Adapting Systems Engineering for Software-Intensive Systems

Sarah A. Sheard
Software Productivity Consortium
Herndon, VA 20170 www.software.org
sheard@software.org (703) 742-7106

Abstract. The nature of systems engineering has changed in the last quarter century because of the increasing presence of software in systems. This paper describes the adaptations that systems engineers should make to their skills and approaches in order to engineer software-intensive systems better, and the places where systems engineers need to reinforce what continues to be useful about traditional systems engineering. Systems engineers should continue to contribute “systems engineering specialist” skills, help the project manager coordinate multidisciplinary activities that ensure a well-engineered system, and continuously learn about software. Finally, the systems engineer might need to advocate for good systems engineering.

Introduction

This paper is written for systems engineers with experience on large systems and who are more familiar with hardware-based systems than software systems. Different cultures and different skill sets require some adaptations for systems engineers to provide maximum value. This paper suggests some of those necessary adaptations.

Clearly there are many different ways to implement systems engineering. This paper addresses problems that may arise where specific people are designated as the “systems engineering group” on a software-intensive systems development effort. [Sheard 2000] calls this “Program Systems Engineering.” In this implementation, the systems engineering group is responsible for the integration and performance of the system as a whole, as compared with experts in specific areas or disciplines, who develop the components that are integrated. This paper focuses on Program Systems Engineering because it is here that the problem of hardware-trained systems engineers having to adapt arises most often.

The term “systems engineer” in this paper therefore refers to an engineer whose job confers responsibility for the system as a whole, rather than to someone holding a systems engineering degree or title. Similarly, the term “software engineer” refers to a person with the job function of creating software and does not imply a degree in computer science or software engineering. (It is noted, however, that the fraction of people with software engineering job functions who have computer science or software engineering degrees is much larger than the fraction of people with systems engineering job functions who have systems engineering degrees [Kasser 2000].)

Why Adapt for Software-Intensive Systems?

The nature of systems has changed significantly since the 1960s and 1970s, as considerably more of the solution complexity is appearing in software today. In part because software allows more functions to be performed than hardware ever did [Sheard 1998], current systems address much more complex problems, such as autonomous decision making. In fact, software development is now the major part of most complex system developments. Eberhardt Rechtin predicted at the third National Council on Systems Engineering (NCOSE) symposium in 1993 that although
the best systems engineers of the past came from command and control backgrounds, because those functions required a broad knowledge of the system, the best systems engineers of the future will come from software background, with software becoming the controller of most systems [Rechtin 1993].

Eleven years later, this prediction has not quite come to pass. Software and systems engineering areas have attracted people with different skill sets; for example, software requires attention to detail, whereas systems engineering requires a broader and shallower approach [Armstrong 1999]. Software engineers can be hesitant to leave a dynamic, rapidly growing field to take responsibility for systems that demonstrate structural failures or unintended chemical interactions. Often, therefore, the systems engineering functions on software-intensive system development programs tend to be populated by people without software backgrounds.

It has not been typical for software development efforts to call specific people “systems engineers.” The discipline of “software engineering” was created in response to a continuing set of project disasters as documented in 1968 [Gibbs 1994], and those who engineer the software as a whole tend to use that term rather than systems engineer. Rechtin’s prediction might have come true by now if software engineers consistently took advantage of the systems engineering knowledge base when engineering their software.

Lack of interaction between systems engineers and software engineers, however, is leading to some problems. For example, the author attended a software engineering conference and noticed repeatedly that many tenets of systems engineering were being discovered again for software. One example is “You really can’t take a specification as the last word; you also must elicit requirements and needs from the customer and user to guarantee that you really understand the problem.” Another example is an organization whose software engineers did not consider the trade studies process as applicable because it had been written by systems engineers. In both these cases, better engineering would have resulted from broader application of traditional systems engineering practices.

One further impetus to combining the skills of systems engineers and software engineers began in 1994 with an article about the “chronic crisis” in software development [Gibbs 1994]. It had become clear that education of individual software engineers was not making enough difference in the large projects that had become standard; software projects also needed more project management discipline. The Software Engineering Institute’s (SEI’s) Capability Maturity Model® for Software (SW-CMM) [Paulk 1993] focused on discipline first and won great praises for helping companies improve cycle time, quality, and cost. The 2002 release of the Capability Maturity Model® Integration (CMMI®) [SEI 2002] combined software with systems engineering because many companies already were performing a great deal of systems engineering on software systems out of necessity, and the walls between these two disciplines were hurting progress.

To break down these walls and reduce the crises, it is critical for software-intensive system development programs to focus better on engineering systems as a whole. Those systems engineers who do not have significant software experience should examine the parameters of a software-intensive-system development effort, to see where they might need to adapt their systems engineering approach, and where they should reinforce traditional systems engineering practices. This paper recognizes that many of the problems are fairly common in the industry. The following discussion focuses on these common problems and on how systems engineers might adapt to address each of them.

---

1 Capability Maturity Model®, CMM®, and CMMI® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.
How to Adapt Systems Engineering for Software-Intensive Systems

Systems engineers on large projects are generally there because the project needs a focus on interfaces and on the value added to systems beyond the value of individual components. The systems engineers need to become aware of key parameters related to all disciplines and of the major functional and design drivers within various domains. They also perform specific systems engineering tasks such as systems requirements analysis and system validation, and they help the project manager ensure that the entire project contributes as appropriate to engineering the system. The systems engineer is responsible, possibly more than others, for promoting the necessity for all to understand each other and benefit from the team’s diversity of strengths, roles, and knowledge. The question is how to improve the way they do these things on software-intensive systems.

The following sections address five basic issues:
1. Address overall conflicts.
2. Apply systems engineering specialist skills to the software-intensive system.
3. Help the project manager ensure that the system is engineered by all necessary contributors.
4. Learn about software.
5. Advocate for good systems engineering.

1. Address overall conflicts. When systems and software engineers begin to work together on projects, some problems occur fairly frequently. Because systems engineering has the responsibility to coordinate and integrate, systems engineers should learn about these differences and adapt their methods of working to facilitate communication and cooperation.

Vocabulary. Terms such as commercial off-the-shelf (COTS), systems-of-systems, interoperability, model, and even “systems engineering” have a broader meaning in the systems engineering world than in the software world. A systems engineer who refers to a “system” may be thinking of an airplane or a squadron, but many software engineers will hear “avionics” or “squadron information system.” [Birmingham 2004] gives additional examples of disparate vocabularies. The systems engineer should convey that the systems engineering definition of “system” consists of more than software plus computer hardware and may require tradeoffs between software and physical items.

Life cycle. Systems built of long-lead hardware necessarily follow some variant of a waterfall development life cycle, to ensure that the system has these difficult-to-procure components on time. By contrast, software efforts have developed a best practice of using spiral development to help zoom in on actual customer needs and preferences without risking the coding of too many features not in top demand. Other differences related to life cycle include whether or not the system is unprecedented, and whether or not defined processes are used for most of the technical work: systems engineering is most often employed on unprecedented systems, and software has been using defined processes for longer.

2. Apply systems engineering specialist skills to the software-intensive system. Some systems engineering activities require specific engineering skills (as opposed to the generalist skills in Section 3 below). These specialist skills are learned over a period of time via familiarity with many different projects and programs, and usually a “systems engineering” group is tasked with performing them at the system level. These skills include the following:

- Requirements elicitation, interpretation, and development
- System architecture, and allocation of requirements to components
- System analysis
- Definition and control of interfaces
- Verification at the system level
- System validation
- Identification of risks and problems
- Coordination to ensure that issues do not “fall through the cracks” and that system balance is maintained

The individual or team given responsibility for these activities is termed a “systems engineering specialist.” The author has seen a project suffer when it was bid without systems engineers because “the project just requires some glue-code among commercial software components.” The programmers were not familiar with the above specialist tasks and regretted the management decision not to include systems engineers; they found these skills difficult to develop quickly. Some issues associated with these specialist tasks in software-intensive systems are discussed below.

Requirements. A major issue for both systems and software engineers, requirements define what the system must do and how well. A primary systems engineering principle is that one must understand the need before one designs the solution. Requirements have some different aspects in the two worlds of software and systems engineering, as follows:

Limitations. First, software requirements are well-specified statements for which the software group can create a design. (Note that software design is limited only by logic, not by physical attributes; therefore the capabilities are almost endless until they are narrowed by requirements.) The first contrast that comes to mind is that one does not have to specify as much when dealing with hardware product lines. Using batteries as an example, all a satellite battery engineer needs to know at first is the specified capacity in ampere hours and an operating temperature range, because many other aspects follow from these two items and from known constraints of space, weight, and lifetime.

Freezing requirements. Another contrast is between software requirements and system requirements. System requirements start out as statements of customer need and evolve to well-defined requirements that the system should meet, but these and their subsystem allocations are subject to revision if one of the assumed components (usually hardware) turns out to be infeasible or if the customer’s situation changes (say a war is fought in a desert instead of in a jungle). It is required to freeze software requirements to get the software developed, but it is rare to have system requirements that are truly frozen.

Countability. A second difference in requirements is that software requirements tie more evenly to “features” than hardware-based requirements do. Many software engineers are accustomed to thinking of a “requirement” as a function the system must perform, such as calculating the year-to-date net present value or providing a form for a certain kind of input. Although any two requirements like these are not alike, it still makes some sense to count this type of requirements for estimation and tracking purposes.

Contrast this to an aerospace requirement for a satellite to “perform at full capacity during an eclipse.” Such a requirement cannot be implemented simply because it has implications in many different subsystems, from battery size and discharge rate to thermal limits, maximum power usage, and even orbit parameters. As another example, a requirement to make a component “man rated” (so it is suitable for use in occupied airplanes and launch vehicles) implies a huge number of derived requirements, from pressure vessel design to logical failure protection to vibration testing limits. Either of these simple requirements may have more impact than hundreds of require-
ments specifying individual component qualification temperatures. Estimating system development effort from number of requirements therefore is very problematic and is sometimes not done at all (preferring instead to scale the actual effort on a similar design) or modified by guesses at how many requirements could be considered “small,” “medium,” “large,” or “huge,” along with typical effort for each type.

Priorities. The systems engineer should recognize that the first priority for software is understanding the features to be created (behavioral requirements), which means that other important requirements, such as hardware interfaces, may not be addressed until later. Non-behavioral requirements also may be given short shrift in some cases because it is not as obvious what to do with them as it is with feature-based requirements. Systems engineers can help by keeping track of the necessary non-behavioral requirements and ensuring that they are addressed at the appropriate time. By the time there is a good idea of high-level software design, the software architecture should be ready to be evaluated with respect to non-behavioral qualities.

The systems engineer must be sure to involve software engineers in systems requirements efforts so that software requirements will be expressed at a meaningful and useful level of detail. In some organizations there is a culture of “handing off” requirements at the system level to subsystem areas without a very useful but occasionally painful phase of negotiation, impact analysis, and restructuring of the requirements and design. A breakage in this step—the translation from system to software requirements—is frequently part of the reason for system development failures.

Finally, in some quarters a fierce debate persists between systems engineers and software engineers about the value of functional requirements analysis. If carried too far, system requirements analysis can interfere with good object-oriented software engineering. An early teacher of an object-oriented software class once advised students to discard software requirements given to them by systems engineers and go back to the needs, starting over with object-oriented requirements analysis [Lilly 1993]. Systems engineers counter that an object view of a system is only one view and most systems require a number of views, including views of system function. For some kinds of systems, more innovative designs can be created if one backs off from known objects like wheels and taillights and concentrates first on what functions the system is supposed to perform.

Systems engineers cannot assume that software engineers will consider functional analysis to be important or useful. It is important for the systems and software engineers to establish a conceptual boundary beyond which known functions are allocated to the software as a whole and not further decomposed within that boundary by the systems engineers. It is then the software engineer’s responsibility to apply good software design principles within that area.

Transition from Requirements to Design. Systems engineering methods emphasize focusing on requirements separately from design (and design usually means an architecture, in the realm of software-intensive systems). If a customer states requirements in terms of desired components, systems engineers learn to say, “But why do you want that piece; what function will it perform for you? Perhaps we can do that better/cheaper/faster…” A whole field of decision analysis has evolved methods to find a design that best meets those requirements, such as trade studies, Analytic Hierarchy Process, and different types of utility function analyses.

In contrast, several well-known software methodologies advocate modeling of the problem space and gradually morphing that model into the software design. Missing from such a methodology is a deliberate attempt to trade off multiple potential architectures against criteria for the

---

2 A systems engineering joke: “How many systems engineers does it take to change a light bulb?” Answer: “Does it have to be a light bulb?”
architecture. This may be sensible for software implementations whose purpose is mostly to
model and control information about real-world objects, but in many cases other designs dis-
covered later may be better. In fact, re-architecting (at a low level) is the basis of the constant
"refactoring" on which Agile Software Development predicates itself [Fowler 1999]. Systems
engineers should insist that at least one alternative architecture should be examined to see if it
may better meet the system needs. At the very least, a systems engineer can mock up a trade
study, guessing at criteria and potential alternatives, to show that a trade need not take up too
much time; then the team may fix the criteria and rethink some of the alternative architectures.

**Architecture.** System architecture interfaces with software architecture in ways that are
somewhat foreign to systems engineers who are used to hardware systems. Software can be
thought of as an overarching conceptual aspect of design, whereas hardware comes in discrete
physical pieces. For example, whereas component diagrams show how hardware elements are
connected, good software architecture tends to be layered, with few layers connected in any way
to physical objects [Rechtin 1997]. Between the software and systems engineering communities,
there should be a serious effort to find ways to describe the relationships of the two architectures.
The state of the art today does not include a cross-architecture representation method that is well
accepted in both communities. Intense review of each group’s work by the other may continue to
be the best one can achieve today.

**Integration and Interfaces.** Systems interfaces take many forms. Logical interfaces are of
course very important to software, but electrical, mechanical, thermal, static and dynamic, per-
sonnel, equipment, and facilities interfaces also are critical in some systems. The key to software-
intensive systems interface design is two-way: the systems engineer should strive to understand in
detail the logical interfaces and what the software engineers need to know about the other inter-
faces before they can work the software, and the systems engineer should also highlight to the
software engineers the complexities of other kinds of interfaces that may be relevant.

To address these issues the systems engineer can push to implement a process that calls for
appropriate people to verify the correctness of interfaces, whether in an ongoing manner or in a
periodic workshop-type review. This approach is not different from a good approach on hardware
systems, but the types of people and types of key interfaces may change.

Another important issue is that software and systems engineering tools must interface as well
as possible, both to pass data and to verify interfaces between items modeled in the software tool
and items modeled in the systems engineering tool. Tools are changing rapidly, both following
and driving the evolution of software components and of software-based systems capabilities.
Tool interface is not trivial, and many systems engineers may not be familiar with the difficulties
because the complexity of all kinds of development tools has taken a major jump in the last
decade.

**Identify system risks.** System risks often relate to the interrelationships among components
or disciplines and to the relationship between the system and the environment. It is important that
systems engineers learn as much as possible about key or difficult design parameters and likely
risky areas within the software development effort, so as to be able to suggest mitigations and to
bring risks to the attention of other appropriate groups. Cooperative investigation into design risks
before the designs have been solidified can be very helpful in preventing rework and schedule
slips.

**Verification and validation.** Systems and software engineering standards use different
definitions of these terms. The CMMI has settled on the systems engineering standards’ defini-
tion: verification means showing that the system was built to its requirements, and validation
means showing that the system works in the operating environment and satisfies the customer’s
true need. Unfortunately not all engineers have adopted these definitions, and miscommunica-
tions may occur about these terms, particularly within the software area. Systems engineers should point out that effective system validation affects earlier phases too (such as seeing that requirements are testable) and should plan system integration, validation, and verification activities. Software still must verify inputs of one phase against outputs, and outputs against requirements, but the system functions also must be validated against the need.

**Whatever it takes.** It is the job of systems engineers to do what is necessary to ensure that the product will work. Software engineers, on the other hand, focus on getting working software, knowing that other people are in charge of the non-software aspects such as logistics, computer hardware, and non-computer hardware. Software engineers may remove themselves from problems that are not due to logic errors, sometimes to the extreme of believing that a requirements error is someone else’s problem. Systems engineers must address interface problems on behalf of the system, in light of what the software engineers see as their scope of knowledge and influence.

3. **Help the project manager ensure that the system is engineered by all necessary contributors.** In contrast to the “specialist” systems engineering activities described in Section 2, other activities to engineer systems can only happen through the contribution of people from many disciplines and areas. Systems engineers in particular can help engineer software-intensive systems if they address the following typical considerations (described in more detail in the following paragraphs):

- Technical management
- Quality assurance
- Configuration and data management
- Process description, implementation, and maintenance
- Technology insertion
- Agile project management

**Technical management** (e.g., planning, tracking, and estimation). Systems engineers, as part of their charter to be sure that the components of a system will integrate well, also may help coordinate and facilitate communication and collaboration among the groups building the components. This role treads delicately on the line between systems engineering and project management responsibilities. In addition to planning and tracking systems engineering tasks, systems engineers often help pull together the total project plan and help track progress against it. Other disciplines may perceive a loss of power when there is technical person between them and the project manager. It is helpful to clarify roles and to specify that the systems engineer is not making decisions, but rather assisting the system in coming together without problems.

Another issue related to technical management is estimating systems engineering. It can be difficult to estimate systems engineering effort in a generic manner when systems engineering differs greatly from project to project. Is systems engineering on a particular project only systems requirements definition, or does it also include determining the source of operational anomalies? Does systems engineering include interface definition and risk management? The University of Southern California-led effort to define a systems engineering estimation model, denoted Constructive Systems Engineering Costing Model (or COSYSMO), is facing the multiplicity of systems engineering definitions. Far more breadth is needed than was required for the related software estimation model. Systems engineers should assist with counting and estimation when

---

3 In fact, “integrate disciplines” was an “engineering” process area in the first systems engineering capability maturity model, the SE-CMM [Engineering Process Improvement Collaboration 1995]; but it became a “management” process area in its successor models, EIA 731 [Electronic Industries Alliance 1998] and CMMI.
possible, for example by categorizing requirements, anomaly resolutions, or change impacts by size.

**Quality assurance.** Many people must contribute to product and process quality. In big companies, often a separate quality assurance group audits product quality; but even in highly mature organizations (as measured against a capability maturity model), some organizations are accustomed to thinking of software defects as the most important or even the only measure of quality. Issues such as requirements clarity or electrical workmanship can seem foreign, as can discussions of process quality. Systems engineers must help implement a quality assurance approach that addresses all these factors and communicates well with the software quality assurance world.

**Configuration and data management.** These areas include tasks that specialists do, such as database entry, and also tasks in which everyone must participate, such as revision numbering. Systems engineers must be aware of varying ways of controlling hardware components (e.g., serial numbers, waiver tagging, and readiness reviews) and software components (e.g., builds, releases, and versions), and should help ensure that the configuration management and data management processes are tailored well to project needs.

**Process description, implementation, and maintenance.** Everyone must participate in using defined processes, and systems engineers often participate to some extent in developing, measuring, and maintaining systems engineering processes. As is described in [Sheard 2004], systems engineers have a particular responsibility to evolve systems engineering processes into integrated processes that support the development of systems. One challenge is to understand how to document the role of systems engineering specialists within an integrated process for engineering systems.

**Technology insertion.** Because the greatest changes in technology in the last few decades has been in computer technology, the word “technology” has come to be nearly synonymous with computers. However, more than computer and software technology may need to be inserted into products and into development processes. For example, new thermal materials may be lighter, less prone to static, or less expensive to maintain. Systems engineers need to understand the benefits and drawbacks of a wide variety of technologies that may be useful, and not allow the technology insertion effort to focus exclusively on computer technology.

**Agile project management.** Agile software development has been increasingly cited in the business press during the past decade. It is partly a set of techniques, such as pair programming, repeated refactoring (reorganizing) of software, and “time-boxed” design spirals, but mostly it is an approach to managing software projects. The following characteristics are important to Agile:

- A move away from waterfall methods that finalize requirements before beginning implementation, toward a “build a little, re-verify with customer” spiral
- A project philosophy that teams can devise ways of working together better on their own than by following reams of process documentation
- A focus on prototyping rather than on in-work documentation

The truth is that several of the aspects of Agile development have promise, particularly on small projects, projects that are mostly software, and projects short enough that the entire development team can expect to stay together for the duration of the project. There are concerns for larger systems, hardware systems, systems with special quality requirements such as man-rating, and projects with long development times. Large, complex, or critical projects require detailed plans to be sure that everyone is working to the same schedules and will provide critical information to each other when needed.
The systems engineer should become aware of Agile development characteristics, pros and cons, and comparisons to capability maturity models [Boehm 2003]. Peer reviewing is an adaptation of Agile techniques that can help systems engineers significantly. Systems engineers working in an Agile environment ought to be forewarned that they may need to defend some documentation practices when a paper trail is needed for operational reasons.

**4. Learn about software.** Systems engineers should learn the language and key parameters of software, the same way they have found it necessary to learn to read blueprints and schematic diagrams. Because software is a broad, deep, and growing discipline, this can be a difficult and even never-ending task. Yet it is very important to improving project communication on software-intensive systems, so that the diversity of knowledge and talent on the project can be used well.

Some specific items to learn are which technical performance measures (TPMs) and measures of effectiveness work well for a particular software system; what requirements are most difficult to meet and therefore most likely to drive software architectural decisions; what types of data are needed first by the software developers and what inputs can be postponed [Rechtin 1997]; and what is the relationship between object-oriented software development and system functionality. Systems engineers also should become familiar with the Unified Modeling Language (UML) and to what extent its capabilities may be applied to system modeling.

Because they will grow in importance, two additional topics that merit specific systems engineering education are (1) network-centric architectures and the interactions of distributed computer systems and (2) computer system security. [Sheard 2003] provides an awareness introduction to the latter.

**5. Advocate for good systems engineering.** Is traditional systems engineering still important for software-intensive systems? Complexity of systems has grown faster than the human ability to comprehend. No one can be an expert in communications, lubricants, databases, structural dynamics, and user education techniques all at the same time. It has been possible to assign many different experts to work on a project together; but unfortunately, if no one is given the charter to balance needs, communication is likely to be poor, leading to issues being resolved in unnecessarily ineffective ways. Thus the systems engineer must sometimes push for the adoption of traditional systems engineering practices, against claims that they are outmoded. Software organizations often discover that one can only go so far in improving software without also looking at the capability of interfacing processes. Rework will continue to be a software problem if system and software requirements are not understood well prior to software design. For these reasons, there is some push from the process improvement side of many organizations for software and systems engineering to work out roles and improve integrated processes [Sheard 2004].

**Establish the right kind of systems engineering.** The products of the organization need to be “systems engineered.” Many organizations choose to establish some kind of systems engineering group to take responsibility for the “system” aspects of products. In some cases, this is a functional group whose members are matrixed out to projects. In others there are systems engineering groups within each large project or program. Other projects set up teams, on each of which someone with known skills in systems engineering participates. In many software development efforts, there may be no acknowledgement of systems engineering, but some employees teach themselves to do what INCOSE calls systems engineering tasks. In any case, the organization should have a strategy for how products will be engineered to perform system functions and meet customer needs.

A systems engineer can establish a personal objective to make systems engineering within the company explicit, even if the organization does not currently use the term. Many INCOSE documents and papers are available that can help the engineer list needed tasks. Eventually the
engineer may sketch up a process for engineering systems using the methods currently being performed in the organization, augmented by some valuable advice from standards and models. If the company subsequently needs to demonstrate compliance with CMMI or another integrated standard, this will be an excellent starting point. The process should include not only how products are engineered in an integrated manner, but also how systems engineering knowledge and lessons learned are shared.

If systems engineering skills need nurturing, then a systems engineering functional group or a cross-team working group may be helpful. It is also beneficial to make projects aware of the individuals who have skills in the systems engineering specialist tasks described above, because projects then will know where to find help. Finally, the systems engineering group should work closely with other disciplines and teach them about the skills, knowledge, and approach of systems engineering, while learning from them at the same time.

Conclusions

Software is becoming the major part of most complex system developments, yet systems and software engineers experience some typical problems related to vocabulary, perspective, specialist skills, generalist skills, and culture resulting from stove-piped organizational structures. Systems engineers can help improve the situation with five practices:

1. Address overall conflicts.
2. Apply systems engineering specialist skills to the software-intensive system.
3. Help the project manager ensure that the system is engineered by all necessary contributors.
4. Learn about software.
5. Advocate for good systems engineering.

References


Lilly, Susan, personal communication, 1993.
Bob Kenley and Sten Dahlberg provided very valuable comments on a draft of this paper. Rich McCabe and Mike Polen were great defenders of the software point of view while educating a systems engineer with a tendency toward generalization. Mike Phillips was a co-presenter on a similar presentation at the Systems and Software Technology Conference in May 2004.

**Biography**

**Sarah A. Sheard** received the 2002 INCOSE Founder’s Award for her work in INCOSE, including publishing more than 30 papers, chairing the Measurement technical committee and the Communications committee, and serving as program chair and Director of the Washington Metropolitan Area chapter. Ms. Sheard has worked in systems engineering and process improvement for more than 20 years and is currently a Chief Technologist leading the systems engineering effort at the Software Productivity Consortium.