29th Annual INCOSE international symposium
Orlando, FL, USA
July 20 - 25, 2019

Systems Engineering–Software Engineering Interface for Cyber-Physical Systems

Sarah Sheard, PhD
Software Engineering Institute
4500 Fifth Ave. Pittsburgh PA 15213
(703) 994-7284
sarah.sheard@gmail.com

Michael E. Pafford
INCOSE Chesapeake Chapter
3516 Marlbrough Way, College Park MD 20740
(301) 980-8474
mepafford@verizon.net

Mike Phillips
Software Engineering Institute
442 Dorseyville Rd. Pittsburgh PA 15215
(412) 260-0364
phillipsdm@aol.com

Copyright © 2019 by Carnegie Mellon University and Mike Pafford. Permission granted to INCOSE to publish and use.

Abstract. Almost all of today's man-made mechanical and electronic systems are actually cyber-physical systems (CPSs). Formerly physical systems, from rockets to hair dryers and faucets, gain capabilities from software sensing, calculating, and control. Although some software will remain mostly divorced from physical items (e.g., cloud systems), much complex software will control, sense, and communicate with physical systems, which are then called CPSs. Although many systems engineers did not come to the discipline from a software background, CPSs still need systems engineering. Additionally, software engineers must step out of their "subsystem" box and work with systems engineers to build tomorrow's systems. To enable that, systems engineers should continue to apply systems engineering principles, including continuous learning (especially about software) and coordinating (including teaching software engineers about systems engineering). The time is now to take on new behaviors to meet the challenges of CPSs. Systems engineers must work alongside software engineers to reach the joint goal of system success.

1. Introduction

Systems engineers and software engineers must work together to properly engineer today's and tomorrow's cyber-physical systems (CPSs), including physical components and the related software, to ensure system quality attributes such as safety, security, and maintainability. This is because without software, new capability is minimal, and without good systems engineering, the costs and schedule of the pieces may not be optimal, the integration may be problematic, and a delivered system may not meet customer needs. Software also provides a flexibility of options that is not seen in hardware. Both systems and software engineering are required for future systems. Their work must be coordinated.

This paper is organized as follows:[1] Section 2 discusses some aspects of how engineering, systems engineering, software, and software engineering came to be what they are today. Section 3 compares systems engineering and software engineering. Section 4 discusses systems of systems and CPSs. Section 5 shows that systems engineering and software engineering are still relevant. Section 6 presents how to coordinate the systems and software engineering of CPSs, including understanding tasks and roles. And section 7 presents a vision of a high-performance systems–software interface.

2. Systems and Software, Past to Present

a. What Is Engineering?

The hardware parts of the system, and the system as a whole, require engineering of the physical elements and combining physical properties with logical properties. Engineering has evolved as a discipline consistent with its key role in society: engineers invent the technology that has improved our quality of life and continues to change how we interact with each other, other living things, and the earth. Because of the importance of good engineering to societal safety and security, professional engineers have legal responsibilities (Illinois Institute of Technology 2019).

One important element of the discipline is a code of ethics. Table 1 shows the IEEE's 10-statement "Code of Ethics" that all members agree to follow (IEEE 2018). Similar codes exist in the Society of Professional Engineers, indeed in most engineering societies, including INCOSE.

Table 1. IEEE Code of Ethics

We, the members of the IEEE, in recognition of the importance of our technologies in affecting the quality of life throughout the world, and in accepting a personal obligation to our profession, its members, and the communities we serve, do hereby commit ourselves to the highest ethical and professional conduct and agree:
1. to hold paramount the safety, health, and welfare of the public, to strive to comply with ethical design and sustainable development practices, and to disclose promptly factors that might endanger the public or the environment;
2. to avoid real or perceived conflicts of interest whenever possible, and to disclose them to affected parties when they do exist;
3. to be honest and realistic in stating claims or estimates based on available data;
4. to reject bribery in all its forms;
5. to improve the understanding by individuals and society of the capabilities and societal implications of conventional and emerging technologies, including intelligent systems;
6. to maintain and improve our technical competence and to undertake technological tasks for others only if qualified by training or experience, or after full disclosure of pertinent limitations;
7. to seek, accept, and offer honest criticism of technical work, to acknowledge and correct errors, and to credit properly the contributions of others;
8. to treat fairly all persons and to not engage in acts of discrimination based on race, religion, gender, disability, age, national origin, sexual orientation, gender identity, or gender expression;
9. to avoid injuring others, their property, reputation, or employment by false or malicious action;

[1] This paper reflects a U.S. Department of Defense (DoD) point of view, but the conclusions apply to other systems engineering applications. The roles discussed here are most applicable to large complex CPSs. See the INCOSE handbook (Haskins 2006) and the Systems Engineering Body of Knowledge for broader roles and types of systems engineering.

10. to assist colleagues and co-workers in their professional development and to support them in following this code of ethics

Systems engineers are often engineers, and while not all systems engineers could pass a "professional engineer" exam, their backgrounds and duties are usually closer to engineering than those of software developers, who tend to have degrees in computer science or mathematics.

The term *software engineering* was used at a NATO conference in 1968, but in an aspirational sense meaning roughly that "We shouldn't just *write* software, we need to *engineer* it" (Naur & Randell 1969). Since then, "software engineering" has been a frequently-used term, but the practice is still an art in many ways. For example, there has been little requirement for software engineers to meet codes of ethics, and often no consequence to the creator for software engineering errors. But the tide is turning: as discussed in Section 5c, Toyota had to pay $1.6 billion ($US) to settle a lawsuit about 89 people killed or injured when a car's throttle software malfunctioned (Koopman 2014). Smart organizations are taking note.

b. What Is Systems Engineering?

Depending on the source, systems engineering may be defined as a discipline, a department, a career path, a process, or a set of activities. INCOSE defines *systems engineering* as follows (INCOSE 2019):

> *Systems Engineering is an interdisciplinary approach and means to enable the realization of successful systems. It focuses on defining customer needs and required functionality early in the development cycle, documenting requirements, then proceeding with design synthesis and system validation while considering the complete problem:*
>
> *Operations; Cost & Schedule; Performance; Training & Support; Test; Disposal; Manufacturing*
>
> *Systems Engineering integrates all the disciplines and specialty groups into a team effort forming a structured development process that proceeds from concept to production to operation. Systems Engineering considers both the business and the technical needs of all customers with the goal of providing a quality product that meets the user needs.*

Some of the tasks assigned to systems engineering teams or individuals on a project include the 12 roles first discussed in (Sheard 1996) and presented in Table 2. These tasks show the differences in assumptions about what systems engineering is and does among early papers and books. No one person is expected to take on all the roles; rather, who performs what role should be negotiated.

Table 2. Twelve Systems Engineering Roles (Sheard 1996)

RO	Requirements Owner
SD	System Designer
SA	System Analyst
VV	Validation/Verification Engineer
LO	Logistic/Ops Engineer
G	Glue Among Subsystems
CI	Customer Interface
TM	Technical Manager
IM	Information Manager
PE	Process Engineer
CO	Coordinator
CA	Classified Ads Systems Engineer

Systems engineering started and matured somewhat before software was a big part of systems. Sheard (2014a) showed the evolution of software (solid red) within satellites (blue boxes), as shown in Figure 1. Software started out as a small part of one of the electronic units, then it became a larger part of more units, then point-to-point connections among a number of units appeared. Around 2000, data buses started appearing, then full software architecture. It is expected that in the future, hardware will be depicted as interconnecting existing software networks.

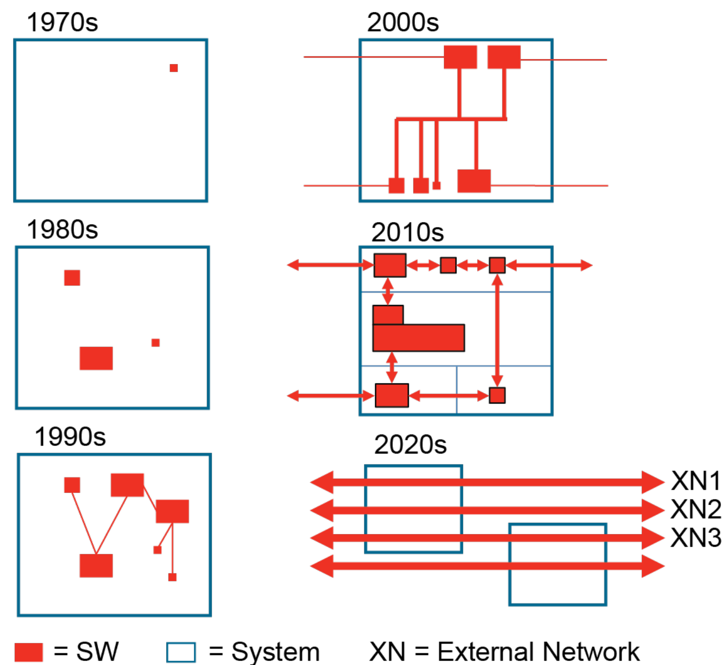

Figure 1. Growth of software within satellites.

With such an increase in the software content of today's systems, systems engineers fear their discipline can become irrelevant if not updated (Lunney 2018).

c. What Is Software?

In the 1960s and 1970s, software was written as small stand-alone programs, for business or engineering purposes, in languages such as Basic, Cobol, and Fortran. Coding was easier, and the tools were more advanced than machine language and then assembly language. Today's sophisticated languages and environments have made coding still easier, with fewer syntactic errors.

Software exists today in many forms, including mobile games, the Internet of Things, large and small (e.g., web shopping carts) business platforms, science and engineering platforms, and embedded systems. This paper addresses embedded systems, in which the software is intimately related to physical sensing and actions. Information systems, in which the relevant hardware is the computer platform (a commodity rather than special-purpose hardware), are not the topic of this paper.

Software has become the dominant way "things" talk to each other. Using the Internet's physical and logical backbone, software interconnects multiple computers along with the physical systems they monitor and control. Vehicles such as cars and airplanes are no longer internally controlled primarily by physical signals (cables, struts, chains, hydraulics, etc.) but rather by buses connecting the computers that run entertainment systems, heating and cooling, propulsion, navigation, and other systems.

These systems evolve rapidly because the software within them (and on the external Internet) is evolving rapidly. Software enables the primary capabilities of most "things" today and empowers rapid growth in capabilities.

d. What Is Software Engineering?

Definitions of software engineering vary. Merriam-Webster calls it "a branch of computer science that deals with the design, implementation, and maintenance of complex computer programs" (Merriam-Webster 2019). The IEEE defines software engineering as "the systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software" (IEEE 2017), while Webopedia insists that a software engineer "is a licensed professional engineer who is schooled and skilled in the application of engineering discipline to the creation of software" (Stroud 2019). Note the progression from *computer science* to *application* to *licensed professional engineer*.

In the authors' experience, most software engineers today consider the first or second definition to describe what they do, not the third. Hardly any software engineers are licensed professional engineers.[2]

Compared to systems engineers, software engineers must have more depth in their field, as individual bits matter greatly to whether a program runs or crashes. Software engineers must also work much harder to stay current, as the field is rapidly evolving and highly complex.

[2] In fact, the National Council of Examiners for Engineering and Surveying (NCEES), decided to discontinue its exam in "Principles and Practice of Engineering Software Engineering" exam after April 2019 for reasons that included the "low candidate population" and the "[low] potential for increasing the number of first-time examinees." (Miller 2018)

This high complexity is managed (and probably can only be managed) through the use of tools that constrain the potentially high complexity of the software. These include modeling, design, test, configuration management, and version control tools and are usually collectively called a software development environment.

To the extent that software engineering is true engineering, systems engineering of software is usually performed by the more experienced software engineers who are interested in the big picture. These are often called software architects. Their job is broader than that of less experienced software engineers, but it is less broad and more detailed than systems engineering of more physical systems. Figure 2 compares a T-shaped systems engineer to a T-shaped software engineer (Sheard 2014b).

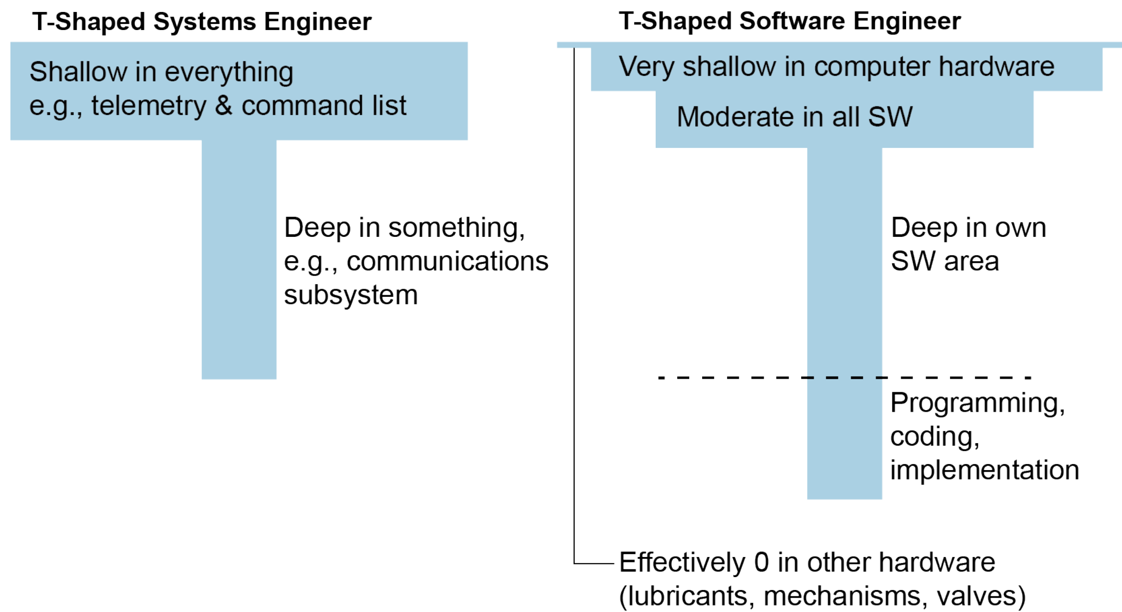

Figure 2. T-shaped Skills: Systems Engineer and Software Engineer

3. A Comparison of Systems Engineering and Software Engineering Today

Figure 3 (Sheard 2014c) shows the differences and overlap between systems engineering and software engineering responsibilities and tasks. Systems engineering performs the tasks in the left crescent, and software engineering performs the tasks in the right crescent. Where the systems and software tasks are nearly the same, they appear in the center, which includes both joint responsibilities and a note that some terms are different on the left and right but are effectively the same (e.g., "ilities" and quality attributes are both non-functional requirements). This figure was created in 2014 and is still appropriate, although today under the red "Breadth" might be added emergence, judgment, and capabilities; and under the blue "Depth (SW)" might be added languages, features, epochs, media, open source tools and source code, machine learning, artificial intelligence, etc.

For CPSs, systems engineering tasks, methods, and competencies overlap with many software engineering tasks, methods, and competencies; systems engineering also extends to a broader scope as necessary for a successful project.

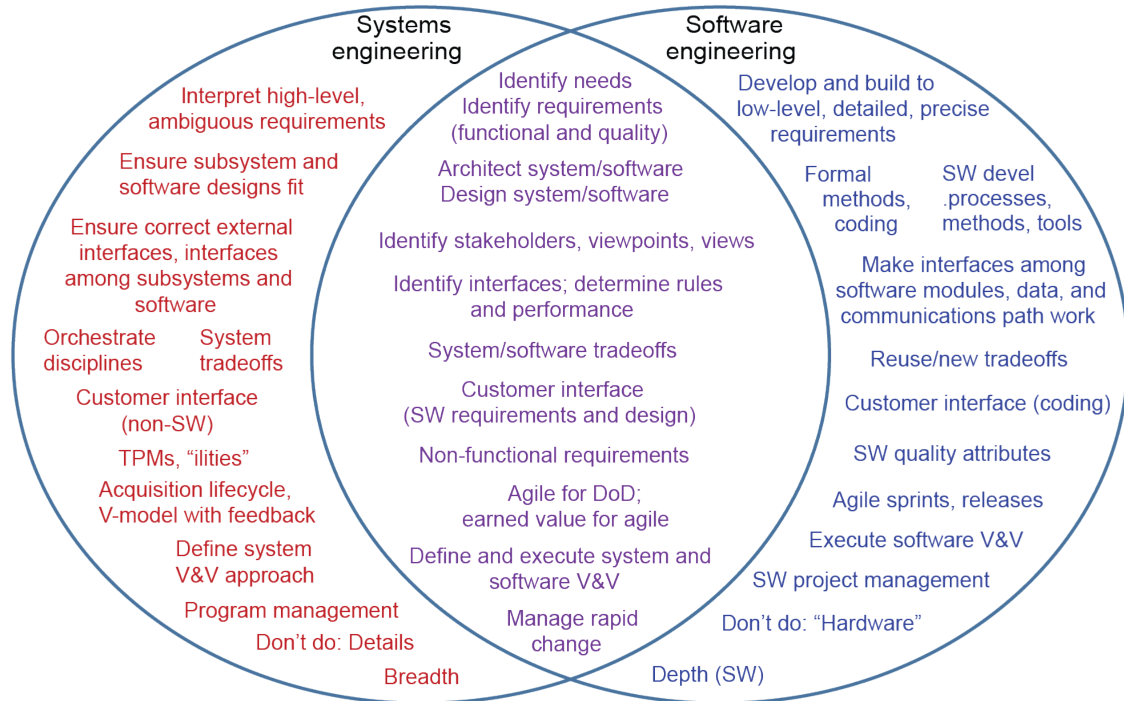

Figure 3. Responsibilities and activities of systems engineering and software engineering.

4. Today to Tomorrow: New Kinds of Systems

Two new kinds of systems have been defined since the 1990s. Both are evolutions of the kinds of systems built up until then. Systems of systems (SoSs) are aggregations of smaller systems to enable new capabilities. CPSs have more software functionality than earlier physical systems.

a. Systems of Systems

Mark Maier's seminal work in the 1990s defined "systems of systems" as different from "systems" in how they are engineered. SoSs are those that have (a) operational independence of the elements, (b) managerial independence of the elements, (c) evolutionary development, (d) emergent behavior, and (e) geographic distribution (Maier 1998).

In response to the ensuing rise of work on SoSs, the DoD pulled together a government–industry team to create its *Systems Engineering Guide for Systems of Systems* (ODUSD A&T 2008). Table 3 shows core elements in the guide: systems engineering roles and activities that are different from systems engineering roles and activities when building standalone systems.

Table 3. Core Elements of SoS Systems Engineering

TCO	Translating Capability Objectives
USR	Understanding Systems & Relationships
OUS	Orchestrating Upgrades to SoS
ARO	Addressing New Requirements and Options
MAC	Monitoring and Assessing Changes
DEM	Developing, Evolving, and Maintaining SoS Design

b. Cyber-Physical Systems

Most of today's man-made physical systems are actually CPSs, consisting of a significant amount of software, as shown in Figure 4. This software performs computational work to receive communications and data, store and process data, and issue communications and instructions for actuators, based on the computational results. The software controls and manages the physical system and carries out communications with external systems. Although physical components and software may each be well-engineered, the physical and logical concerns are not always balanced properly.

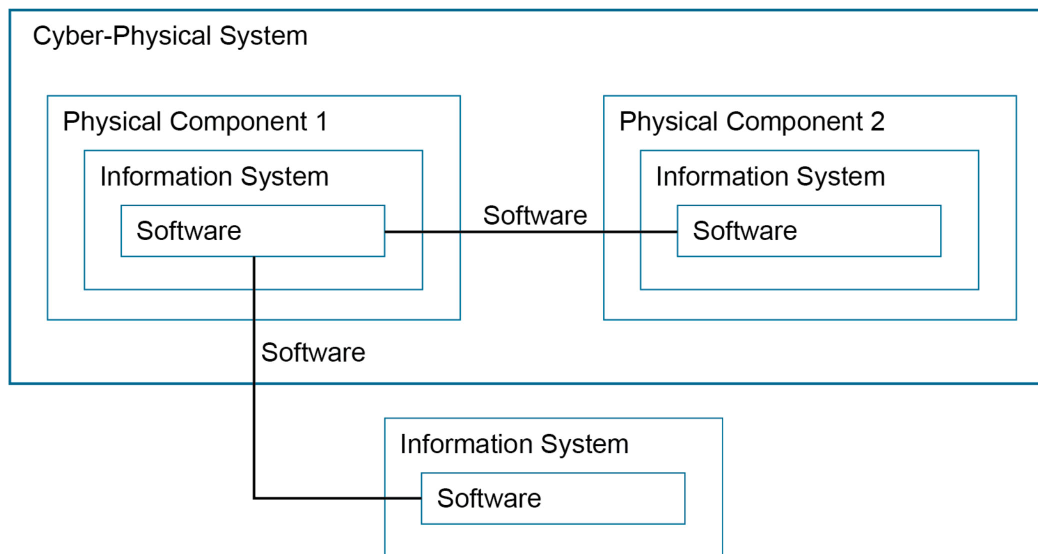

Figure 4. Software in Cyber-Physical Systems

CPSs are the systems of the future, but they are also the systems of today. It is of great urgency that systems and software engineers come to agreement as to how CPSs should be systems engineered.

Whereas software engineers can usually perform most or all of the systems engineering for the software in information systems, software engineers cannot speak for the non-software parts of CPSs. There should be a systems engineer who can and does, and there should be a software engineer who speaks for the software and who performs systems engineering for the software. These two people should coordinate easily and often. All systems engineering work should be cleared for software acceptability, and all software engineering work should be cleared for systems acceptability.[3]

On a program developing a complex CPS, a systems engineer performs the systems engineering. A software engineer systems engineers the software, with this external "system" being the computer system on which the software will run, as well as the computer system's external system.

[3] John Klein, personal communication, 9 November 2018.

The two disciplines collaborate to ensure that the performance of the CPS and its development cost and schedule are satisfactory.

5. Relevance

The thoughtful combination of systems and software engineering will be relevant in the future because customers will continue to require high-capability, working systems that meet their needs. This sections shows three examples of the need for this joint work.

a. Example: Processes and Agile

The U.S. DoD has an extensive history of using a "waterfall" acquisition model that assumes requirements are set, contracts are let, systems are built, and then the customer tests the systems to ensure they perform per the original requirements. In the 1990s, both government and contractors realized that the time frames to build systems were longer than the time that a requirement stayed steady, and even if it were practical to gather *all* requirements before a contract is begun, the requirements would be obsolete before the system was delivered.

For the past 20 years or so, software developers have created a method for developing software in pieces, from small numbers of requirements, which are then evolved to form increasingly capable software. Although the DoD is actively trying to ensure it has policies that permit and even encourage agile software development, the wheels of change roll slowly, and some engineers perceive a disconnect between the older acquisition regulations and newer software practices.

In the same time period, INCOSE has studied how to do engineering in a more agile manner and how to systems-engineer more agile systems (INCOSE 2018). Systems engineers who are aware of the constraints on both the government acquisition side and the agile software development side need to be aware of these resources in order to make both happen.

b. Example: Early System Decomposition

One major problem arises when the system is decomposed into physical pieces (and then contractual pieces) without the participation of software engineers and architects. A government systems engineer from the Office of the Secretary of Defense once lamented that analysis of a communication path in a major program showed that one calculation (which bounced around between physical parts) required fast Fourier transforms five times. The contractual structure led to various contractors each building software that should have been common. They each transformed latitude and longitude to geo coordinates, had unshared memory hierarchies, and had separate support routines, data dictionaries, and mathematical routines like Fourier transforms. It would have been much better had software engineering been involved in early system decomposition decisions.

Today, program managers treat systems engineers rather than software engineers as their technical points of contact. Often, software engineering concerns are unheard unless their systems engineering counterparts raise the issues to their managers. For example, the GAO reports,

> *Of the 54 current and future programs we assessed, 41 reported software development as a high-risk area. Despite this, … half of the programs we assessed do not track the cost of their software development. (GAO 2017)*

Systems engineering is understood to focus on the whole product, and it sets the scope for both the program and its elements, such as software. Software engineers needs to make their case initially and throughout the program in the scramble for resources, because software is too important to come in last with respect to hardware.

c. Example: System Safety

Can software kill? No one has died solely from the existence of sequences of 1s and 0s in computers, but people can and do die when software interacts with the physical components of CPSs.[4] In 2014, Toyota was required to pay $1.6 billion to settle a lawsuit concerning people injured or killed in accidents caused by throttle software in its vehicles. The large amount was due in part to Toyota continually reassuring the public that it was not the fault of the vehicles or their software (suggesting driver error or interfering floor mats first), when it was indeed due to inadequately safe software. The complexity of software is measured by an index that states the number of linearly independent paths through a program's source code, called cyclomatic complexity (also called McCabe complexity). Good software practice limits the cyclomatic complexity of programs to 25 or 30 for understandability, and programs with complexity over 50 are considered untestable. In comparison, the complexity of Toyota's throttle angle function was 146, and there were 66 other functions with complexity over 50 (Koopman 2014). In this system, both software engineering and systems engineering seemed to have been inadequately performed for societal safety.[5]

Safety continues to be an unresolved challenge. As of March 2017, the Boeing 737 Max airliners have all been grounded because of multiple crashes in a short timeframe. The last accident is too recent to identify the precise cause, but it appears to be related to mounting a too-big engine with a second control system (that fights with the original) and both use the same sensor, whose data is unreliable. (Evers, 2019) In any case, the larger question is how software deals with possibly unreliable sensors and whether, when, and how it communicates such information to the pilots.

6. Adapting the Systems–Software Interface for the Future

This section discusses what must be done to ensure that CPSs have the right systems and software engineering. Systems and software engineers must agree on who does what (roles and tasks), and both groups should modify what they do to improve communication between the two groups.

a. Perform Coordinated Roles

Table 4 shows tasks (first column) and roles for both systems engineering (middle column) and software engineering (last column). Entries in the systems engineering column are based on the 12 systems engineering roles (Sheard 1996) in Table 2 (one- or two-letter roles), and on the 7 systems-of-systems engineering tasks (ODUSD A&T 2008) in Table 3 (three-letter roles).

---

[4] The question of whether the software is responsible when physical items harm humans is not trivial. If a Mafioso hacked into Fitbit's geolocation database and thereby found the location of his target, is the subsequent murder the fault of software?

[5] Aviation accident reports provide another rich source of system safety experiences.

Table 4. Systems and Software Roles and Tasks

Tasks	Systems Roles and Activities	Software Roles and Activities
	Technical Roles	
1. Implement	(none)	Programmer or coder (from detailed design) and testing of the code Agile team roles involving coding Includes debugging and documentation as required Maintenance coding Maintain current software engineering skills
2. Architect and design	Systems Architect RO Requirements Owner SD System Designer G Glue Among Subsystems (including risk identification) TCO Translating Capability Objectives USR Understanding Systems and Relationships ARO Addressing New Requirements and Options DEM Developing, Evolving, and Maintaining SoS Design	Software architecture design Architecture analysis needed to perform software architecture design Agile team roles involving refactoring and structuring Refactoring needed during maintenance Software architecting, including detailed design. Includes identification of components, relationships, and interfaces; allocation of requirements to next-level components.
3. Lead and coordinate	Liaison to other disciplines, including software engineering, and to component building groups Risk identification CO Coordinator OUS Orchestrating Upgrades to SoS	Software risk identification and escalation to system risks Liaison with other software engineers building components and with systems engineering Agile team roles and maintenance roles involving leading and coordinating
4. Analyze system and own interface external systems	Perform system analyses including budgets, margins, timing, and failure modes Maintain current system budgets with inputs from other groups Characterize external systems interfaces SA System Analyst CI Customer Interface APO Assessing Performance to Capability Objectives MAC Monitoring and Assessing Changes	Perform analyses including budgets, margins, timing, and failure modes Characterize external systems with which the software will interface Architecture analysis to determine best match of architecture to customer needs Analysis required during maintenance
5. Verify and validate	Plan and monitor system test processes and results Validate requirements through system operation VV Validation and Verification Engineer LO Logistics and Operations Engineer	Architecture evaluation Plan and execute software verification and validation from earliest component tests through software part of system testing and in operations

Tasks	Systems Roles and Activities	Software Roles and Activities
		Agile team roles involving determining adequacy of structure relative to today's understanding of need and to backlog Verification and validation of software developed for maintenance, including debugging and testing
	Management Roles	
6. Manage people	Manage systems engineers Ensure they can learn broadly	Manage software engineers Ensure their skills remain current
7. Coordinate with other groups	Interact with system-level customers Obtain agreements and resources as needed CO Coordinator OUS Orchestrating Upgrades to SoS	Interact with users and software-proficient customers Obtain agreements and resources as needed
8. Plan and monitor	Technical management TM Technical manager PE Process Engineer OUS Orchestrating Upgrades to SoS	Software task or sprint management Agile sponsor roles
9. Manage risk	Balance application of resources to reduce/mitigate system-level risks SA System Analyst G Glue Among Subsystems TM Technical Manager	Application of resources for software risk mitigation Escalation of risks to system risks
10. Manage configurations, data, and quality	Perform these tasks for system as a whole and possibly for some pieces IM Information Manager	Perform these tasks for software, data, and possibly computer hardware

b. Software Engineers Must Architect, Design, and Implement Their Software in a System Context

Software engineers are primarily responsible for these tasks (note: numbers in parentheses refer to tasks in Table 4):

- Design and implement the intelligence of the system. This is because the software implements the intelligence of the system, and the systems engineer may not have the background or years of experience to ensure this is done properly. If so, the software engineer or architect is critical. The systems engineer has secondary responsibility and should work closely with the software person to ensure the requirements are as correct as possible, address constraints such as low vulnerabilities to cyber-attack or safety hazards, and determine strategies for contingency situations (1, 2).

- Take responsibility for the entire software product. Understand activities and concepts such as code review, security vulnerabilities and countermeasures, coding/architecture patterns, and systems engineering tasks and responsibilities[6] (2, 6–10).
- Engage in analysis and design, allocation of requirements (2), oversight of component development (3), component integration (2), verification and validation (5), life-cycle sustainment (1–5), and system retirement (4 and maybe 3) (SEBOK Wiki 2018).
- Work with or as component specialists (for example, user interface, database, computation, and communication specialists) who construct or otherwise obtain the needed software components (1, 2) (SEBOK Wiki 2018).
- Adapt existing components and incorporate components supplied by customers and affiliated organizations (1 and 7, with plenty of 2 and 4) (SEBOK Wiki 2018).
- Allocate requirements to software modules; do component integration and test (1, 2, 5).
- Write and document the code (1).
- Verify and validate the code (5).
- Interface with users (agile). Software is increasingly being developed using agile techniques that include users on the teams. Thus software engineers are the primary contact with these users. In contrast, the customers who allocate the money for the software development often interact with the management chain first and systems engineering second. If the users lead the project toward substantial requirements or scope changes, software engineers should inform the systems engineers (1, 2, 3, 7, and some of 4).
- Keep current with rapidly changing environments and conditions, especially cybersecurity, languages and operating systems, development methodologies such as agile practices, and development environments as they evolve. Software changes much faster than an engineer with a broad responsibility could track it. Software engineers should inform systems engineers when industry changes lead to major changes in requirements or project scope (1, 4, 6).
- Maintain software: repair bugs (corrective), accommodate changes in the software environment (adaptive), implement new or changed user requirements to established software (perfective), and keep software reliability and maintainability high (preventive). Every change goes through a miniature version of a full life cycle (1–10).

c. Systems Engineers Must Adapt Systems Engineering Practices to Include Software Engineers as Important Participants

Since systems engineers often are asked to help the program office early in the program, they need to be aware that "big-picture-aware" software engineers (sometimes these are the software architects) need to be invited "to the system table" for the earliest discussions about a potential solution

[6] Note: Software architects are occasionally assumed to be just software developers with improved "soft skills" (John Klein, personal communication, 9 November 2018). While soft skills are indeed important for any kind of architect, software architects also need to have specific technical skills in addition to programming.

to a customer problem. Ideally this will occur before the program is broken into pieces to be let separately. This is because software will be providing the interfaces among those pieces, and splitting the right way will make the software, and even the non-software, parts of the program much easier to implement than if the pieces are broken apart in a way that makes software very difficult.

Systems engineers also need to know what software engineers are asking for when they ask for data. For example, one responded, "Tell me what I need to know … not just what sensor data to send, but where do I get it, how often, to what accuracy, how often transmitted, validation, ack/nack, etc."[7]

Systems engineers are primarily responsible (with input from all disciplines) for these tasks (note: numbers refer to tasks in Table 4):

- Design the system initially, in a preliminary way, down to the level of software, physical systems components, and system-wide concerns. Create requirements for software, components, and interfaces. Identify tradeoff factors (including software-related). Ensure that software engineers participate in allocation decisions (2, 4).
- Coordinate allocation of requirements to software and to physical subsystems, units, and modules. Liaise with disciplines, ensuring software is aware of pertinent decisions. Respond to software engineers' requests for information (3).
- When software engineers need information that is not yet available, such as how often a sensor measurement should be read and how long it must be stored, coordinate with them to determine the best nominal information and a method and time when the information may be updated.
- Make defensible decisions, documenting the rationale via trade studies (or other means) (3).
- Ensure correct interfaces to hardware and to other systems in the environment, at a high level. Systems engineers are responsible for balancing technical details, budget, and schedule across competing demands. Although systems engineers are not expected to understand the workings of software completely, they are required to understand them well enough, and to partner with software enough, to balance software against other aspects of the system. This requires a transparent working relationship in which both parties agree on system and project goals as well as their own contributions to them (4).
- Perform engineering, including evaluation of safety and security (4)—especially analyses regarding sociotechnical, safety, and security issues (2)—and ensure those issues are respected in the system design (2, 3).
- Identify, tally, manage, and assess system-level risks (2, 3, 4).
- Interface with customers at a high level (customers, agencies, managers, and organizations) to understand the value stream (3, 4, 7). This includes ultimate system verification and customer validation (5). This is in contrast to software engineers' responsibility for coordinating with users.

[7] J. Hudak, personal communication, 11 July 2018.

- Maintain expertise in customer (3), domain, total system (4), and systems engineering processes and technologies.
- Manage systems engineering (6–10), including the systems engineering team, the systems engineering process (8), systems engineering schedules and plans (8), and configuration and information management (10).

d. Learn from Each Other's Methods

Systems engineers should use methods invented by software engineers. Software engineering methods that have been adapted to systems engineering are as follows (numbers refer to lines in Table 1) (SEBoK Wiki 2018, credited to Fairley & Willshire 2011):

- Model-driven development (relates to model-based systems engineering), but systems engineering also looks intensely for things that fall in cracks. Document each step (2, 4).
- UML (software) – SysML (systems) (2, 4)
- Use cases (2) for expressing requirements
- Object-oriented design (2)
  (Note: There is an INCOSE working group on this topic.)
- Iterative development (2)
  (The INCOSE Agile Systems & SE Working Group touches on this topic.)
- Agile methods (the same INCOSE WG) (2, 3)
- Continuous integration (3)
- Process modeling (8)
- Process improvement (8)
- Incremental verification and validation (5, 7, 8)

Software engineers should use methods invented by systems engineers. Methods developed within systems engineering that are or should be adapted to software engineering include the following (SEBoK wiki 2018, credited to Fairley & Willshire 2011):

- Stakeholder analysis (3, 4)
- Requirements engineering (2)
- Functional decomposition (2, 4)
- Design constraints (4)
- Architectural design selection from options (2)
- Design criteria and utility functions (3, 4)
- Design tradeoff methods (3)
- Interface specification (2, 4)
- Traceability (2, 3, 4)
- Configuration management (3)
- Systematic verification and validation (5)

e. Ask Questions of the Other Discipline

INCOSE Past President Bill Schoening suggested that systems engineers could best engineer the system-wide aspects if they asked penetrating questions that "illuminate serious underlying issues

without sounding confrontational" (Schoening 1997). In this spirit, both systems and software engineers should try to understand the other's work through specific questions such as the following.

Questions systems engineers should ask of software engineers. In addition to asking about cost, schedule, quality, and test, systems engineers should ask:

- What is the software development process? What inputs is it expecting, and what outputs is it planning? What is the schedule? (7)
- What major software architectural decisions are needed, and when? What impact will the various options have at the system level? (2)
- What procedures reduce risk of cyberattack, and how might they change? (2)
- What makes development difficult? How could you change that if you could? (8)
- What software risks could become system risks? (3, 4)
- What do you need from me, the systems engineer, and when? (3)
- Will this change I'm asking you to make be a small change? Or might it become a huge problem? (3)
- What decisions must be made to ensure that software is not unreasonably constrained, so developers will be able to create the software economically, and it will serve the operational needs correctly, including adaptation, reconfiguration, and graceful degradation? (3)

Questions software engineers should ask of systems engineers. In addition to asking what the software requirements are, software engineers should ask:

- How well versed in systems and software engineering are the project leaders and managers? How about the customers? (3)
- What information will the project require from software engineering, and when? (3)
- To what level of detail will you, the systems engineers, decompose the system before bringing in software architects? Can the software architects be part of the entire system design process? (2)
- What groups and what systems will this system have to interface with, and to what specifications? (3, 4)
- Who can I talk to, to understand the customer value stream better? (3)
- Who can I talk to, to understand how each sensor and actuator works? (2)
- What "big picture" or overall analysis will be requested and when? (2, 4)
- How final are these requirements, or when will they become final? (2)
- What do you need from me, the software engineer, and when? (3)

These questions should help build the transparent, open relationship that will help make CPSs functional, safe, and efficient throughout this century.

f. Ensure Requirements Are Handled Correctly

The complexity of today's requirements merits special mention. Requirements consist of user requirements, which tend to be feature-oriented and thus easy for software engineers to address, and system requirements. System requirements are either functional or non-functional (the latter are called "quality attributes" in the software world). While two of these nonfunctional requirements

types (reliability and availability) can be quantified and have well-accepted quantification algorithms, assessing other nonfunctional requirements requires engineering judgment. Software engineers prefer not to deal in judgments and tend to toss out such requirements early in the software requirements definition process. In contrast, systems engineers apply "engineering judgment" all the time. Such judgments are important when validating and verifying the system as a whole as well as compliance with quality attribute requirements. Both should be up front with the other about what is or is not known, qualitatively and quantitatively.

Moreover, a joint decision also has to be made when requirements conflict. Relative to a product, security requirements are often at odds with performance or usability. Where's the optimum compromise? This is not a question that can be solved by one group or by fiat. Thus software engineers need systems engineers to understand their concerns at a high level and bring them to the right management and sponsor personnel.

g. Systems Engineers Must Still Do the "Other"

Systems engineers must still do the "other": necessary tasks that aren't being done by subsystems and disciplines. Historically that has meant, for example, verifying that anomalies are being examined by the correct disciplines and subsystems, participating in investigations when the correct disciplines had not yet been determined, and leading the investigation into anomalies not addressed by anyone else and anomalies with clear system responsibilities. Some of the changes now and in the future are as follows.

Systems engineering manages the things that there aren't tools for. Much of systems engineering is human-facing: understanding, convincing, and discovering. Sometimes other engineers recognize only concrete issues, so systems engineers have to initiate work where the issues are not clear. If a tool is created to address a systems engineering responsibility, the importance of that responsibility shifts from doing the calculation to getting the right input and determining the meaning (and recipient) of the output, rather than using the tool.

Systems engineering addresses non-deterministic behaviors. Once software requirements are modeled and ready for implementation, the implemented software ordinarily has only deterministic behaviors. Its functions and outputs can be predicted and measured. When the system as a whole, however, is complex, then system behaviors may be non-deterministic. Systems engineers have been taught during their career how to recognize, characterize, and integrate non-deterministic elements, such as humans. Some software engineers consider that dealing with non-deterministic elements is "someone else's job." If this is the case, the systems engineer must understand and address the non-deterministic behaviors of the software.

Systems engineers engineer emergence. Complex systems and integrated systems have emergent attributes, characteristics, properties, and behaviors. Systems engineers must recognize, categorize, record, manage, and engineer these emergent behaviors to serve the needs of the customer and the operators or users and to provide value to the organizations that pay for the system. This is a broader viewpoint than most software engineers take.

7. Vision

A vision of a high-performance system–software interface for a CPS is as follows:

- Systems engineers and software engineers work closely together and ensure that the best CPS is designed, built, and maintained.
- A chief software architect and chief systems engineer (or equivalent titles) coordinate regularly to benefit the CPS and its customers. Systems and software engineers jointly plan what information they need and when, and what they can provide and when.
- Timely trade studies, performed jointly, ensure affordability.
- Software architectural concerns are known and are satisfied during system architecture development.
- Software people remain up to speed with a rapidly evolving knowledge base while systems people remain knowledgeable about a broad domain and customer.
- Methods of identifying and escalating risks are jointly determined and used.
- System designs are developed in modeling tools that interface seamlessly with the modeling tools used by software engineers.
- Systems engineers maintain responsibility for the non-deterministic and emergent needs of the system, while software engineers help ensure their deterministic and evolving software meets those needs as much as possible.

8. Conclusion

Both systems and software engineering are required for future systems. Their work must be collaborative, because without software there is no new capability; and without good systems engineering the integration is problematic, and the delivered system may not meet customer needs. Both systems engineers and software engineers need to learn what they can about the other discipline and establish coordinating relationships. Ask questions throughout, to ensure the most positive outcome for the system.

Acknowledgments

Copyright 2019 Carnegie Mellon University and Michael E. Pafford. All Rights Reserved.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

DM19-0288

References

Evers, Marco. “Two Crashes in Five Months: What’s Wrong with Boeing’s 737 Max 8?”. Spiegel Online <http://www.spiegel.de/international/europe/what-s-wrong-with-the-boeing-737-max-8-a-1257608.html>

Fairley, RE & Willshire, MJ 2011, ‘Teaching systems engineering to software engineering students’, *24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T), May 22-23, 2011, Honolulu, HI*, pp. 219-226.

GAO 2017, *High-Risk Series. Progress on many high-risk areas, while substantial efforts needed on others*, GAO-17-317, February 2017, Washington, DC.

Haskins, C (ed.) 2006, *Systems engineering handbook*, version 3, INCOSE, San Diego.

IEEE 2017, *ISO/IEC/IEEE 24765:2017 Systems and software engineering—Vocabulary*, ISO, Geneva, Switzerland.

———, *IEEE Code of Ethics*, IEEE, viewed 16 November 2018, <https://www.ieee.org/about/corporate/governance/p7-8.html>.

Illinois Institute of Technology n.d., *Engineers and legal issues*, Center for the Study of Ethics in the Professions, viewed 9 March 2019, <http://ethics.iit.edu/projects/engineers-legal-issues>.

INCOSE n.d., *What is systems engineering?*, INCOSE, viewed 9 March 2019, <https://www.incose.org/about-systems-engineering>.

——— 2018, *Agile Systems & SE Working Group Mission & Objectives*, viewed 10 March 2019, <https://www.incose.org/incose-member-resources/working-groups/transformational/agile-systems-se>.

Koopman, P 2014, ‘A case study of Toyota unintended acceleration and software safety’, Slide Presentation, viewed 16 January 2018, <https://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf>.

Lunney, K 2018, ‘Is systems engineering ready for the future?’ *13th IEEE System of Systems Engineering Conference, June 19-22, 2018, Paris, France*, slide 42.

Maier, MW 1998, ‘Architecting principles for systems-of-systems’, *Systems Engineering*, vol. 1, no, 4, pp. 267-284.

Merriam-Webster 2019, ‘Software engineering’, viewed 10 March 2019, <https://www.merriam-webster.com/dictionary/software%20engineering>.

Miller, Tim. “NCEES discontinuing PE Software Engineering exam” Posted March 13, 2018. < https://ncees.org/ncees-discontinuing-pe-software-engineering-exam/>

Naur, P & Randell, B (eds.) 1969, *Software engineering: Report on a conference sponsored by the NATO Science Committee, Garmisch Germany, 7th to 11th October, 1968*, NATO, Brussels, Belgium, viewed 16 November 2018, <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF>.

ODUSD (A&T) Systems and Software Engineering 2008, *Systems engineering guide for systems of systems*, version 1.0, ODUSD(A&T)SSE, Washington, DC.

Schoening, WW 1997, ‘Penetrating the technical fog’, *INCOSE Insight*, Summer 1997.

Sheard, SA 1996, ‘Twelve systems engineering roles’, *INCOSE International Symposium*, vol. 6, no. 1, pp. 478-485.

——— 2014a, ‘The changing relationship of systems and software in satellites: A case study’, *SEI Insights*, viewed 10 March 2019, <https://insights.sei.cmu.edu/sei_blog/2014/07/the-changing-relationship-of-systems-and-software-in-satellites-a-case-study.html. 2014a>.

——— 2014b. ‘Needed skills for managing complexity’, slide 19 of IEEE webinar: Complexity, Systems, and Software.

——— 2014c, ‘Needed: Improved collaboration between software and systems engineering’, *SEI Insights*, viewed 10 March 2019, <https://insights.sei.cmu.edu/sei_blog/2014/05/needed-improved-collaboration-between-software-and-systems-engineering.html. 2014c>.

SEBOK (Systems Engineering Body of Knowledge) Wiki 2018, viewed 24 December 2018, <https://www.sebokwiki.org/wiki/Systems_ Engineering_and_Software_Engineering>.

Stroud, F 2019, ‘Software engineer’, *Webopedia*, viewed 10 March 2019 <https://www.webopedia.com/TERM /S/software-engineer.html>.

Biography

Sarah Sheard, INCOSE Fellow, earned INCOSE’s 2002 Founder’s Award and a CSEP certification. A member of INCOSE since 1992, she heads INCOSE’s System and Software Interface Working Group. She has over 40 systems engineering publications, including “Twelve Systems Engineering Roles,” “The Frameworks Quagmire,” and “Principles of Complex Systems for Systems Engineering.” At the SEI, Dr. Sheard is a systems and software engineering researcher and consultant. Previously she was a consultant and teacher of systems engineering at the Systems and Software Consortium, and a software systems engineer at Loral/IBM Federal Systems and an aerospace systems engineer at Hughes Aircraft Company. Her PhD, which looked at system development complexity, is from the Stevens Institute of Technology.

Michael E. Pafford has over 45 years of military, government, industry, and research center experience in the analysis, architecting, design, development, testing, operations, and management of complex socio-technical and cyber-physical system solutions. He holds a BS from the University of Maryland University College and an MS from the Naval Postgraduate School. He has been a member of INCOSE since 1998 and is a Past President of the Chesapeake Chapter. For 10 years, he taught Software Systems Engineering at Johns Hopkins University. He has also facilitated tutorials and workshops in using the Lean Startup Method and Agile to improve Initial Project Planning.

Mike Phillips is a principal engineer who helps government clients adopt SEI technologies and other capabilities. He was previously the SEI’s CMMI Program Manager, leading the creation of the CMMI Product Suite, and is an author of *CMMI for Acquisition*. Previously, he led a program team that developed effective techniques to help organizations accomplish Technology Change Management. As an Air Force senior officer before joining the SEI, he led the development and acquisition of the software-intensive B-2 Spirit stealth bomber using integrated product teams. During his Pentagon tour, his office provided oversight within the Pentagon of the highest security, special access programs at both Air Force and OSD levels. He has four master’s degrees, in nuclear engineering, systems management, international affairs, and strategic studies.