JOURNAL OF AIRCRAFT
Vol. 48, No. 3, May–June 2011

Systems Engineering Perspectives on Technology Readiness Assessments in Software-Intensive System Development

Peter Hantos*
The Aerospace Corporation, El Segundo, California 90245

DOI: 10.2514/1.C000276

Technology readiness assessments as formal technology maturity validation tools play an important role in system development and acquisition. Because of the particular diversity of technologies in software-intensive systems, such assessments pose a special challenge. This paper's objective is to show how technology readiness assessments control the technology transition process from research to productization and to demonstrate that effective execution of this transition requires sophisticated systems engineering perspectives. To achieve this objective, a rigorous attempt was made to seek out and clarify the ambiguous elements of the process and to provide tangible guidance on the topic. Examples from the emerging technology-rich domains of unmanned aerial vehicles are used to facilitate a deeper understanding of the concepts introduced.

I. Introduction

PRODUCT development success is highly dependent on the ability to capitalize on innovative technologies. There is a delicate balance of risks and opportunities, which has to be understood, and the transition from research to development needs to be properly managed. The assessment of technology readiness poses serious challenges within the defense acquisition system. According to the 2006 Defense Acquisition Performance Assessment Report [1], the inability to define and measure technology readiness led to numerous decisions to incorporate immature technology upon entry to the system design phase, subsequently leading to technical difficulties with far-reaching consequences. Both commercial and defense approaches to carrying out technology readiness assessments (TRAs) wrestle with numerous ambiguities that contribute to these problems, a detailed analysis of which is beyond the scope of this paper. Our focus is on a particular aspect of TRAs, which is similar in market-driven and military contexts alike. Technology, or technologies, usually relates to methods of a particular technical discipline. In the case of software-intensive systems, these are hardware and software. Traditionally, the associated maturity assessments are carried out in a stove-piped fashion. However, for software, separating these assessments in isolated silos renders the results inadequate and misleading. Using primarily aerospace considerations, it will be argued that successful TRAs of software-intensive systems require a more holistic, comprehensive, and systems engineering perspective. Choosing case studies from the unmanned-aerial-vehicle (UAV) area is not accidental. UAVs are the favorite weapons of choice in the new, asymmetric warfare situations, and demand for them is unprecedented. Quoting Tim Owings, manager of the U. S. Army aviation programs, on Army drones in Afghanistan [2]: "It took the Army 13 years to fly the first 100,000 hours. From the start of the war, it took 10 months to fly the next 100,000 hours." In fact, Lieutenant General Norman R. Seip, quoted in the same article, predicted that over the next five years, more unmanned vehicles will be flown than manned aircraft. The new missions are also increasingly complex. Swarms of UAVs are engaged in elaborate missions, controlled from multiple ground stations, and integrated with miscellaneous ground, air, naval, and space assets of the theatre in a system-of-systems (SOS) situation.

Are aircraft and satellites software-intensive systems? What is a software-intensive system anyway? There is no commonly accepted criterion to label a system software intensive, and there is no common denominator that would facilitate the necessary comparison between hardware and software. For example, in the case of UAVs, weight is a key size measure from a hardware perspective, while source lines of code (SLOC) are used as the primary size measure for software. SLOC are meaningless for hardware, and software is weightless; as a result, neither measure is appropriate to express the relationship. Also, hardware can function without software, but software always needs a hardware platform to exist. Consequently, a classification separating pure hardware and software systems would also be false. The F-22, the U. S. Air Force fighter plane, is famous for its stealth technology and ability to disguise its infrared emissions, making the aircraft harder to detect by heat-seeking missiles. The F-22 onboard computers contain roughly 2 million lines of code (LOC) software, implementing 80% of the overall functionality of the aircraft. These numbers, even separately, should be quite convincing that the F-22 is, indeed, a software-intensive system. A word of caution is warranted regarding the quest for an exact metric. Using the software-intensive label to emphasize the need for a software insight during evaluation is beneficial. On the other hand, trying to use it as a formal, mechanical discriminator to decide if such insight is needed would be counterproductive. Systems engineers need to learn from the currently raging debate on the use (and abuse) of the concept of statistical significance [3]. Authors of the referenced book warn that the "magic 5%" as a statistical significance threshold employed in many situations is arbitrary and, as they repeatedly demonstrate, has resulted in unsavory data manipulation by medical researchers and pharmaceutical companies. The lesson they impart is that data need to be examined on their own merits and that statistical tools should not be used blindly. Similarly, systems engineers should be concerned with understanding the true impact and contribution of technologies rather than pursuing a seemingly simple, but ultimately misleading, metric.

We also need to fight some historical ballast, consequences of the hardware bias in systems engineering. All systems in the distant past were hardware systems, so systems engineering concepts and approaches were defined for hardware. A good example is the Defense Support System (DSP). The DSP is an early U. S. Air Force reconnaissance satellite system used to detect intercontinental ballistic missile launches from which the first satellite was launched on 6 November 1970. The DSP satellite had zero LOC onboard, making it a clear-cut hardware system. Modern large-scale systems, however, always rely on a substantial amount of software. Space-based infrared radar (SBIRS) is the new system that is slated to

Received 25 January 2010; revision received 13 July 2010; accepted for publication 19 August 2010.

*Senior Engineering Specialist, Software Acquisition and Process Department, 2310 East El Segundo Boulevard, Mail Stop M1-112.

replace the aging DSP constellation. SBIRS satellites run about 400,000 LOC software on their numerous onboard computers, supported with ground stations containing as high as 1.4 million LOC. The sheer magnitude of the SBIRS reliance on software is convincing, making its classification as a software-intensive system simple. (One could argue, though, that even DSP was a software-intensive system, since DSP ground contained 0.75 million LOC software, and the combination of satellites and ground stations could be viewed holistically as a software-intensive system.)

What are the signs of hardware bias in systems engineering? Systems engineers have traditionally been involved in requirements engineering and system integration and test activities. These functions are at the front end and the back end of the systems engineering life cycle, respectively. The front-end/back-end designation is significant, because the prevailing life cycle model (called the Vee model [4]) is inherently a sequential model, where software-related activities are hidden deeply in the bottom of the V structure of the life cycle model. A state-of-the-art systems engineering function must now go beyond these limited roles. Systems engineering should represent the technical conscience of software-intensive system development, providing technical support to program/project management in fulfilling an overarching cradle-to-grave responsibility. What are the implications of this overarching role? The idea behind relying on systems engineering at the early phases of the system life cycle is to bring in appropriate, holistic approaches to comprehend the complexity of the designed system and to interact with the customer to determine the required capabilities. However, even at this early stage, potential discipline-specific implementation options need to be considered; consequently, a good understanding of hardware and software concerns is a must, even in the early phases.

As the project progresses, the division between disciplines becomes more pronounced. In these stages, systems engineering must act as a balancing force, facilitating both the technical aspects of the trades and the logistics aspects of the technical interactions between the contributing disciplines. Development of large-scale software-intensive systems inherently requires concurrent engineering. As a result, for most of the life cycle, hardware and software development are carried out concurrently, in parallel process streams, following different life cycle models. The coordination and harmonization of these hardware and software life cycles should also be a systems engineering responsibility. Risk management is a particularly sensitive management area. For various reasons, software risks are usually not as visible to program management as hardware risks. Therefore, a major challenge for systems engineers is to ensure that the risks of all contributing disciplines, including software, are well comprehended and treated as equally important. Software integration, in reality, is not taking place as a big bang during the last phase of the development life cycle. Software is developed iteratively and integrated and tested incrementally. As a result, systems engineering efforts must address both cross-domain coordination and cross-domain integration. In-domain integration, however, is a special case. In the hardware domain, it is traditionally considered a systems engineering task, while in the software domain, it transparently blends into the software development activities. Even where systems engineering efforts are defined in a holistic discipline-neutral way, to counterbalance the underlying hardware bias, software-specific amplifications are needed to help comprehend the full depth and complexity of activities.

This paper is not intended to be a complete tutorial on TRAs. Most of the logistics and policy issues will not be covered. However, enough detail is provided to develop the necessary level of appreciation for the importance of the process. Although this paper discusses an array of technologies, it is not intended to be a tutorial on any of them. All technologies introduced will be discussed only to the extent needed to demonstrate successful technology transition, and management requires substantial contributions from the systems engineering discipline.

The rest of the paper is organized as follows. Section I describes technology readiness principles, Sec. II details the concept of software technology readiness, and Sec. III provides an overview of cross-domain concerns during TRAs. Finally, four case studies are discussed in Sec. IV to further clarify the issues raised.

II. Technology Readiness Principles

A. Definition of Technology

Even though the word technology is widely used, there is no agreement on what it means. Some people equate technology with science, while others might use the term as a synonym for technical. However, we cannot leave its definition ambiguous if we are planning to assess its readiness. Failing to have an accepted definition, technology assessments might become excessive and duplicate other existing assessments, or we might run into the opposite problem; that is, we fail to assess essential aspects of product development. The Merriam–Webster Dictionary[†] offers the following definition: technology is 1) "the practical application of knowledge in a particular area," 2) "a capability given by the practical application of knowledge," 3) "a manner of accomplishing a task especially using technical processes, methods, or knowledge," and 4) "the specialized aspects of a particular field of endeavor."

Beyond the technology vs technical terminology ambiguity, the systems engineer needs to deal with several other ambiguities. Before we can discuss those, let us review the corresponding, relevant definitions for product and system, also from the Merriam–Webster Dictionary. A product is 1) "something produced; especially a commodity," 2) "something (as a service) that is marketed or sold as a commodity," and 3) "something resulting from or necessarily following from a set of conditions." A system may be 1) "a regularly interacting or interdependent group of items forming a unified whole," 2) "a group of devices of artificial objects. . .serving a common purpose," and 3) "an organized set of doctrines, ideas, or principles."

In reviewing these definitions, we can conclude that technology is clearly a conceptual term, while product in the industrial environment primarily refers to tangible objects. System can go either way, depending on the nature of its building blocks. Nevertheless, systems engineering's focus is systems construed of tangible objects and not conceptual ones: for example, the capitalist system that economists would be concerned about. The major source of confusion is that technology is carried by and manifested via tangible objects. After discussing the fundamental concepts of TRA, we will revisit the controversial issues stemming mostly from this ambiguity.

B. Technology Readiness and Assessment

The following discussion is based on a commercial market-driven product development environment [5]. In this context, technology readiness is defined as a state from which a product can be designed and implemented with predictable performance, cost, delivery, and quality characteristics. Technology readiness level (TRL) is the measure of technology maturity. High TRLs indicate a high level of confidence that no special solutions will be invented beyond normal design engineering practices to satisfy the product requirements. According to this commercial scheme, TRLs are defined for hardware only, and technology readiness is demonstrated by meeting the following five criteria:

1) Failure modes are identified.

2) Critical parameters that control the failure modes are identified.

3) Safe operating latitudes for failure modes and their controlling parameters are optimized.

4) Manufacturability requirements are met.

5) Hardware is capable of delivering the required performance in the absence of identified failure modes.

Note that, here, technology maturity is defined in statistical process control terms: a high TRL implies that the centerline of design capability is centered on the manufacturing variability, and the distribution of manufacturing variability remains within the

[†]Data available at http://www.merriam-webster.com/dictionary/technology [retrieved 10 January 2010].

design latitude. It is obvious that this approach cannot be easily adapted or interpreted for software, since software is not manufactured. In the 1990s, NASA standardized a nine-level scheme [6], which later became the foundation of the TRLs of the Defense Acquisition Framework [7]. This scheme is also hardware centered but, with some abstraction and interpretation, it can be used for software:

1) TRL 1 represents basic principles observed and reported.

2) TRL 2 represents a technology concept and/or application formulated.

3) TRL 3 represents an analytical and experimental critical function and/or characteristic proof of concept.

4) TRL 4 represents a component and/or breadboard validation in a laboratory environment.

5) TRL 5 represents a component and/or breadboard validation in a relevant environment.

6) TRL 6 represents a system/subsystem model or prototype demonstration in a relevant environment.

7) TRL 7 represents a system prototype demonstration in an operational environment.

8) TRL 8 represents an actual system completed and qualified through test and demonstration.

9) TRL 9 represents an actual system proven through successful mission operations.

To determine these ratings, the assessors examine program concepts, requirements, and demonstrated technology capabilities. The assessment is not intended to predict future performance of the evaluated technologies, nor does it assess the quality of system architecture, design, or program plans. While a TRA is a mechanism to identify technology risks, it is different from the well-known conventional risk management approaches. The result of a TRA is a single number on a 1–9 ordinal scale. This metric does not quantitatively reflect either the likelihood of attaining the required maturity or the impact of not achieving the required maturity. The constrained nature and the ambiguous interpretation of the TRLs pose a serious challenge for program management. In defense acquisition, the prevailing rule is to use TRL 6 as the condition to initiate the actual acquisition program and transition from technology development to engineering and manufacturing development. However, aggressive program managers would prefer to incorporate technologies with a lower TRL 5 rating, while according to more risk-aversive opinions, only TRL 7 should suffice. With respect to the logistics of these TRAs, current U. S. Defense Acquisition policies and public law require two flavors. Heavyweight TRAs are conducted by a special, independent review panel (IRP) in every major acquisition milestone, and the results of such assessments are to be provided to the milestone decision authority (MDA). The law also mandates a series of calendar-year-based technology reviews during the whole acquisition life cycle, beginning at the early stages of the acquisition (material solutions analysis phase) by the Office of the Director, Defense Research and Engineering (ODDRE) of the DOD. IRPs and the ODDRE reviewers are obligated to use the earlier mentioned DOD Technology Readiness Assessment Deskbook [7].

The commercial market-driven world uses TRLs differently. The TRA mechanism is primarily applied to control the productization of in-house technologies. In large companies, there is a pronounced organizational separation between their research centers and development divisions, and the TRA functions as a gating mechanism between the two kinds of entities. At the same time, executives of these companies might also aggressively pursue the development of certain new products but, via balanced portfolio management, these companies can afford to initiate product development more conservatively, mature the new technologies longer in their research centers, and experiment with breakthrough technologies in only a few products. In such settings, intermediate rating levels are not particularly useful; the technologies considered must fully satisfy all of the readiness criteria before productization can start. On the other hand, small companies and startups do not have this luxury. Their livelihood depends on a limited number of technologies and, in such firms, the boundaries between technology development and product development are blurred. In most cases, they would not bother with TRAs, and even early assessments and demonstrations would be directly product oriented. The lesson systems engineers need to take away from this analysis is similar to what was said about codifying software-intensive systems; that is, instead of looking for a simple, silver bullet solution, they should focus their energies on the true understanding of the technologies involved. From a logistics perspective, the implication is that market-driven companies mostly do only one high-ceremony TRA, while the DOD conducts a series of the earlier discussed event-driven and calendar-driven assessments.

Finally, it is important to note that, even though most commercial approaches are different from the DOD-embraced approach, TRAs in both domains share an important characteristic that, unfortunately, is also very often misunderstood. These TRAs are always conducted in the context of the development or acquisition of a specific product; a TRL rating is never an abstract, general attribute. Consequently, technology maturation life cycles, like the one introduced by Redwine and Riddle [8], do not apply. Redwine and Riddle's model assumes a three-phased life cycle with subsequent phases of technology development, exploration, and maturation during use. The maturation phase in their model assumes 70% or more industry use of the technology in question. On the other hand, the demonstration of a technology in a relevant environment is an essential condition in DOD TRAs. While use in other contexts certainly increases the comfort level with the technology in question, the extent of industry use is irrelevant in determining the actual TRL rating.

C. Critical Technology Elements

It is neither feasible nor necessary to evaluate all technologies in a program. Critical technology element (CTE) is a defense acquisition term, and it refers to those technologies that will be subjected to the TRA process. According to [7], a technology element is critical if 1) the system being acquired depends on the element to meet operational requirements with acceptable development costs, schedule, and production and operation costs and 2) the technology element itself or its application is either new or novel. While the term CTE is not used in commercial environments, a similar need for identifying the list of candidate technologies exists. In such settings, the primary source of finding candidate technologies is called the vector of differentiation. The vector is based on the features of the planned product that would enable the firm to capture new markets or to expand the sphere of influence in existing ones. The equivalent idea in defense acquisition is the use of key performance parameters (KPPs). KPPs are thresholds applied to the measures of performance, which are quantitative measures of the lowest level of required physical performance or physical characteristics. However, neither the use of the vector of differentiation nor KPPs are sufficient to identify all CTEs for similar reasons. Both are defined on a high level and can only facilitate the identification of a limited number of needed technologies. As a result, many of the future implementation details and their corresponding technology needs at this stage remain hidden. The DOD Deskbook [7] also recommends the use of the contract's work breakdown structure (WBS) or the system architecture to identify CTEs; concerns regarding this approach will be discussed in the following section.

D. Using Work Breakdown Structure or System Architecture

The WBSs examined here are based on [9], but they were slightly modified to better facilitate the discussion. Table A1 in the Appendix is a WBS for a UAV system, including air vehicle, a ground segment, support equipment, and facilities. Table A2 is the WBS of a space system, including space vehicles, ground systems, and launch vehicle. Table A3 shows how the combination of the previous WBSs and the addition of some selected, additional systems result in a SOS WBS. A closer analysis of these WBSs reveals that the contract's WBS might be satisfactory in identifying hardware CTEs, but it is inadequate for identifying software CTEs. The WBS is derived via a hierarchical decomposition process, which stops before software

details are explored and documented. The space system WBS goes the deepest, but even at level 5 (which would correspond to the sixth level of the SOS WBS), the items are still very high level, with quite unexpressive names for the software build designations. For the sake of brevity, no system architecture examples are provided, but it is easy to concede that the same limitations apply.

III. Software Technology Readiness

A. Definition of Software Critical Technology Elements

TRA is an imposed process-improvement step, not an inherent engineering process. Therefore, we do not have, in the hardware or the software domain, any design/development process that could not be executed without including a maturity assessment step. In commercial market-driven companies, TRA is imposed by executive management, while in defense acquisition, it is mandated by the U. S. Congress via public law. In the presence of ambiguous definitions, it is an extreme challenge for program management to determine what should be the object of these imposed reviews and what should be handled during the normal course of technical reviews that mark the milestones of the engineering processes. Unfortunately, the situation is even more ambiguous in software. An important question that needs to be answered is whether the software itself that is to be written should be considered a CTE. Strictly speaking, software in general could be considered a technology, since it satisfies the earlier presented Merriam–Webster definition. However, this is not very helpful from a TRA perspective, so we will use the definition introduced in [10]:

> Software technology is the theory and practice of various sciences (including computer, cognitive, statistical sciences, and others) applied to software development, operation, understanding, and maintenance. Specifically, software technology is any concept, process, method, algorithm, or tool for which the primary purpose is the development, operation, and maintenance of software-intensive systems.

Software technology may include the following:

1) The first kinds of technologies that may be included are directly used in operational systems, such as two-tier or three-tier software architectures, public key digital signatures, service-oriented architecture (SOA), etc.

2) The second kinds technologies that may be included are used in tools that produce, help to produce, or maintain operational systems, such as graphical user interface builders and miscellaneous code generators, cyclomatic complexity analyzers, programming languages and compilers, etc.

3) The third kinds of technologies that may be used are process technologies that make people more effective in producing and maintaining operational systems and tools by structuring development approaches or enabling analysis of systems and product lines. Examples include the Personal Software Process (PSP), Cleanroom Software Engineering, CMMI®, etc.

For software TRA purposes, software CTEs will be classified in the following three categories: 1) software algorithms for the objective system; 2) to be reused, commercial off-the-shelf (COTS), and government off-the-shelf (GOTS) software for the objective system; and 3) tools for software development and maintenance (these are usually COTS as well).

From a TRA perspective, software to be reused or COTS/GOTS software to be inserted in the objective system must be distinguished from software to be written. The focus of maturity assessments for software technologies is not simply the technical artifacts but the knowledge embedded in those artifacts and the knowledge required for their effective use. Consequently, during a TRA, such software components are evaluated from a technology perspective only (i.e., as carriers or hosts of important algorithms, methods, or solutions). It is understood that for the final design decisions, detailed, comparative trade studies, cost and availability analyses, and a benchmarking of potential vendors are essential as well, but such activities would not be in scope for a TRA. This is the right time to refer back to our discussion on the differences between technology and product. Since they are essentially different concepts, the associated life cycles (i.e., technology development life cycle and product development life cycle) are also different, and so should be the attached maturity metrics. The lack of clarity on this issue is the source of endless debates regarding the scope of TRAs, leading to either the proliferation of candidate CTEs or the omission of important ones, depending on the debaters' idea of software technology.

B. Algorithms

The confusion over types of algorithms is a major source of ambiguity surrounding software technology. According to the conventional definition, an algorithm is a sequence of finite instructions. It is formally a type of effective method in which a list of well-defined instructions for completing a task will, when given an initial state, proceed through a well-defined series of successive states, eventually terminating in an end state. From a TRA perspective, it is necessary to distinguish between user domain-specific algorithms and technology domain-specific (i.e., hardware and software) algorithms. User domain-specific algorithms implement various tasks in the user's domain rather than the implementer domain, while software algorithms relate directly to software implementation tasks. User domain-specific or hardware algorithms may be developed and evaluated with software tools, sometimes even by writing some code, but this still would not make them software algorithms.

Another source of confusion is that user domain-specific algorithms might be implemented in software, hardware, or firmware. However, the first-order evaluation of user domain-specific algorithms (i.e., the determination of whether they fit their purpose in the required product or mission context) must be done independently from their future implementation. This separation is necessary, because algorithm fitness evaluation does not require software engineering expertise and is not a software engineering responsibility; the core intellectual content of such algorithms needs to be assessed by the appropriate, functional domain experts. Nevertheless, hardware and software specialists of a TRA team need to analyze such validated, user domain-specific algorithms for possible decomposition and implementation options, because this second round of evaluations may yield specific hardware or software CTE candidates. A typical systems engineering task is the early reconciliation of possible tensions between user domain-specific algorithms and their potential implementations. It is a frequent scenario when the review of implementation plans highlights irresolvable technology conflicts, and the only available solution is to go back and change the original science that has been selected to address product requirements. A word of caution, though, regarding software algorithms. Software development is, after all, about implementing algorithms and, in essence, what we see during software development is a sort of algorithm metamorphosis. Software algorithms are refined, validated, and decomposed successors of user domain-specific algorithms and are specified in engineering, rather than user, terms. Software algorithms are to be programmed in the normal course of software development; in fact, selecting appropriate algorithmic building blocks is a routine, transparent element of programming. Consequently, during CTE identification, the high-impact algorithms need to be distinguished from the routine ones (routine algorithms are those taught in introductory computer programming classes; e.g., manipulating data structures and memory, implementing the program's control flow, converting variable types, etc.). Figure 1 summarizes these ideas; dark nodes represent items that would not be in scope for software TRA.

C. Role of Software Architecture

It has been shown that the WBS is inadequate to identify most CTEs, particularly software ones. The successful identification of critical software technology-element candidates requires the analysis of reasonably detailed software architecture. Of course, the term reasonable is quite subjective. However, in this situation, it simply means that the resolution and depth of the available architecture documentation would determine how far we could proceed with identifying CTEs. The current, prevailing architecture standard for

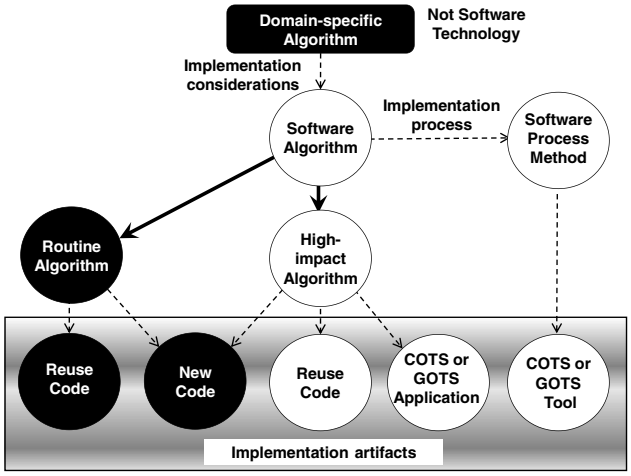

Fig. 1 Classification of algorithms from a TRA perspective.

software-intensive systems is the International Organization for Standardization/International Electrotechnical Commission (ISO/IEC) 42010:2007 [11]. For software CTE identification, the architecture documentation needs to provide appropriate architectural viewpoints of the structural and dynamic aspects of the software-intensive system.

To carry out the necessary environmental analysis according to ISO/IEC 42010:2007, at minimum, the following architectural viewpoints are recommended.

The structural viewpoint addresses the following concerns:

1) What are the elements of the system, including its components?

2) What are the interactions between these components (called connectors by the standard)?

3) How these elements are structurally organized?

The behavioral viewpoint addresses the following concerns:

1) What are the dynamic actions of and within the system?

2) What are the kinds of actions the system produces and participates in?

3) How do these actions relate and what is their ordering and synchronization?

4) What are the behaviors of system components and how do they interact?

The physical interconnect viewpoint addresses the following concerns:

1) What are the physical communications interconnects among system components?

2) What is the layering among system components?

3) What is the feasibility of construction, compliance with standards, and evolvability?

To facilitate early software CTE identification and cross-domain analysis of technology elements, a customized TRA-oriented version of the ISO/IEC 42010:2007 standard's structural viewpoint has been created [11]. Figure 2 depicts a single node, but the use of this viewpoint to depict the physical interconnect viewpoint of networked nodes is straightforward via the use of the indicated bus.

IV. Overview of Cross-Domain Concerns in Technology Readiness Assessments

A. Disciplines Involved in Achieving Technology Readiness Level Objectives

Figure 3 was devised to summarize and highlight the required cross-domain systems engineering activities during the determination of TRLs for a software-intensive system. The basic definitions of TRLs are from [7], but the rest of the table is new. From left to right, the next column shows the evolution of TRL goals, followed by the details of various disciplines that are involved in achieving specific TRL objectives. Please note that cross-domain evaluation systems engineering contributions are needed during CTE identification and selection as well, but those activities precede TRL rating and could not be shown.

B. Technology Readiness Levels of Dependent Critical Technology Elements

The layers of the software internal physical environment that were introduced in Fig. 2 are not independent, even though the original, core TRL rating scheme assumes the independent, implementation domain-specific rating of the CTEs. The hardware–software and software–software hierarchical dependency needs to be considered during CTE identification and TRL rating as well. The concept is illustrated in Fig. 4.

The consequences of this dependency during CTE identification are as follows. The criticality of the various hardware and software elements of the layers needs to be proven or disproven from the bottom up, successively. If any element is identified as a CTE, then all elements above it in the hierarchy automatically become CTEs as well. For example, let us assume a situation where a new operating system is introduced on legacy hardware. The hardware is not a CTE, but the newness of the operating system is obvious, hence its classification as a CTE. However, even if the application software existed before, porting it to the new environment makes it an unprecedented solution and, as a result, the otherwise legacy software also becomes a CTE. A similar principle is applied during the determination of TRLs. In mathematical terms, the following relationships apply:

$$\begin{aligned}&\text{TRL(software applications+human interface)}\\&\quad\leq \text{TRL(tools + system software)}\\&\quad\leq \text{TRL(hardware platform)}\end{aligned}\tag{1}$$

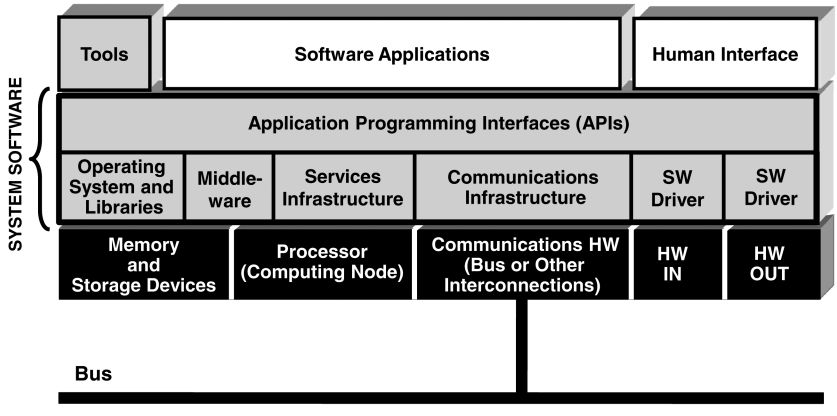

Fig. 2 Simplified structural viewpoint for TRA (SW denotes software, and HW denotes hardware).

TRL	Basic Hardware TRL Definitions From The DOD TRA Deskbook	Basic Software TRL Definitions From The DOD TRA Deskbook	TRL Goals	Knowledge Involved in Achieving Hardware Objectives	Knowledge Involved in Achieving Software Objectives	Systems Engineering Responsibilities
1	Basic principles observed and reported	Basic principles observed and reported	Demonstrate Scientific Feasibility	Natural Sciences	Computer Science	
2	Technology concept and/or application formulated	Technology concept and/or application formulated				Requirements, Trade Studies
3	Analytical and experimental critical function and/or characteristic proof of concept	Analytical and experimental critical function and/or characteristic proof of concept				
4	Component and/or breadboard validation in a laboratory environment	Module and/or subsystem validation in a laboratory environment	Demonstrate Engineering Feasibility	Hardware Engineering	Software Engineering	
5	Component and/or breadboard validation in a relevant environment	Module and/or subsystem validation in a relevant environment		Systems Engineering	Systems Engineering	In-Domain Integration
6	System/subsystem model or prototype demonstration in a relevant environment	Module and/or subsystem validation in a relevant end-to-end environment				Cross-Domain Evaluation
7	System prototype demonstration in an operational environment	System prototype demonstration in an operational, high-fidelity environment	Demonstrate Operational Feasibility	Hardware Engineering	Software Engineering	
8	Actual system completed and qualified through test and demonstration	Actual system completed and mission qualified through test and demonstration in an operational environment		Systems Engineering	Systems Engineering	Cross-Domain Integration
9	Actual system proven through successful mission operations	Actual system proven through successful mission-proven operational capabilities	Demonstrate Operations	Mission Domain	Mission Domain	Mission Domain Demonstration

Fig. 3 Disciplines involved in achieving TRL objectives.

C. Technology Readiness as Emergent Property

To understand that the maturity of technology elements cannot be assessed independently, the recognition of the emergent nature of technology readiness is very important. Emergence is hotly debated in philosophy, biology, and other sciences; here, based on [12], we only provide an essential definition that can be easily applied in systems engineering. Emergent properties are systemic features of complex systems that could not be predicted from the standpoint of a pre-emergent state, despite thorough knowledge of the features of, and laws governing, their parts. In practical terms, we are concerned about the unintended consequences of CTE interplay for the CTEs participating in the dependency chain. The question of what thorough knowledge means still remains. It is very difficult to quantify the depth of thoroughness, so for practical reasons, we might augment the term and refer to reasonable thoroughness. Using the previous processor—operating-system example, one can be intimately familiar with the processor architecture and the internals of the operating system; this familiarity would guarantee reasonableness. However, without further experimentation, it might be still very difficult, if not impossible, to predict their combined behavior. The recognition of emergence will also be important in characterizing the potential interplay between formally independent CTEs, as will be shown in the next section.

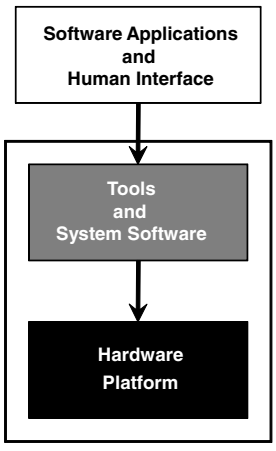

Fig. 4 Dependency of architectural layers from a TRA perspective.

D. Quest for Characterizing System Maturity

Numerous sources in the literature express the need to have a single measure of system maturity as an extension of technology maturity. The idea is that a single system maturity index would make the MDA's job easier at the milestones of the system acquisition life cycle. However, the author's opinion is that the motivation behind such a single measure is misguided. Let us consider a hypothetical space vehicle acquisition, with five CTEs identified in the following areas: solar array, propulsion subsystem, real-time operating system (RTOS), antenna element, and thermal blanket. Let us also assume that a system maturity index has been determined. How would such an index be used? One might speculate that, in the spirit of the current public law, a minimum value of six for the system maturity index would have to be achieved at the entry of the engineering and manufacturing development phase. However, a deeper analysis is needed and a simple go–no-go decision will not suffice, particularly if the value of the index is close to but less than the magic number six. The analysis would have to go back to basics and review the independent status of the CTEs, since they would have to be matured independently. Independent maturation here means that there is no expectation of technical coordination, for example, between the development of the solar array and the antenna element. Unfortunately, there are even more serious reservations about system maturity indices related to the way their determination may be suggested.

The simplest approach is aggregating TRLs. If aggregation cannot be avoided, then one should use the lowest TRL for the overall system or SOS. This approach is based on well-known reliability principles. If all the selected CTEs are indeed critical, then the

weakest link should be used to characterize the system as a whole. While the use of such a number would be theoretically correct, its use could drastically slow down the acquisition; hence, program managers are not in favor of using it. Unfortunately, there is often a temptation to take the arithmetic average of TRLs or apply similarly flawed and even more complicated formulas, as we will see later. However, one must not do arithmetic on ordinal scales under any circumstances, because doing so hides and confuses the impact of the real technology risks. For example, consider two TRLs: TRL 3 and TRL 9. Obviously, the technology with the top rating should not represent any concerns. On the other hand, TRL 3 is quite low, and the associated technology element has only been validated in a laboratory environment. At this point, not only the pace of the maturation but even the ability to ever achieve higher maturation involving a real component in a relevant environment is questionable. Nevertheless, the arithmetic average yields TRL 6, which would indicate to program management that, as far as critical technologies are concerned, it is safe to progress to the next acquisition phase. Let us examine another scenario with TRLs 3 and 8. Again, a CTE with TRL 8 seems to be a safe technology, and TRL 3 is still very low. The average value now is 5.5. Would it be okay to round it up to TRL 6 and let the program proceed, or should it be rounded down, and the program should stay in the technology development phase? The games we are playing here with the fractions totally obfuscate the real, underlying technology issues.

A more sophisticated approach is the system readiness level (SRL) [13]. SRLs are computed in three steps. First, conventional TRLs are determined. Second, so-called integration readiness levels (IRLs) are determined for every possible connection between CTEs, where IRLs are similar to TRLs (i.e., single numbers on a 1–9 ordinal scale). Finally, some matrix operations are applied to the TRLs and IRLs, and an SRL is calculated. The results are normalized, so the computed SRLs are fractions and fall between the 0–1 range. One of the key weaknesses of this approach is the way IRLs are defined and rated. For example, IRL 1 is defined as follows: "An Interface between technologies has been identified with sufficient detail to allow characterization of the relationship." Considering our earlier space example, there is no established connection between the RTOS and the thermal blanket, so we could not even give one, the lowest rating to this interface. As it was shown, the same is true for all of the other CTEs of our example space system. This problem is the reason why the emergent nature of technology readiness and the futility of arithmetic manipulations have been emphasized. Another key deficiency lies with the normalization of TRL and IRL ratings. For example, the 1–9 TRL scale is mapped into a 0–1 range, with equidistant, fractional steps. However, the distribution of represented technology risks is far from equidistant; the risk of not being able to progress to the next technology maturity level always drastically jumps whenever the validation environment changes, and the most prevalent situation is the migration from the laboratory environment to the relevant environment. These risks are highly dependent on the user's domain. Space is the most sensitive domain, particularly if the NASA scheme is used, considering that NASA only accepts space as a relevant environment for satellite CTEs. (Incidentally, this criterion is unacceptable for military space acquisitions, creating another controversy but, unfortunately, at the time of the writing of this paper, there was no formal military space acquisition policy in effect.)

Integration is a very contentious issue in TRAs, mainly because the term is overloaded or misused. In the SRL scheme, IRLs are defined, from [13], as "a systematic measurement of the interfacing of compatible interactions for various technologies and the consistent comparison of the maturity between integration points." Since technology is a conceptual term, the integration in IRLs must refer to integration in a conceptual sense as well. Unfortunately, in acquisition, the term integration refers to the physical assembly of system components (integration and test life cycle phase/activity). Because it is a risky process, some people want to declare it a CTE. However, normal integration is the source of programmatic risks, and not the technology risks that are supposed to be assessed during a TRA. What kind of integration is implied during TRL analysis? The definitions show a successively higher-level WBS context, implying the partial completion of system integration. This is confusing, because according to the definitions, up to TRL 7, we are still only expecting model or prototype integration; references to the actual system can only be found in the definitions of TRLs 8 and 9. There is no resolution to this logic conflict other than to accept that, even though the TRL scheme is supposedly meant for rating technology elements, TRLs 8 and 9, and possibly even TRL 7, can only be achieved in the presence of the final, objective system.

V. Case Studies

The following case studies are not intended to be complete TRA examples. Their objective is limited to clarifying specific cross-domain systems engineering concerns, and they are simplified to the extent that an in-depth knowledge of the domain is not needed to understand the concepts introduced.

A. Service Network Model for Multi-Unmanned-Aerial-Vehicle Search Missions

The problem, as stated in [14], is as follows. A multivehicle search mission consists of many resources interacting to concurrently execute many tasks in a dynamic fashion. For example, UAVs might leave and rejoin the team, actual network routers might go up or down, missions start and terminate, etc. According to the originally proposed solution, a mission is modeled as a directed application graph, where nodes represent service providers and edges represent services. The search mission algorithm then builds such application graphs dynamically, and it reconfigures them as service providers depart or arrive. Authors in [14] described a custom middleware implementation. Let us examine this solution path first from a TRA perspective. It is clear that the multivehicle search algorithm itself is not a software algorithm, even though software tools would be used to validate it and it would be implemented in software. Consequently, the solution on the top level would have to be evaluated by mission experts. Next, the planned middleware would be new code and, as such, would not be (could not be) the subject of a preliminary technology analysis. However, an alternative approach is available, using SOA for implementing the service network model. A system architecting style, SOA, takes advantage of networking capabilities to integrate applications in a way that is independent of architecture, programming language, development platform, and vendor. Through a set of standard interfaces, services are made available to any consumer willing to follow the rules of interface and consumption. Besides being an architecting style, numerous COTS products are available to support the implementation of such architecture. A COTS SOA framework, besides documenting the standard interfaces, would also provide middleware components to implement a service registry that is needed for the seamless integration, upgrade, discovery, and invocation of services: low-level and user-level alike. Would SOA be a software CTE? Our first reaction is that it should not be. A recent Google search produced 40 million hits for SOA; would that not be proof of it being a mature technology? Unfortunately, the situation is much more complex; using SOA is a risky proposition and extreme caution is needed.

Figure 5 shows how pervasive SOA is and indicates why the special consideration is justified. Also, we need clarity what SOA COTS/reuse assessment attributes are in scope for a TRA, and what SOA inquiries should belong to the routine, programmatic risk reduction area. A TRA should analyze availability, robustness, security, performance, testability, version compatibility, intercomponent compatibility, and functionality aspects of the proposed SOA solution. On the other hand, the following inquiries should be delegated to design trades and risk reduction activities: quality of documentation, ease of use, flexibility, ease of upgrade, portability, price, vendor support and maturity, training, and vendor concessions. Also, the assessment of the COTS SOA framework evaluation effort, glue code writing, and the cost and schedule implications of the integration effort is also out of scope for a TRA, although it is a very critical planning activity.

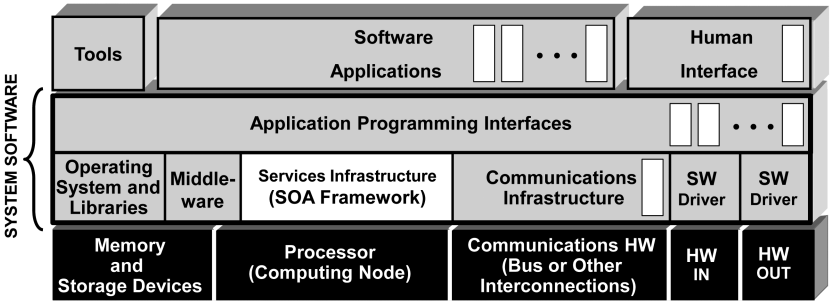

Fig. 5 Potential SOA elements in a ground control system.

Besides the assessment of the SOA framework, a final, cross-domain evaluation is needed. This evaluation needs to be based on the dependency of TRLs described in Sec. III. Consequently, the effective maturity rating of a COTS SOA would be limited by the TRLs of the hardware and software platform and the planned development environment and tools. In the absence of total harmony and compatibility among these entities, one cannot expect a predictable implementation path for the service network model, and the lowest TRL should be used as the actual maturity rating for the COTS SOA in question.

B. Suppression of Enemy Air Defense Unmanned-Aerial-Vehicle Mission Algorithm

Suppression of enemy air defense (SEAD) as a system requirement can be summarized as follows. It is assumed that the details of the terrain, such as location of targets, physical obstacles, and threats in the theatre of operations, are known. There are multiple targets and multiple UAVs launched from different locations. Teams of UAVs are assembled by mission control, and each UAV team is assigned to a particular target. In a typical scenario, to successfully achieve mission objectives, all members of each team have to reach their assigned target simultaneously. The key challenge in this kind of mission is to deal with the timing constraints stemming from the variable speed range of UAV teammates. One of the emerging approaches to manage such cooperative task assignments and path planning for UAVs is the use of genetic algorithms [15]. Genetic algorithms are a class of algorithms that use techniques inspired by evolutionary biology concepts, such as inheritance, mutation, selection, and crossover of chromosomes. In biology, crossover is the process by which two chromosomes pair up and exchange sections of their deoxyribonucleic acid (DNA). In the UAV example, a set of the assigned target's identification (ID) combined with the ID of the selected path that can be used to reach the corresponding target would represent a chromosome. During the execution of a genetic algorithm, the objective is to achieve an appropriate configuration of the mentioned target/path data set, and this objective might be achieved with a series of genetic crossover operations. Genetic algorithms are fundamentally combinatorial optimization tasks and, as part of the solution, authors of [15] proposed the use of the well-known traveling salesman problem (TSP). TSP is a deceivingly simple problem, studied extensively in various domains, such as operations research and theoretical computer science. TSP is described as follows. Given a list of cities and their pairwise distances, find the shortest possible trip that visits each city only once. When it is used as a subsolution in the genetic algorithm context, DNA fragments represent cities and, in the UAV context, they become the equivalents of targets. While it is a simple problem, it is also known among researchers that no fast solution exists, and the time required to solve the problem using any currently known solution increases very quickly as the size of the problem (i.e., number of targets and number of UAVs) grows. However, researchers are also aware of the possible techniques that can be used in lieu of a direct solution, for example, approximation, randomization, or the use of heuristics, to mention a few. All of these methods have inherently different constraints, running times, and success characteristics.

The case study demonstrates that a series of different scientific principles and tools need to be invoked to find a solution to key system functionality. Genetic algorithms and the use of TSP are not software algorithms, even though, during their evaluation, we are successively more concerned about execution constraints. However, when implementation decisions need to be made, computer science and software engineering are unequivocally the necessary disciplines for algorithm selection and evaluation. For example, computers with a large number of general-purpose registers might provide an effective solution for heuristic approaches, even though a common characteristic of heuristics is that there is usually no proof that they would both work fast and produce an acceptable result. In summary, we need to go through at least four layered generations of algorithms to solve the problem: SEAD, genetic algorithms, TSP, and (finally) the algorithmic details of implementation. The decisions on all levels require different kinds of domain knowledge; however, to achieve an optimal solution, we might have to vertically iterate between these layers of algorithms, and such iteration requires an overarching understanding of all affected domains.

C. Wide-Area Search-and-Destroy Unmanned-Aerial-Vehicle Mission Algorithm

This system function involves transmission of detected target information to a command and control authority and reception of engagement authorization for a previously identified target. The scenario (i.e., the closed-loop interaction between a control center and the UAV's onboard weapon system) is not drastically different from the classic challenge army field artillery systems face, and in such systems, rate monotonic analysis (RMA) is often used. RMA is a method for scheduling periodic tasks: tasks that have a periodic request rate and a hard deadline immediately preceding the next periodic request. The fundamental idea of RMA is assigning priorities to tasks according to the period with which they occur; hence, tasks with a shorter period (and consequently with a higher request rate) receive a higher priority. RMA is an abstract algorithm. It is both application and implementation independent, and it is widely used in many different areas. RMA seems to be naturally slated for the UAV function. During a peak-time fire mission scenario, a large number of rapid real-time messages and signals are to be processed, including target intelligence, status reports from various sensors, etc. RMA is ultimately implemented in software, but from a TRA perspective, would it be classified as a user domain-specific or a software algorithm? Based on the criteria set earlier, in this context, RMA is a user domain-specific algorithm. It should be evaluated by military specialists and not software engineers, and a relevant environment validation would have to include actual field data, even before software implementation details are considered.

What makes this case study interesting and unique is that the same algorithm might be used in more than one domain, to be evaluated successively by different domain experts. After the feasibility and viability of RMA for the specific UAV mission is established, the

next step is to review software implementation concerns. For example, the operating system choice would have a major impact, due to the varying levels of real-time support availability in different operating systems. On the other hand, for a UAV onboard solution, a rudimentary approach might also be considered, where an RMA-based scheduler is directly implemented instead of using a COTS operating system that might require complicated licensing and has a much bigger footprint. Similarly, the choice of the programming language is very critical, since coding restrictions and limitations might determine the software engineering skills and effort that is needed to implement RMA. For high assurance software development, Ada is the programming language of choice. However, without going into technical detail, the use of multiple accent statements, selective wait with delay alternatives, pragma shared statements, exception handlers, and rendezvous statements in the Ada programming language all pose serious challenges and might violate one or more fundamental principles of RMA. All these considerations are definitely in scope for a software inquiry.

D. Guidance, Navigation, and Control for Unmanned Aerial Vehicles

Filters in signal processing are mathematical algorithms to remove part(s) of a signal. In most cases, the goal is to remove interfering noise to facilitate easier processing of the objective signal. In the UAV context, the objective signal might be video, infrared, communication, inertial navigation, etc., depending on mission details. The so-called Kalman filter was introduced by Kalman in 1960 [16] and has been widely used ever since for a large class of problems. A typical UAV application is in guidance, navigation, and control, where the continuous real-time solution of the state estimation problem is required for the flying aircraft. On the basis of its track record, it does not seem that the Kalman filter choice would represent a CTE. However, as it will be shown, it would be a grave mistake to let a developer proceed on the basis of a claim that the Kalman filtering algorithm would be used, and its implementation is straightforward and problem free. All expected risks are programmatic in nature and, as such, standard continuous risk management should be able to handle them. Figure 6 shows the emergence of implementation and associated cross-domain concerns during the evaluation of filter implementation choices.

The Kalman filter itself is a mission-specific or user domain-specific algorithm. It could be simulated in advance, but its feasibility would have to be validated using real-flight data. The next step would be to decide on implementation. Again, different analog and digital electronics realizations are available. While it is not used in UAVs, we mention as an interesting piece of information that, in submarines, acousto-optic implementations are very popular. Let us assume that the decision is to use digital hardware. In this case, the final implementation choices are application-specific integrated circuit (ASIC) or field programmable gate arrays (FPGAs), although the flexibility of an FPGA can facilitate hardware–software hybrid solutions as well. We can go down the other path, too, and explore pure software implementations. The algorithm of choice could be programmed from scratch, yielding new code that could not be the object of an advance technology assessment, since it is not even designed yet. However, let us consider a different source. NASA offers a package called a generic Kalman filter (GKF) [17]. GKF is software written in a relatively low-level widely used, standardized programming language: American National Standards Institute (ANSI) C. It consists of a generic Kalman-filter-development directory that, in turn, contains generically designed/coded implementations of a wide variety of Kalman filtering algorithms, with prototype and template versions.

Once the user has provided its own calling routine, the application-specific Kalman filter software can be compiled and executed immediately. The first question we need to answer is whether GKF is GOTS software? Yes. It will have key GOTS characteristics, since its source, NASA, is a government entity. However, from a TRA perspective, the primary classification should be reused code, and all the cautions that are well known for reuse need to be exercised. The use of GKS might, indeed, shave off some of the development time, assuming that the programmers can easily understand how the NASA program works. Unfortunately, there are no guarantees, even with government-originated software, that it is clearly written and well documented. In fact, most of the time, such software components come without formal support, making the understanding, use, and maintenance more difficult. Consequently, rather than delaying the exploration and resolution of problems to the programmatic phase, the prudent approach is a thorough qualification and performance assessment of all provided GKF code via prototyping before use.

Why is this caution particularly warranted for Kalman filters? Even though they are widely used, their accuracy can be inadequate in some applications due to nonlinearities of the physical environment or inaccurate and incomplete modeling of the underlying physical problem. Mission requirements have already raised the bar

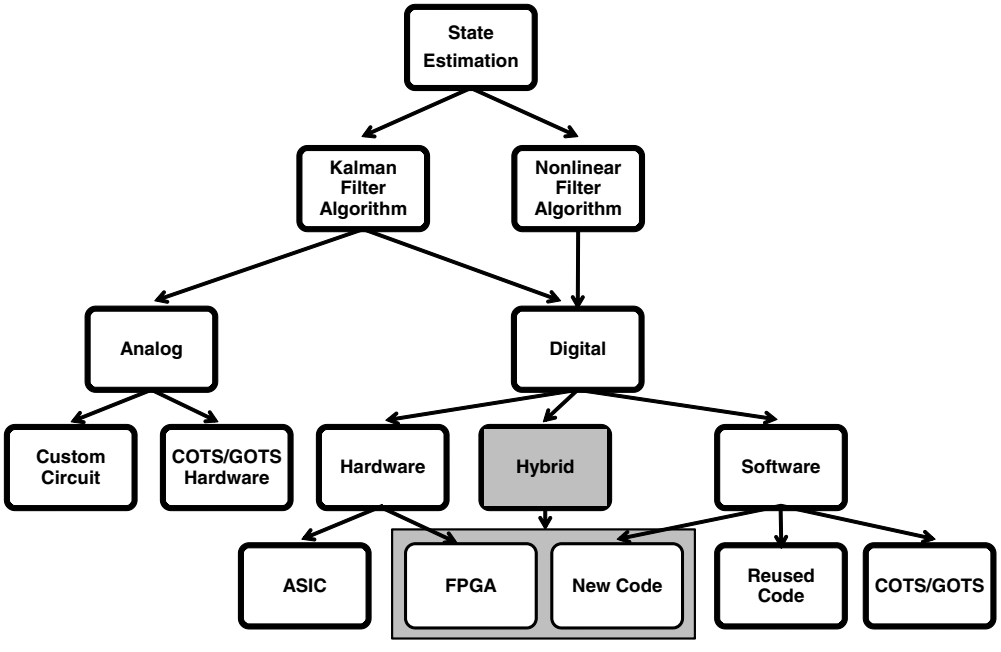

Fig. 6 Filter implementation choices.

and, more recently, extensive research has been initiated to invent nonlinear filters with improved estimation accuracy [18]. In the case of nonlinear filters, the implementation issue we face is real-time computational complexity that is needed to achieve the required accuracy. The process involves cross-domain trades: the implementation of a classic, linear Kalman Filter is easier, but the achieved accuracy might not be adequate. Hence, we might have to iterate between solutions before a production-intent development, either hardware or software, could be started.

VI. Conclusions

Technology readiness is a seemingly simple and straightforward but, in reality, it is an ambiguous and controversial concept representing an emerging system property during acquisition and development. This paper exposed the problem that TRAs for software-intensive systems were traditionally executed in isolated silos of the contributing technical disciplines. The specific domain knowledge that is needed to evaluate these technologies was clarified and structured. How systems engineering can resolve this isolation and provide the needed synergy across user and implementation domains was explained. Essential definitions were systematically introduced for all the fundamental building blocks of the process, and extensive examples were presented to clarify the terms technology, software technology, software technology readiness, and algorithm. It was also demonstrated how the ISO/IEC architecture standard could be used to explore system-wide technology issues in a software-intensive system and seek out and rate CTEs, hardware and software alike.

A final message of this paper is that, without changing the fundamental definitions and policies controlling TRAs, only a diligent and competent systems engineering approach can mitigate the inherent ambiguities and outright shortfalls of the method, in military and nonmilitary acquisition domains alike.

Appendix: Work Breakdown Structure Examples

Table A1 UAV system WBS

Level	Description
1	UAV system
2	Air vehicle
3	Airframe
3	Propulsion
3	Communications/identification
3	Navigation/guidance
3	Central computer
3	Auxiliary equipment
3	Air vehicle application software
3	Air vehicle system software
3	Integration, assembly, test, and checkout
2	Payload (1, . . . , n)
3	Survivability
3	Reconnaissance
3	Electronic warfare
3	Armament
3	Weapons delivery
3	Payload application software
3	Payload system software
3	Integration, assembly, test, and checkout
2	Ground segment
3	Ground control systems
3	Command and control subsystem
3	Launch and recovery equipment
3	Transport vehicles
3	Ground segment application software
3	Ground segment system software
3	Integration, assembly, test, and checkout
2	System integration, assembly, test, and checkout
2	Systems engineering/program management
2	System test and evaluation
	. . .
2	Training
	. . .
2	Data
	. . .
2	Peculiar support equipment
	. . .
2	Common support equipment
	. . .
2	Operation/site activation
	. . .
2	Industrial facilities
	. . .
2	Initial spares and repair parts

Table A2 Space system WBS

Level	Description
1	Space system
2	Systems engineering, integration, and test/program management and common elements
2	Space vehicle (1, . . . , n as required)
3	Systems engineering, integration, and test/program management and common elements
3	Spacecraft bus
4	Systems engineering, integration, and test/program management and common elements
4	Structures and mechanisms subsystem
4	Thermal control subsystem
4	Electrical power subsystem
4	Attitude control subsystem
4	Propulsion subsystem
4	Navigation subsystem
4	Spacecraft bus control
5	Spacecraft bus processor hardware
5	Spacecraft flight software
6	Spacecraft flight software build 1, . . . , k
3	Communication
4	Systems engineering, integration, and test/program management and common elements
4	Communication hardware (1, . . . , n as required)
4	Communication flight software (1, . . . , n as required)
5	Communication flight software build 1, . . . , k
3	Payload
4	Systems engineering, integration, and test/program management and common elements
4	Payload hardware (1, . . . , n as required)
4	Payload flight software (1, . . . , n as required)
5	Payload flight software build 1, . . . , k
3	Booster adapter
3	Space vehicle storage
3	Launch system integration
3	Launch operations and mission support
2	Ground (1, . . . , n) as required
3	Systems engineering, integration, and test/program management and common elements
3	Ground terminal subsystems
3	Command and control subsystem
3	Mission management subsystem
3	Data archive/storage subsystem
3	Mission data processing subsystem
3	Mission infrastructure subsystem
3	Collection management subsystem
2	Launch vehicle
2	SOS systems engineering, integration, and test/program management support
2	SOS test assets

Table A3 SOS generic WBS

Level	Description
0	SOS
1	Space system
2	Systems engineering, integration, and test/program management and common elements
2	Space vehicle (1, . . . , *n* as required)
3	Systems engineering, integration, and test/program management and common elements
3	Spacecraft bus
	. . .
3	Communication
	. . .
3	Payload
	. . .
3	Booster adapter
3	Space vehicle storage
3	Launch system integration
3	Launch operations and mission support
2	Ground (1, . . . , *n*) as required
3	Systems engineering. Integration, and test/program management and common elements
3	Ground terminal subsystems
3	Command and control subsystem
3	Mission management subsystem
3	Data archive/storage subsystem
3	Mission data processing subsystem
3	Mission infrastructure subsystem
3	Collection management subsystem
2	Launch vehicle
2	SOS systems engineering, integration, and test/program management support
2	SOS test assets
1	UAV system
2	Air vehicle
	. . .
2	Payload (1, . . . , *n*)
	. . .
2	Ground segment
	. . .
2	UAV system integration, assembly, test, and checkout
2	Systems engineering/program management
2	System test and evaluation
1	SOS engineering/program management
1	SOS test and evaluation
2	Development test and evaluation
2	Operational test and evaluation
2	Mockups/system integration labs
2	Test and evaluation support test facilities
	. . .
1	Aircraft system
	. . .
1	Electronic/automated software system
	. . .
1	Initial spares/repair parts

Acknowledgments

This work would not have been possible without the funding source, The Aerospace Corporation's Independent Research and Development Program, Software Acquisition Task. Use of any trademarks in this material is not intended in any way to infringe on the rights of the trademark holder. All trademarks, service marks, and trade names are the property of their respective owners. Personal Software Process and PSP are service marks of Carnegie Mellon University. CMMI is registered in the U. S. Patent and Trademark Office by Carnegie Mellon University.

References

[1] "Defense Acquisition Performance Assessment Report," U. S. Dept. of Defense, Jan. 2006.

[2] Erwin, S. I., "Washington Pulse," *National Defense*, Vol. 93, No. 664, March 2009, p. 8.

[3] Ziliak, S. T., and McCloskey, D. N., *The Cult of Statistical Significance*, Univ. of Michigan Press, Ann Arbor, MI, 2008.

[4] Forsberg, K., Mooz, H., and Cotterman, H., *Visualizing Project Management*, 2nd ed., Wiley, New York, 2000, Chap. 3.

[5] Holmes, M., "Competing on Delivery," *Manufacturing Breakthrough*, Vol. 2, No. 1, Jan./Feb. 1993, pp. 29–34.

[6] Mankins, J. C., "Technology Readiness Levels: A White Paper," NASA, 6 April 1995.

[7] "Technology Readiness Assessment (TRA) Deskbook," U. S. Department of Defense, July 2009.

[8] Redwine, S., and Riddle, W., "Software Technology Maturation," *Proceedings of the 8th International Conference on Software Engineering*, May 1985, IEEE Computer Society Press, Los Alamitos, CA, pp. 189–200.

[9] *Work Breakdown Structures for Defense Materiel Items*, MIL-HDBK-881A, U. S. Department of Defense, 30 July 2005.

[10] Foreman, J., Gross, J., Rosenstein, R., Fisher, D., and Brune, K., "C4 Software Technology Reference Guide: A Prototype," Carnegie Mellon Univ. Software Engineering Inst., CMU/SEI0-97-HB-001, Pittsburgh, Jan. 1997, http://www.sei.cmu.edu/reports/97hb001.pdf [retrieved 20 Sept. 2010].

[11] "Systems and Software Engineering: Recommended Practice for Architectural Description of Software-Intensive Systems," International Organization for Standardization/International Electrotechnical Comm. ISO/IEC 42010, Geneva, 2007.

[12] Emmeche, C., Køppe. S., and Stjernfelt, F., "Explaining Emergence: Towards an Ontology of Levels," *Journal for General Philosophy of Science*, Vol. 28, No. 1, 1997, pp. 83–119. doi:10.1023/A:1008216127933

[13] Sauser, B., Ramirez-Marquez, J., Verma, D., and Gove, R. "From TRL to SRL: The Concept of Systems Readiness Levels," Conference on Systems Engineering Research, IEEE Publ. Paper 126, Piscataway, NJ, April 2006.

[14] Zennaro, M., Ko, J., Sengupta, R., and Tripakis, S., "A Service Network Architecture for a Multi-Vehicle Search Mission," *Proceedings of the 40th IEEE Conference on Decision and Control*, Vol. 2, IEEE Publ., Piscataway, NJ, Dec. 2001, pp. 1503–1508.

[15] Eun, Y., and Bang, H., "Cooperative Task Assignment/Path Planning of Multiple Unmanned Aerial Vehicles Using Genetic Algorithms," *Journal of Aircraft*, Vol. 46, No. 1, Jan.–Feb. 2009, pp. 338–343. doi:10.2514/1.38510

[16] Kalman, R. E., "A New Approach to Linear Filtering, Prediction Problems," *Transactions of the ASME: Journal of Basic Engineering, Series D*, Vol. 82, 1960, pp. 35–45.

[17] Lisano, M. E., II, and Crues, E. Z., "Generic Kalman Filter Software," *NASA Tech Briefs* [online journal], 1 Sept. 2005, http://www.techbriefs.com/component/content/article/426 [retrieved 20 Aug. 2010].

[18] Daum, F., "Nonlinear Filters: Beyond the Kalman Filter," *IEEE Aerospace and Electronic Systems Magazine*, Vol. 20, No. 8, Aug. 2005, pp. 57–69. doi:10.1109/MAES.2005.1499276