01 May 2001

Software Engineering Metrics for COTS-Based Systems

Sahra Sedigh
Missouri University of Science and Technology, sedighs@mst.edu

Arif Ghafoor

Raymond A. Paul

Recommended Citation

S. Sedigh et al., "Software Engineering Metrics for COTS-Based Systems," *Computer*, vol. 34, no. 5, pp. 44-50, Institute of Electrical and Electronics Engineers (IEEE), May 2001.
The definitive version is available at https://doi.org/10.1109/2.920611

Software Engineering Metrics for COTS-Based Systems

The growing reliance on commercial off-the-shelf components for large-scale projects emphasizes the need for adequate metrics to quantify component quality.

Sahra Sedigh-Ali
Arif Ghafoor
Purdue University

Raymond A. Paul
US Department of Defense

The paradigm shift to commercial off-the-shelf components appears inevitable, necessitating drastic changes to current software development and business practices. Quality and risk concerns currently limit the application of COTS-based system design to noncritical applications. New approaches to quality and risk management will be needed to handle the growth of CBSs.

Our metrics-based approach and software engineering metrics can aid developers and managers in analyzing the return on investment in quality improvement initiatives for CBSs. These metrics also facilitate the modeling of cost and quality, although we need more complex models to capture the intricate relationships between cost and quality metrics in a CBS.

COTS COMPONENTS

With software development proceeding at Internet speed, in-house development of all system components may prove too costly in terms of both time and money. Large-scale component reuse or COTS component acquisition can generate savings in development resources, which can then be applied to quality improvement, including enhancements to reliability, availability, and ease of maintenance.

Prudent component deployment can also localize the effects of changes made to a particular portion of the application, reducing the ripple effect of system modifications. This localization can increase system adaptability by facilitating modifications to system components or integration code, which are necessary for conforming to changes in requirements or system design.

COTS component acquisition can reduce time to market by shifting developer resources from component-level development to integration. Increased modularity also facilitates rapid incremental delivery, allowing developers to release modules as they integrate them and offer product upgrades as various components evolve.

These advantages bring related disadvantages, including integration difficulties, performance constraints, and incompatibility among products from different vendors. Further, relying on COTS components increases the system's vulnerability to risks arising from third-party development, such as vendor longevity and intellectual-property procurement. Component performance and reliability also vary because component-level testing may be limited to black-box tests, and inherently biased vendor claims may be the only source of information.[1]

Such issues limit COTS component use to noncritical systems that require low to moderate quality. Systems that require high quality cannot afford the risks associated with employing these components.

METRICS FOR COTS-BASED SYSTEMS

In deciding between in-house development and COTS component acquisition, software engineers must consider the anticipated effect on system quality. We can define software quality in several ways:

- satisfaction level—the degree to which a software product meets a user's needs and expectations;
- a software product's value relative to its various stakeholders and its competition;

0018-9162/01/$10.00 © 2001 IEEE

- the extent to which a software product exhibits desired properties;
- the degree to which a software product works correctly in the environment it was designed for, without deviating from expected behavior; and
- the effectiveness and correctness of the process employed in developing the software product.[2]

Quality factors and quality metrics

Norman Schneidewind discusses quality in terms of quality factors and quality metrics.[3] He defines a quality factor as "an attribute of software that contributes to its quality, where quality is the degree to which software meets customer or user needs or expectations." For example, Schneidewind mentions reliability as a quality factor. Direct measurement of quality factors is generally not feasible, so we often measure them indirectly—for example, by counting the number of failures reported for a particular module.

Schneidewind defines a quality metric as "a function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that may affect its quality." In contrast with quality factors, which are user-oriented, quality metrics are developer-oriented because developers can use them to estimate quality at a very early stage in the software development process. Before using metrics for design or integration decisions, software engineers should validate them, establishing a statistical relationship between metrics and quality factors and ensuring that the metrics provide a correct estimate of the attribute visible to the user.

In addition to traditional software metrics, COTS-based systems require metrics that capture attributes such as integration complexity and performance. Combining component-level metrics to obtain system-level indicators of quality is a challenging issue that is further complicated by COTS components' black-box nature, which masks their internal workings and restricts system developers to accessing their interfaces.

To be thorough, we should test the operational profiles of both the COTS components with respect to both their own operational profiles and that of the overall system. But when inaccessibility of the source code for some components prevents such comprehensive testing, we can use metrics to guide the software development process. In addition to metrics data, certain aspects of the software product impact and guide software development decisions:

- the system's expected functionality and the customer's requirements;
- the makeup of the various organizations involved in the project and the level of maturity and capabilities of the participating teams;
- the developers' use of innovative processes and the methods they adopt as a part of the software engineering environment to manage cost and value, including details of development process models such as the waterfall or spiral models; and
- features of the preexisting COTS components that the system will use.

The unpredictable quality of third-party software creates a unique set of risks for software systems using COTS components.

Risk management

The unpredictable quality of third-party software creates a unique set of risks for software systems using COTS components. The CBS development process, then, should include risk management, which identifies high-risk items that can jeopardize system quality and attempts to resolve them as early as possible to ensure high quality and rapid delivery. The two major steps in risk management are

- *risk assessment:* assess the probability and magnitude of loss for each risk item and prioritize risk items according to their expected loss; and
- *risk control:* generate and execute plans to resolve the risk items.

Developers apply these two steps repeatedly throughout the software development life cycle.[4]

In CBSs, risk management focuses on evaluating alternative components that meet system requirements, either selecting the component that fits best or choosing in-house development. In either of these tasks, developers or integrators can decide to relax the requirements to allow a particular choice, using risk management to determine the extent of tolerable relaxation.[5]

Risk and quality-management metrics

Metrics can guide risk and quality management, helping to reduce risks encountered during planning and execution of software development, resource and effort allocation, scheduling and execution, and product evaluation.[4] Risks can include performance issues, reliability, adaptability, and return on investment. Risk reduction can take many forms, such as using component wrappers or middleware, replacing components, relaxing system requirements, or even issuing legal disclaimers for certain failure-prone software features. Metrics let developers identify and isolate these risks, then take corrective action.

The key to success is selecting appropriate metrics—especially metrics that provide measures applicable over the entire software cycle and that address both software processes and products. In choosing metrics, developers should consider several factors:

Table 1. System-level metrics for component-based systems.

Category	Metric	Evaluates or measures
Management	Cost	Total software development expenditure, including costs of component acquisition, integration, and quality improvement
	Time to market	Elapsed time between development start and component acquisition to software delivery
	Software engineering environment	Capability and maturity of the environment in which the software product is developed
	System resource utilization	Use of target computer resources as a percentage of total capacity
Requirements	Requirements conformance	Adherence of integrated product to defined requirements at various levels of software development and integration
	Requirements stability	Level of changes to established software requirements
Quality	Adaptability	Integrated system's ability to adapt to requirements changes
	Complexity of interfaces and integration	Component interface and middleware or integration code complexity
	Integration test coverage	Fraction of the system that has undergone integration testing satisfactorily
	End-to-end test coverage	Fraction of the system's functionality that has undergone end-to-end testing satisfactorily
	Fault profiles	Cumulative number of detected faults
	Reliability	Probability of failure-free system operation over a specified period of time
	Customer satisfaction	Degree to which the software meets customer expectations and requirements

- the intended use of the metrics data;
- the metrics' usefulness and cost-effectiveness;
- the application's functional characteristics, physical composition, and size;
- the installation platform;
- the software engineering environment of the development phase;
- the software engineering environment of the integration phase; and
- the software development or maintenance life-cycle stage of both the components and the system.[6]

Table 1 shows our set of 13 system-level metrics for CBS software engineering. These metrics help managers select appropriate components from a repository of software products and aid in deciding between using COTS components or developing new components. The primary considerations are cost, time to market, and product quality.

We can divide these metrics into three categories: management, requirements, and quality.

Management. These metrics include cost, time to market, system resource utilization, and software engineering environment. Developers can use management metrics for resource planning or other management tasks or for enterprise resource planning applications.

- The cost metric measures the overall expenses incurred during the course of software development. These expenses include the costs of component acquisition and integration and quality improvements to the system.
- The time-to-market metric measures the time needed to release the product, from the beginning of development and COTS component acquisition to delivery. A modified version of this metric can evaluate the speed of incremental delivery, measuring the amount of time required to deliver a certain fraction of the overall application functionality.
- The software engineering environment metric measures the capability of producing high-quality software and can be expressed in terms of the Software-Acquisition Capability Maturity Model.[7]
- System resource utilization determines the percentage of target computer resources the system will consume.

Requirements. Developers use requirements metrics to measure the CBS's conformance and stability so they can monitor specifications, translations, and volatility, as well as the level of adherence to the requirements. COTS components are often unstable, and component-level stability can affect requirements stability if developers adapt requirements to incorporate changes to selected components.

Quality. These metrics include adaptability, complexity of interfaces and integration, integration test coverage, end-to-end test coverage, reliability, and customer satisfaction.

- Adaptability measures a system's flexibility, evaluating its ability to adapt to requirements changes, whether as a result of system redesign or to accommodate multiple applications.
- Complexity of interfaces and integration provides an estimate of the complexity of interfaces, middleware, or glue code required for integrating different COTS products. Overly complex interfaces complicate testing, debugging, and maintenance, and they degrade the system's quality.
- Integration test coverage and end-to-end test coverage indicate the fraction of the system's functionality that has completed those tests, as well as the effort testing requires.[8] Developers can use known measures to evaluate coverage, such as statement or path coverage, depending on the level of access to system source code.

- Reliability estimates the probability of fault-free system operation over a specified period of time. To obtain this metric, developers use techniques similar to the techniques they use in traditional systems, including fault injection into the integration code.
- Customer satisfaction evaluates how well the software meets customer expectations and requirements. Beta releases can help estimate predictors of customer satisfaction before final product delivery. Sample predictors include schedule requirements, management maturity, customer culture, marketplace trends, and the customer's proficiency. Such estimates can guide development decisions such as release scheduling and can aid in developing a test plan that accurately reflects the product's field use.

CBS metrics differ from traditional metrics in that they do not depend on the components' code size, which is generally not known. If developers require a size measure, they can use alternate measures such as the number of use cases—business tasks the application performs—that a given component supports.

CBS metrics also approach time to market differently. Component acquisition changes the concept of time to market because developers may not know the component development time and cannot incorporate it into time calculations. For CBSs, a simple delivery rate measure can replace the time-to-market measure. One proposed measure divides the number of use cases by the elapsed time in months.[9]

Because our metrics are interdependent, understanding the relationships between them can aid decision making regarding CBS quality-improvement investments. The most obvious relationship is between cost and quality metrics, such as reliability. However, more subtle relationships exist, such as among time to market, test coverage, and reliability. Delayed product release because of testing and debugging can result in reduced revenues or, in extreme cases, loss of the market to a competitor with an earlier release. On the other hand, premature product release can lead to lower reliability. Understanding the relationships among time to market, test coverage, and reliability can help in selecting a suitable release schedule.

Developers can combine the cost metric and the system resource utilization metric to determine whether the budget allows purchasing additional computer resources that will enhance the product's quality. Another effective strategy involves using the software engineering environment in conjunction with the quality metrics to encourage vendors to improve their software development process and adhere to standards, thus increasing the likelihood that users will select their component.

Table 2. Software quality cost categories.

Category	Typical costs of CBS software quality
Appraisal costs	Integration or end-to-end testing, quality audits, component evaluation, metrics collection, and analysis
Prevention costs	Training, software design reviews, process studies, component upgrades
Internal failure costs	Defect management, design and integration rework, component replacement, requirement relaxation
External failure costs	Technical support, maintenance, defect notification, remedial component upgrade or replacement

COST OF QUALITY

The cost of quality (CoQ) represents the resources dedicated to improving the quality of the product being developed. For example, increasing or maintaining reliability incurs costs that can be considered the costs of reliability. The overall CoQ is the sum of such costs plus other costs that we cannot directly attribute to factors that quality metrics measure. Quality costs, then, represent "the difference between the actual cost of a product or service and what the reduced cost would be if there were no possibility of substandard service, failure of products, or defects in their manufacture."[2]

We concern ourselves with the cost of software quality (CoSQ) metric—corresponding to the cost metric in Table 1—which we divide into two major types: cost of conformance and cost of nonconformance.

The cost of conformance derives from the amount the developer spends on attempts to improve quality. We can further divide conformance costs into prevention and appraisal costs. Projects incur prevention costs during activities targeted at preventing defects, such as training costs, software design reviews, and formal quality inspections. Likewise, activities that involve measuring, evaluating, or auditing products to assure conformance to quality standards and performance incur appraisal costs. These activities include code inspections, testing, quality audits, and software measurement activities such as metrics collection and analysis.

The cost of nonconformance includes all expenses the developer incurs when the system does not operate as specified. Internal failure costs stem from nonconformance occurring before the product ships to the customer, such as the costs of rework in programming, defect management, reinspection, and retesting. External failure costs arise from product failure after delivery to the customer. Examples include technical support, maintenance, remedial upgrades, liability damages, and litigation expenses.

Table 2 shows the various categories of software quality costs for CBSs.

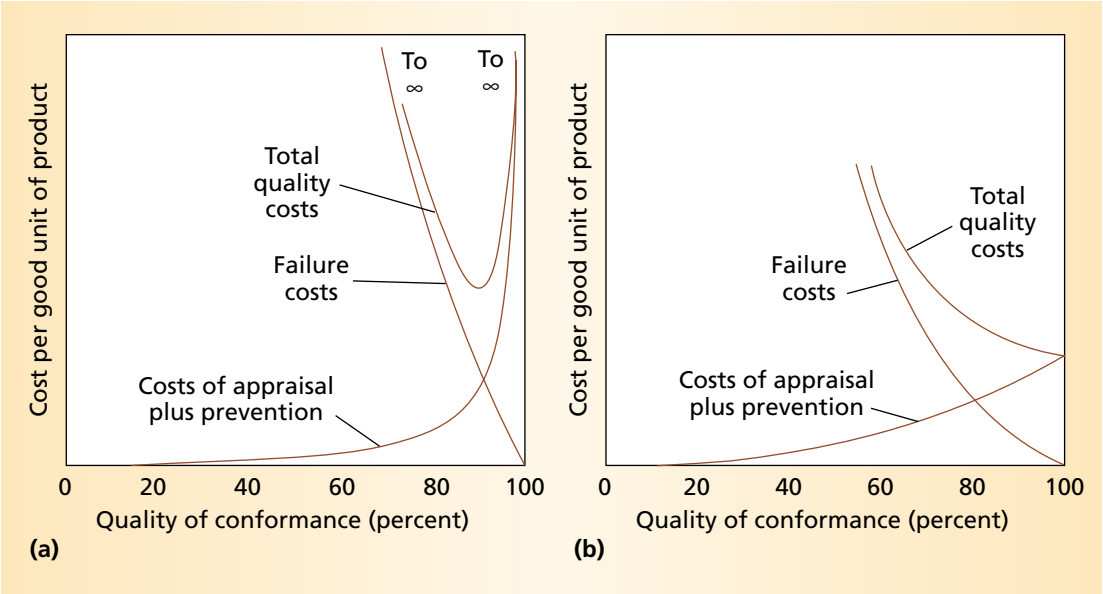

Figure 1. Models depicting the relationship between costs and quality. (a) The optimum quality cost model shows the relationship between cost per good unit of product and quality of conformance. (b) The revised model benefits from technological developments that reflect the ability to achieve very high quality at finite cost. Figures adapted from Campanella.[2]

CoQ and CoSQ models

In any development process, models that depict the relationship between costs and quality can guide decisions regarding investments in quality improvement. Discussions of such models in the economics and management literature generally depict a nonlinear relationship between CoQ and quality.[2] Accurate cost-quality models can be invaluable to managers and developers, guiding resource and cost management and other aspects of the software development process.

Figure 1a depicts the classic model of optimum quality costs. In this model, which shows the relationship between the cost per good unit of product and the quality of conformance, expressed as a percentage of total conformance, prevention and appraisal costs rise asymptotically as the product achieves complete conformance.

Recent technological developments inspired a revised model that reflects the ability to achieve very high quality, or "perfection," at finite costs. Shown in Figure 1b, this model, proposed by Frank Gryna, has two key concepts:

- moderate investments in quality improvement result in a significant decrease in the cost of nonconformance, and
- focusing on quality improvement by defect prevention results in an overall decrease in the cost of testing and related appraisal tasks.

We can analyze these models in terms of our proposed quality metrics. The quality of conformance in the original model can represent one quality metric, such as adaptability or reliability. Accordingly, the vertical axis represents a CoSQ component—namely, the portion of quality costs dedicated to improving the particular quality factor. Intuitively, the same nonlinear relationship should hold. Increasing the investment in improving a certain quality factor should increase the value of the corresponding metric, and the amount of this increase should taper off as the product achieves high quality levels. "Perfect" quality may not be achievable at finite costs, particularly in CBSs, where we cannot accurately determine the quality and performance of the COTS components.

Although we may be able to determine the overall CoQ with reasonable accuracy, determining the amount dedicated to improving a particular quality factor is difficult because all factors interrelate. For quality metrics such as customer satisfaction, the relationship between cost and quality may be too complex for such a simple model, as increased investments in quality improvement may be invisible to the customer. For example, users may find 95 percent reliability satisfactory, making further investments in reliability pointless. Further, customer satisfaction may increase in jumps, resulting in a discontinuous cost-quality curve, although empirical studies should verify this behavior.

Capability maturity models

Quality improvement's return on investment depends on the software engineering environment. Stephen Knox discusses the cost of software quality based on the Software Capability Maturity Model.[10] The SW-CMM maintains that a software development envi-

ronment has a measurable process capability analogous to industrial processes. The SW-CMM quantifies the capability and maturity of the software development process using five levels, ranging from a chaotic, ad hoc development environment to one that is mature and optimizing. These levels can also express the software engineering environment metric we propose. Based on the data presented, Knox makes two assumptions:

- The total cost of quality at SW-CMM Level 1 equals approximately 60 percent of the total cost of development.
- The total cost of quality will decrease by approximately two-thirds as the development process reaches SW-CMM Level 5, or full maturity.

Figure 2 shows the various software cost-of-quality categories, as well as the total cost of software quality, according to Knox, for the five SW-CMM levels.

A combination of Knox's model and traditional models provides a more accurate view of the CoQ in CBSs. A three-dimensional model based on CoQ, quality, and the software engineering environment can help determine financially sound investments in quality, based on the development environment. In the case of CBSs, where different vendors can have widely varying software engineering environments, such a model can help guide the vendor-selection process.

In place of Knox's SW-CMM levels, we can use the Software Acquisition Environment CMM to express the software engineering environment.[7] The levels of acquisition maturity range from Initial, at Level 1, to Optimizing, at Level 5. This model defines key process areas for Levels 2 through 5, in which a key process area states the goals the software must satisfy to achieve each level of maturity. SA-CMM and SW-CMM share a synergistic relationship, and we can use them in parallel by defining a software engineering environment metric with two weighted components, one corresponding to each CMM.

Applying the metrics

One objective of evaluating costs of quality is to determine ways to reduce them. A basic method involves investing in prevention costs, with the goal of eliminating nonconformance costs. As confidence in system quality increases, we can afford reductions in appraisal costs, leading to a reduction in total CoQ.

We can approach investments in quality improvement from the perspective of return on investment and increased conformance to requirements such as reliability,[11] then use the metrics to evaluate the actual quality improvement achieved as a result of a particular investment in software quality improvement.

Cost-benefit analysis of traditional software systems concludes that quality improvements yield the greatest returns early in the life cycle.[11] In CBSs, we cannot make quality improvements during the early stages of the acquired components' development. To compensate for this drawback, we must spread quality improvement efforts through the various stages of system design and development. In the design phase, such initiatives include

- identifying cost factors and cost-benefit analyses that address the unique risks associated with CBSs,
- determining the level of architectural match between the application and the COTS components, and
- evaluating the complexity and cost associated with integration, interoperability, and middleware development.

Our metrics can help decide between in-house development and COTS acquisition and, if the latter, how to select the most suitable component. In the development phase, metrics can help estimate the costs associated with the traditional development process. During the entire life cycle, metrics can guide the estimation of costs associated with the unique testing requirements of COTS-based systems, such as integration testing, end-to-end testing, and thread testing. After delivery, we can use cost metrics for trend analysis of the COTS market.

In the cost-benefit analysis of CBSs, we must avoid premature judgment, as the benefits of COTS component acquisition may materialize gradually. The paradigm shift from conventional to COTS-based

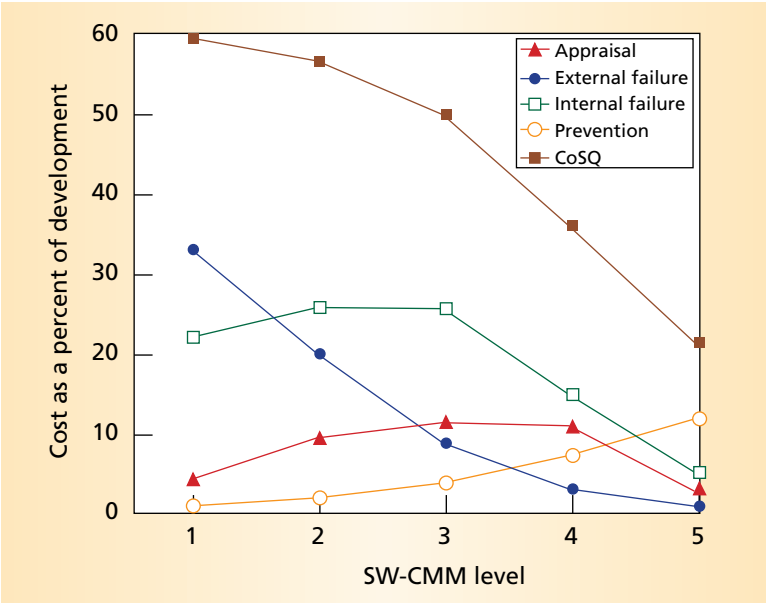

Figure 2. Knox's model for the cost of software quality, which tracks individual software cost-of-quality categories as well as the total cost of software quality. Figures adapted from Campanella.[2]

software engineering requires a considerable initial investment. Short-term analysis results may favor in-house development over COTS component acquisition, which argues for considering the software life cycle and level of reuse when making such decisions.[12] COTS products change rapidly, with long-term effects, and research on CBS development is still in the early stages. Given that cost-effectiveness and quality are the two major factors in deciding for or against component acquisition, we face an urgent need for empirical and analytical research that will lead to more accurate models of cost and quality in CBSs.

References

1. B. Boehm and C. Abts, "COTS Integration: Plug and Pray?" *Computer*, Jan. 1999, pp. 135-138.
2. J. Campanella, *Principles of Quality Costs: Principles, Implementation, and Use*, ASQ Quality Press, Milwaukee, Wis., 1999.
3. N.F. Schneidewind, "Software Metrics for Quality Control," *Proc. 4th Int'l Software Metrics Symp.*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 127-136.
4. R.A. Paul et al., "Software Metrics Knowledge and Databases for Project Management," *IEEE Trans. Knowledge and Data Eng.*, Jan./Feb. 1999, pp. 255-264.
5. P. Brereton and D. Budgen, "Component-Based Systems: A Classification of Issues," *Computer*, Nov. 2000, pp. 54-62.
6. R.A. Paul, "Metrics-Guided Reuse," *Int'l J. Artificial Intelligence Tools*, vol. 5, nos. 1 and 2, 1996, pp. 155-166.
7. J. Cooper, M. Fisher, and S.W. Sherer, *Software Acquisition Capability Maturity Model (SA-CMM) Version 1.02*, tech. report CMU/SEI-99-TR-002, Carnegie Mellon Software Eng. Inst., Pittsburgh, 1999.
8. R.A. Paul et al., "Assurance-Based Y2K Testing," *Proc. 4th Int'l Symp. High-Assurance Systems Eng.*, IEEE CS Press, Piscataway, N.J., 1999, pp. 27-34.
9. M. Tsagias and B. Kitchenham, "An Evaluation of the Business Object Approach to Software Development," *J. Systems and Software*, June 2000, pp. 149-156.
10. S.T. Knox, "Modeling the Cost of Software Quality," *Digital Technical J.*, Fall 1993, pp. 9-16.
11. S.A. Slaughter, D.E. Harter, and M.S. Krishnan, "Evaluating the Cost of Software Quality," *Comm. ACM*, Aug. 1998, pp. 67-73.
12. C. Sledge and D. Carney, "Case Study: Evaluating COTS Products for DoD Information Systems," *SEI Monographs on the Use of Commercial Software in Government Systems*, Carnegie Mellon Software Eng. Inst., Pittsburgh, 1998.

Sabra Sedigh-Ali *is a PhD candidate in the School of Electrical and Computer Engineering at Purdue University. Her research interests are software testing and quality management and component-based software development. She received an MS in electrical engineering from Purdue University. Sedigh-Ali is a student member of the IEEE and the ACM. Contact her at sedigh@ecn.purdue.edu.*

Arif Ghafoor *is a professor in the School of Electrical and Computer Engineering at Purdue University. His research interests are multimedia information systems, database security, and distributed computing. He received a PhD in electrical engineering from Columbia University. Ghafoor is a fellow of the IEEE. Contact him at ghafoor@ecn.purdue.edu.*

Raymond A. Paul *is the deputy director of investment and acquisition in the Office of the Assistant Secretary of Defense. His research interests are system and software testing and software measurements. He received a DE from the University of Tokyo. Paul is a senior member of the IEEE and the ACM. Contact him at raymond.paul@osd.mil.*