing component–based software: A cautionary tale. *IEEE software, 15*(5), 54 – 59.

Biographies

Narasimhan is a Professor of Software Engineering in the Department of Computer Science at East Carolina University, USA. He has published over 180 papers in the areas of Software Engineering and Information Engineering. In particular, his research interests are in computer architecture, parallel and distributed computing, software testing, text & audio processing and mining, E-Commerce, Software process, asset management systems and Standards, and information management & fusion. His papers have appeared in such archival journals and international conferences. Prof. Narasimhan is a Senior Member of the IEEE and ACM, Fellow of ACS, IEAust and IEE (UK). He is a Technical Member (representing USA) of the Expert Panel of ISO (International Standards Organization) and ANSI.

Miss. Prapanna Parthasarathy obtained her BS in Computer Science from Sri Venkateswara University, India, and MS in Computer Science from Western Kentucky University, USA. Parthasarathy is currently working for Lexmark Printers Inc. at Louisville, KY, USA. Her research interests are in the areas of component based software engineering, software metrics and testing.

Manik Lal Das received his Ph.D. degree from Indian Institute of Technology, Bombay in 2006. Currently he is an Assistant Professor at Dhirubhai Ambani Institute of Information and Communication Technology, Gandhinagar, India. He has over 11 years of R&D and teaching experience in Computer Science. He has published over 40 research articles in refereed Journals/Conferences. He is a member of the IEEE, Cryptology Research Society of India and Indian Society for Technical Education. His research interests include Cryptology and Information Security.