Applying Technical Readiness Levels to Software: New Thoughts and Examples

James R. Armstrong
Stevens Institute of Technology
Castle Point on Hudson
Hoboken, NJ 02070

Abstract. The concept of Technical Readiness Levels has been applied to software. However, the criteria for levels are more related to design maturity of a specific product than to software technologies. This paper reviews the current work and addresses areas that are more technology related including examples.

Technical Readiness Levels in Programs

Introduction. Programs that are pushing the technical envelope can run into serious problems when they are relying on critical technologies that have not been proven and do not mature to a useable state in time. This has been a serious problem with large government programs that are aiming at the maximum performance advantage. The concept of Technical Readiness Levels (TRLs), introduced by NASA, has evolved as one approach to managing the technical maturity risk. The US Department of Defense positions TRLs in this role in response to various GAO reports in the Technology Readiness Assessment (TRA) Deskbook (DoD, 2005). The combination of TRLs at the macro level with Technical Parameter Measurements (TPMs) at the micro level as been proposed as a combined means for mapping and monitoring the Technical Critical Path in a development program (Armstrong, 2009).

The issue with this approach is that the original concept of TRLs was focused on, or at least phrased in terms of, hardware. As large, complex systems are becoming more dependent on software for critical performance functionality, the application of TRLs to software gains importance. Both NASA and DoD have developed descriptions for software TRLs. However, both versions tend to be more descriptive of specific product maturity than the general development of a technology. The following sections will address the existing definitions of hardware and software TRLs and propose extended definition of application to software technology as opposed to software products.

Basic Technical Readiness Levels. NASA introduced the concept of TRLs as a means to track the progress of a new technology and determine how far it was along the path from just a novel idea to being ready for prime time. The nine levels are shown in figure 1 along with a short description.

The lower readiness levels address technologies that are typically being investigated in laboratory experiments. An example of a technology in this region would be single electron logic gates that are based on electron spin and quantum mechanics. While the actual electron gate is obviously small, the apparatus used to operate a single gate is a sizable set of laboratory equipment. Do not expect this technology to be appearing in stores soon.

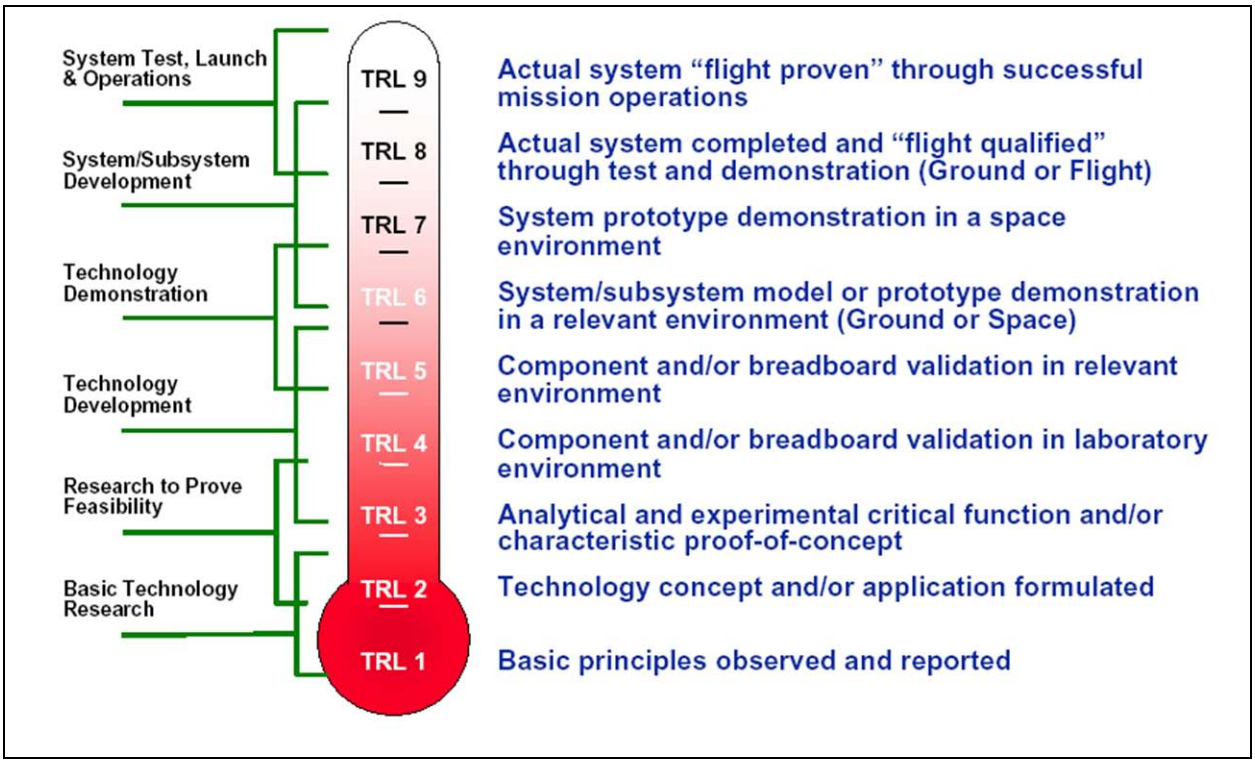
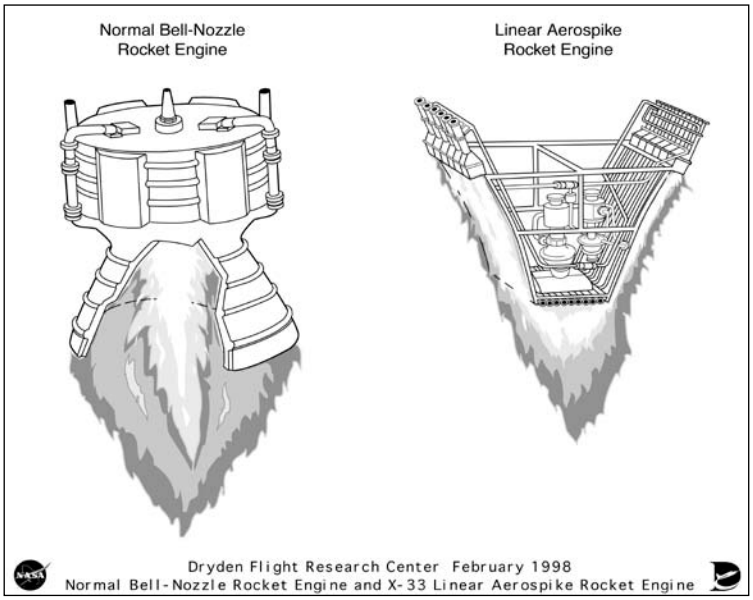


Figure 1, NASA Technolgy Readiness Levels



In the mid range, an example might be the aerospike engine. The concept is depicted in Figure 2. The normal bell shaped engine is only optimal at one altitude where the shape of the bell matches the flow characteristics for the atmospheric pressure at that point in the trajectory. In multistage rockets, each stage has a different shape optimized for its range. When designing for a single stage flight to orbit, this creates a problem. One solution is to effectively reverse the design and place the bell on the inside letting the outside vary with the flow of the atmosphere as pressure changes.

The linear form, as shown in the figure, was part of the X-33 program. Other engines using a circular form have been applied in smaller rocket projects. The design has been around for many years and much is known about it. However, it is not to the point that it can be selected for the next moon launch design.

The highest levels require that the technology be tested, demonstrated, or actually used in simulated or real environments. Technology that is currently in use in the shuttle or space station certainly qualifies as would a new technology that undergoes thermal and vacuum testing to prove

that it's ready for launch. Simulated environments uncover such problems as the satellite that would operate in space, but would not have been warm enough to turn on had it been launched without space environment testing.

Software TRLs. Although not specifically excluding software, most of the emphasis in the initial TRL work was on hardware technologies similar to the examples given. The question has been asked, "How do these concepts apply to software?" Several efforts have addressed this question. However, most of the factors are more design maturity issues than to technology readiness. In fact, the appropriate application of the term technology to software is not really addressed.

One of the most referenced sources on this topic is an SEI report TRLs for non-developmental software (Smith, 2005). It describes several concerns that should be addressed in selecting off-the-shelf software products from commercial, government, or internal sources including open source. The attributes that are proposed to determine a TRL are requirements satisfaction, environmental fidelity, product criticality, product aging - availability and product aging – maturity. Although these are important factors to consider, they are descriptions of a specific product and its design status. Generally speaking, a software technology must have reached a reasonably high level of maturity for it to already be in use in an off-the-shelf product.

NASA has provided its own extension of the TRL definitions for software (NASA 2008) as shown in Table 1. I should be noted that the lower levels address architecture, mathematical formulation, and algorithms. The language quickly changes to coded principles, experiments with data, functionality, component integration, and removal of bugs. While the general flow from lab to actual operational environment remains, the focus is clearly on design of a specific product without a clear idea of technology. This view is certainly valid and important in developing the software that will be used in a specific program; however, it still lacks the taste of technology that can be sensed in the hardware world.

TRL	Description
1	Scientific knowledge generated underpinning basic properties of software architecture and mathematical formulation.
2	Practical application is identified but is speculative, no experimental proof or detailed analysis is available to support the conjecture. Basic properties of algorithms, representations and concepts defined. Basic principles coded. Experiments performed with synthetic data.
3	Development of limited functionality to validate critical properties and predictions using non-integrated software components.
4	Key, functionally critical, software components are integrated, and functionally validated, to establish interoperability and begin architecture development. Relevant Environments defined and performance in this environment predicted.
5	End-to-end software elements implemented and interfaced with existing systems/simulations conforming to target environment. End-to-end software system, tested in relevant environment, meeting predicted performance. Operational environment performance predicted. Prototype implementations developed.
6	Prototype implementations of the software demonstrated on full-scale realistic problems. Partially integrate with existing hardware/software systems. Limited documentation available. Engineering feasibility fully demonstrated.

7	Prototype software exists having all key functionality available for demonstration and test. Well integrated with operational hardware/software systems demonstrating operational feasibility. Most software bugs removed. Limited documentation available.
8	All software has been thoroughly debugged and fully integrated with all operational hardware and software systems. All user documentation, training documentation, and maintenance documentation completed. All functionality successfully demonstrated in simulated operational scenarios. Verification and Validation (V&V) completed.
9	All software has been thoroughly debugged and fully integrated with all operational hardware/software systems. All documentation has been completed. Sustaining software engineering support is in place. System has been successfully operated in the operational environment.

Table 1. NASA Software TRL Descriptions

The US Department of Defense has developed a definition of TRLs for software as shown in table 2. As can be seen, it follows the NASA descriptions rather closely. In doing so, it has the same bias towards the product maturity of a specific software product.

TRL	Description	Supporting Information
1 Basic principles observed and reported.	Lowest level of software technology readiness. A new software domain is being investigated by the basic research community. This level extends to the development of basic use, basic properties of software architecture, mathematical formulations, and general algorithms.	Basic research activities, research articles, peer-reviewed white papers, point papers, early lab model of basic concept may be useful for substantiating the TRL level.
2 Technology concept and/or application formulated.	Once basic principles are observed, practical applications can be invented. Applications are speculative, and there may be no proof or detailed analysis to support the assumptions. Examples are limited to analytic studies using synthetic data.	Applied research activities, analytic studies, small code units, and papers comparing competing technologies.
3 Analytical and experimental critical function and/or characteristic proof of concept.	Active R&D is initiated. The level at which scientific feasibility is demonstrated through analytical and laboratory studies. This level extends to the development of limited functionality environments to validate critical properties and analytical predictions using nonintegrated software components and partially representative data.	Algorithms run on a surrogate processor in a laboratory environment, instrumented components operating in laboratory environment, laboratory results showing validation of critical properties.
4 Module and/or subsystem validation in a laboratory environment (i.e., software prototype development environment).	Basic software components are integrated to establish that they will work together. They are relatively primitive with regard to efficiency and robustness compared with the eventual system. Architecture development initiated to include interoperability, reliability, maintainability, extensibility, scalability, and security issues. Emulation with current/ legacy elements as appropriate. Prototypes developed to demonstrate different aspects of eventual system.	Advanced technology development, stand-alone prototype solving a synthetic full-scale problem, or standalone prototype processing fully representative data sets.
5 Module and/or subsystem validation in a relevant environment.	Level at which software technology is ready to start integration with existing systems. The prototype implementations conform to target environment/interfaces. Experiments with realistic problems. Simulated interfaces to	System architecture diagram around technology element with critical performance requirements defined. Processor selection analysis, Simulation/Stimulation (Sim/Stim) Laboratory buildup plan. Software placed

	existing systems. System software architecture established. Algorithms run on a processor(s) with characteristics expected in the operational environment.	under configuration management. COTS/GOTS in the system software architecture are identified.
6 Module and/or subsystem validation in a relevant end-to-end environment.	Level at which the engineering feasibility of a software technology is demonstrated. This level extends to laboratory prototype implementations on full-scale realistic problems in which the software technology is partially integrated with existing hardware/software systems.	Results from laboratory testing of a prototype package that is near the desired configuration in terms of performance, including physical, logical, data, and security interfaces. Comparisons between tested environment and operational environment analytically understood. Analysis and test measurements quantifying contribution to system-wide requirements such as throughput, scalability, and reliability. Analysis of human-computer (user environment) begun.
7 System prototype demonstration in an operational high-fidelity environment.	Level at which the program feasibility of a software technology is demonstrated. This level extends to operational environment prototype implementations where critical technical risk functionality is available for demonstration and a test in which the software technology is well integrated with operational hardware/software systems.	Critical technological properties are measured against requirements in a simulated operational environment.
8 Actual system completed and mission qualified through test and demonstration in an operational environment.	Level at which a software technology is fully integrated with operational hardware and software systems. Software development documentation is complete. All functionality tested in simulated and operational scenarios.	Published documentation and product technology refresh build schedule. Software resource reserve measured and tracked.
9 Actual system proven through successful mission-proven operational capabilities.	Level at which a software technology is readily repeatable and reusable. The software based on the technology is fully integrated with operational hardware/software systems. All software documentation verified. Successful operational experience. Sustaining software engineering support in place. Actual system.	Production configuration management reports. Technology integrated into a reuse "wizard"; out-year funding established for support activity.

Table 2. DoD Software TRL Descriptions

Software Technologies

Technology versus Product. The development of a single product may advance a technology, or at least address specific issues of its application in a particular application. Hardware technologies are things most people can relate to. Cell phone systems, flat screen displays, electric automobiles, touch screens, and microwave heating are understood to be technologies as opposed to the specific products that use them. There are several analogous technologies and categories of technologies for software. By identifying them as such, the application of TRLs to software-intensive programs can be more effective.

Algorithms. The NASA and DoD descriptions to include algorithms in their discussion of TRLs. However, their definition and development are only referred to in levels 1 and 2. Later mention in the DoD version is limited in context to the processor they run on. A broader context would be to define the type of algorithm and its maturity in regard to a class of products in which it can be applied. An example would be the algorithms that are used to detect missile launches from space. When the Defense Support Program was in its early stages, there was considerable concern as to

whether or not the software could detect a missile launch, given the IR data that the sensors would provide. There was no prior history to rely on and the maturity would have to be scored as a low TRL. A low orbit prototype gave some credence to the technology and moved it to a mid range value. However, that data was not from the orbit and with the sensors that would be in the actual system. Problems were solved and the system worked. Today, there is not much question that the algorithm can perform in another application. It may slip back a few levels if new sensors or systems architectures are used but not back to the lower levels as a general technology.

In the civilian world, aircraft collision avoidance algorithms have similarly matured. The testing of decades ago to see if such a concept was at all feasible have passed and we are now in the stage of application to products. Improvements in the algorithms are certainly being developed but the overall technology is mature.

Commercially, the technology of the search engine has become very mature. We rely on it in various applications on a regular basis. New products continue to arrive on the market and old ones are upgraded.

The upper levels of TRL which address specific environments continue to be applicable. For instance, if applying a commercial product, or reuse a component from a prior application, we have to ask whether the product and its technology have been applied in this specific application. One example would be the use of commercial search engines or other software technology in a security or safety driven application. Another would be the reuse within a similar but different environment as was the case with the Arian V.

Languages. One of the first questions asked of an applicant for a programmer position concerns languages. As new languages are developed for various reasons, there is a normal tendency to jump on the bandwagon and claim that all the old problems have been solved. Premature use of a new language can be very problematic for several reasons. First, the developers have not learned its strengths and weaknesses, or its traps. Many times programmers will continue to use the methods they learned with prior languages and undermine the benefits of the new language. The support environment may not be fully developed and compilers, debuggers, and other tools are not in place or fully developed. These problems were certainly seen in the introduction and use of Ada and, in addition to being “that DoD mandated language”, helped limit its popularity and effective use.

Architectures. As new approaches to the overall architecture of software systems are developed, they need to be looked at from a TRL point of view. We now are being deluged with discussion of Service Oriented Architectures as being the ultimate solution to everything. In a related vein, cloud computing is also becoming an in vogue buzzword. However, we need only look back to how past architectural technologies were over applied in inappropriate situations to see where a new technology needs to be more carefully thought through before immediate use. In the early days of local network, one organization decided to interconnect all of the computers using a central server-based software architecture. Unfortunately, the computer selected had very limited input/output capability and the mass memory was tape technology. When using the word processing function, keyboard inputs regularly were delayed by a few seconds as they waited in the 2800 baud queue. More importantly, bringing up an existing document to work on it involved calling the computer center to find the correct tape. The users revolted and the system was soon abandoned. Other, more successful applications of this centralized processing architecture worked well for those in the building but performed miserably for a significant part of the workforce that

were working remote either permanently or on travel status.

Design methods. The initial software functional design technology was very effective, and remains so, for algorithmic calculation. As more database oriented applications came about, the approach of Object Oriented Analysis (OOA) and Object Oriented Design (OOD) were developed. For the purpose it was invented, it was very effective. It did take a while for the best practices to be developed and communicated. Now, it is considered a common approach to use OO and there are large numbers of options for detailed methodologies and tools to support it. Of course, it became the favored approach for everything and people forgot that, like many technologies, it is not necessarily the best choice for everything. Customers have been quoted as saying that they want only OO and don't want to hear the word 'function' at all. Even after OO was a mature technology for database oriented programs, developers found that it was not the most appropriate and mature technology for algorithmic software such as radar signal processing.

Protocols. As different applications, particularly communications methods, are developed, some of their basic operating rules are defined in protocols. These can take significant time to work out to assure that they properly handle the content intended in the expected environment. As the internet has evolved into a significant part of individual and organizational life, the maturity of the internet protocols has been a significant factor in its success. New applications of use of the web have brought about versions such as secure HTTP and any new use should go through the higher TRL questions. However, the HTTP technology has to be considered as relatively mature.

Agents. A favorite example of software technology maturity is the use of agents. It would be interesting to propose that the FAA rely on agent based software to negotiate among aircraft and between aircraft and ground facilities for flight path and terminal operations coordination.

Genetic programming. In a similar vein, the artificial intelligence community has used the concepts of genetic programming for some time in addressing complex problems for which the best approach is not clear. In this method, the software self selects and modifies some of the program content as it works towards finding the best fit. While this approach might be acceptable in an application such as trying to find the best predictor of hurricane paths, it is probably not going to be seen as mature enough to be part of a safety critical program.

Software support and testing. Support is general area of software related technologies that have their own TRL issues. One such area is the field of automated testing and test case generation. This technology has had several advances over the last decade and has proven valuable in many situations. Yet it still has areas of application where work has not been completed to the point where it can be considered mature for that application. On the other hand, there are many who tried it in the earlier, less mature stages and will continue to consider it as immature regardless of what has happened in this technology since then.

Summary

TRLs have significant relevance in defining and monitoring the technical risk of a program. This is particularly true of large, complex systems that are pushing the technology state-of-the-art to achieve significantly advanced performance. As these programs become more software-intensive, they need appropriate guidance in the application of TRLs to software. While the current guidance does give such guidance with regard to the development of specific products for the system, additional guidance on how to view software from a technology standpoint will be of benefit. As described in this paper, some of the things that should be considered are algorithms, languages,

architectures, design methods, protocols, specific design approaches such as agents or genetic programming, and software support elements such as automated testing.

Each of these can be considered for their application, not just in the program in question, but in all software development in general. As new ideas come into being, they should be evaluated against the TRL criteria to establish their maturity before grabbing them as the next best thing for immediate application. Even those that have been around for years with successful application in other environments should be carefully stepped through the higher TRLs to identify the problem areas of the new application. This broader interpretation of software TRLs may be a significant help in avoiding the problems associated with immediately selecting the latest software silver bullet idea for immediate application.

References

Smith, James D. II, *An Alternative to Technology Readiness Levels for Non-Developmental Item (NDI) Software*, Carnegie Mellon Software Engineering Institute, 2005

NASA Research and Technology Program and Project Management Requirements, NASA Procedural Requirements 7120.8, Appendix J. Technology Readiness Levels (TRLs), February 05, 2008.

Department of Defense, *Technology Readiness Assessment (TRA) Deskbook*, Deputy Under Secretary of Defense for Science and Technology (DUSD(S&T)), May 2005.

(Additional references will be added.)

BIOGRAPHY

Jim Armstrong has practiced systems engineering for over 40 years, performing various roles including configuration management, test, deployment, chief engineer, program manager, and program element monitor. For the last 20 years, he taught, consulted, and appraised systems engineering in industry and government. Also, he was on the author teams for several of the systems engineering standards and models