2017-03

# Transformation of Test and Evaluation: The Natural Consequences of Model-Based Engineering and Modular Open Systems Architecture

Guertin, Nickolas H.; Hunt, Gordon

Monterey, California. Naval Postgraduate School

SYM-AM-17-081

# Proceedings
## of the
# Fourteenth Annual Acquisition Research Symposium

## Thursday Sessions
## Volume II

**Acquisition Research:**
**Creating Synergy for Informed Change**

**April 26–27, 2017**

**Published March 31, 2017**



Prepared for the Naval Postgraduate School, Monterey, CA 93943.

Acquisition Research Program
Graduate School of Business & Public Policy
Naval Postgraduate School

# Transformation of Test and Evaluation: The Natural Consequences of Model-Based Engineering and Modular Open Systems Architecture

**Nickolas H. Guertin—**PE, is a Senior Systems Engineer at Systems Planning and Analysis, Inc. He is a recently retired Navy Civilian, where he held a variety of technical and business leadership positions, including Director for Rapid Prototyping and Strategic Planning at NSWC Carderock and Director for Transformation in the Office of the Deputy Assistant Secretary of the Navy for Research, Development, Test, and Evaluation. He holds a BS in Mechanical Engineering from the University of Washington and an MBA from Bryant University. He is a registered Professional Engineering and is DAWIA certified in Program Management and Engineering. Guertin is also a retired Navy Reserve Engineering Duty Officer.

**CAPT Gordon Hunt, USN—**is a Naval Reserve Engineering Duty Officer and supports PEO IWS and NAVSEA on naval combat system software solutions and design principles. His civilian employment (Skayl) focus is on system of systems integration and semantic data architectures, enabling increased systems flexibility, interoperability, and cyber security. He holds an MS in Aeronautical Engineering from Stanford University and a BS in Aerospace Engineering from Purdue University. He also serves in industry leadership as an appointed member for the Advisory Board of the Open Group's Future Airborne Capability Environment (FACE) architecture consortium.

## Abstract

This paper examines the technologies and architecture patterns that are transforming software-intensive systems and the Internet of Things (IoT) that are currently being designed and implemented. The use of these practices should create an ensuing transformational shift in the relationships between the Test and Evaluation (T&E), development, and operational communities.

Based on the findings of this research, a set of practices for a coordinated set of hardware, software, functional, and data architecture patterns and testing strategies is presented. This paper will show how these need to be applied via a data architecture that defines the declared test points between modular components in software intensive systems. This will support affordable and rapid integration of innovation through a business model that uses small-scale component replacement. This research ends with an assertion that, when the right architectural elements are standardized, regular incremental improvement is both affordable and effectively applied throughout system development.

## Introduction

This paper proposes a new path toward a robust and affordable approach for product development to achieve the fundamental purposes of T&E—to validate and verify the acquisition of excellent military capability. The architecture itself, not just the content, should also be testable to its own set of requirements. As such, there needs to be a set of practices that can directly test such architecture characteristics of flexibility, scalability, interoperability, and so forth, prior to making major investments in detailed development. Then, when the content of the components that make up the system are filled out, the test and evaluation process can validate and verify that the content is following the constraints of the architecture. In this way, when the full system is completed, the program is not at the beginning of traditional T&E, but at the end of the product development and test process, and can quickly transition to fielding. Synergistically, when open and functional architecture steps are followed, some of the smaller testable chucks of software prior to the "full system" being created, thus expanding opportunities for broader enterprise value of strategic reuse.

DoD procurement is changing, driven by a combination of the national strategic imperative to much more rapidly address the needs of the warfighter who is facing an asymmetric enemy, who is also able to access the fundamental underlying building blocks of capability derived from globally available commercial technology (National Defense Authorization Act, 2017). As a result, defense procurement needs to become nimble, deriving new mission capability through flexible and rapid integration of capability modules instead of classically procured standalone systems (Richardson, 2016).

A natural extension of these assertions is that fielding of new capability must also be made more fluidly, frequently, and in smaller increments than the large-scale major systems deployments typical of classic Program of Record approaches that follow a design/build/field/sustain/dispose life cycle as shown in Figure 1. The environment must adjust to a different deployment model for capability where new features and performance capabilities can be delivered when they are needed in the field.

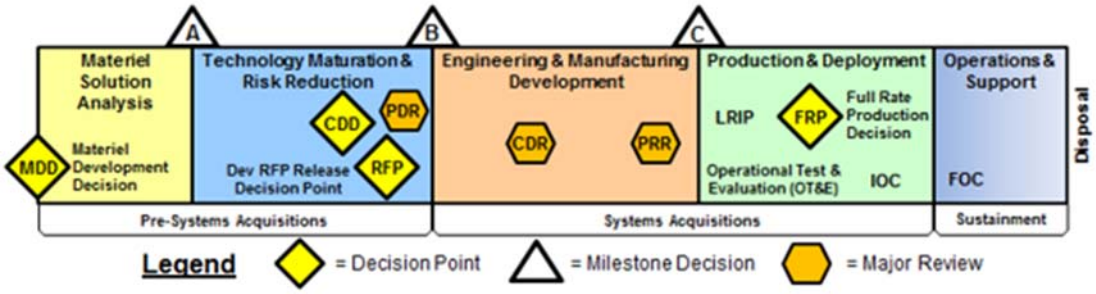


**Figure 1. Defense Acquisition Framework**

The new model changes the design/build/field/sustainment approach of discrete and separate phases into one of continuous engineering and deployment. In order to achieve this, new increments of capability, acquired from a wide range of offerors, must be able to be affordably tested and fielded within days or weeks. Both the testing community and the acquisition environment must have strong evidence and buy-in that such an approach can be risk-prudently performed. This will be a difficult change in culture as both of these communities are steeped in the natural cadence of the defense acquisition framework, which can require years to move from characterizing a problem to having a fully fielded system-specific capability.

## The Changing Environment

Energetics and life-costing decisions hang in the balance of military products. The rigor of testing and certification required for warfighting capability is very different from standard industry practices for consumer products, such as testing an incremental release of a popular mobile application. As such, software for military warfighting systems, regardless of its origin, must be governed and implemented with rigor. This is made more urgent by the growing and persistent cyber threat to software-intensive systems. Consider the following observations on current DoD software architecture practices:

- The System's Engineering "V" diagram is being eclipsed to today's by new forms of robust model-based systems engineering and systems-of-systems design environments and tools (Micouin, 2014). In addition, system performance requirements must adjust as the technology matures and the warfighting problem space changes. Architectures and test capabilities are

needed to readdress the duration of the continuous engineering, testing, and deployment phases.

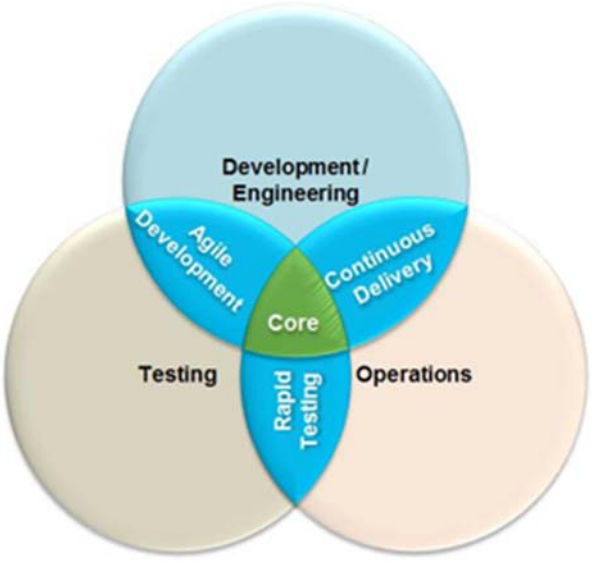


**Figure 2. Continuous Engineering**
(Rahman, 2014).

- Modern cyber-physical software development practices are using decomposition of capabilities into smaller, individually competed functions (Guertin, Schmidt, & Sweeney, 2015). These functions also need to come built with accessibility to internal software "test-points" and conformant external interfaces. Both of these will be necessary to support automation of tests.
- Today's delivered capabilities span multiple systems, programs, and services (Jamshidi, 2008). The overarching integrated capabilities are composed of orchestrated behaviors forced through an array of architectures, deployed on different hardware, using different internal interfaces with a multitude of different data representations.
- Advances in Model Based Engineering (MBE) now enable the acquisition community to explicitly address integration complexity for definitions of system software specifications (DoD, 2017). This action will fundamentally address the one issue that simultaneously decreases system costs and reduces time to deployment.
- The Defense Department Services are revolutionizing software development with researched, tested, and validated Open Architecture approaches (Guertin & VanBenthem, 2016). However, successfully delivering its full value to the warfighter requires the entire DoD procurement cycle to fully embrace its potential and to deliver on its value.
- The T&E community also develops large, complex software intensive systems to support testing (Deputy Assistant Secretary of Defense [DT&E], 2016). Those MBE practices need to be aligned with the associated MBE efforts being used in the acquisition community. The T&E community has the ability to develop a rich set of cohesive testing infrastructure and tools while still preserving their independent role.

Testing assembled systems and systems-of-systems (SoS) has been long, complex, and expensive. As such, the effort associated with fielding adequately tested products is rapidly increasing (Deputy Secretary of Defense for Systems Engineering, n.d.). Automation of testing activities is necessary to address these activities to improve test breadth and penetration, while simultaneously increasing speed of delivery and reducing the overall test burden (Elfriede, Rashka, & Paul, 1999). This will similarly require robust testing frameworks that are not intimately tied to the product being tested. To align the product development and system test domains, the full panoply of complexity must be addressed, such as: internal component functional testing, system integration, and cross-system behavioral testing of software-intensive systems.

This creates an opportunity for alignment of the MBE efforts across the acquisition workforce. The exploding complexity can be managed through a greater emphasis on defining the data artifacts of the modules of the systems under test, while also enabling their extensibility for re-use. This includes tools that can specially address the complexity of testing software intensive systems, a market-place of T&E products and test artifacts, and a third-party marketplace of innovative products to support the T&E workforce.

Design and testing of interfaces are also going through fundamental changes in both approaches and results. The classic approach of an Interface Control Document can be replaced by using a combination of MBE and a supported data model. These together fully define what data moves across the system and how data is used internally to a module to support functional performance. The consistent and testable MBE processes and software architectures can then be used to provide “managed” automation of testing and interoperability. This results in testing products that are open to support integration and are readily reusable across programs.

## Defining a “Testable” Architecture

The T&E community can test components and software early and often by first decomposing the criteria for the fundamental building blocks of software intensive systems. An analogy is beneficial to set the stage—the most accessible and reliable one is your house.

To better understand the relationships of enterprise design, consider the comparative example of how communities build out their towns. Figure 2 depicts the relationship of the enterprise architecture to the community’s master plan. The enterprise architecture is the first tier of a multi-level design process. Both the large-scale plan and the individual house plan represent a forward-looking vision of the eventual community or product-line implementation. Both the developer and the inspector are governed by regulatory practices and architectural patterns and styles, and they must be responsive to future market and business-driven factors. In short, building codes define the architecture rules.

At the highest level in the building architecture analogy, business and community leaders determine what they want their town to look like and what infrastructure requirements they might need. Roads, utilities, and capacities are examples of the highest level requirements of a community. Likewise, an enterprise product line for a defense system constructs a set of rules to facilitate systems domain business requirements, such as portability, reuse, interoperability, speed, and scalability. These requirements are then translated into attributes that the resulting architecture must possess.

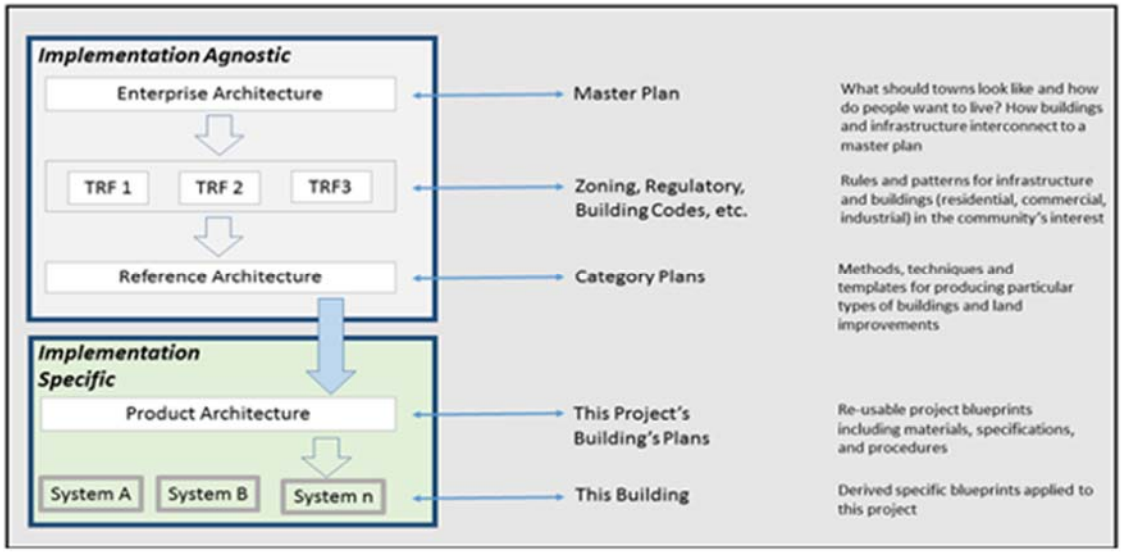


**Figure 3. Architecture via Building Codes Example**

When a house is built, it must conform to these overarching requirements. The rules of construction are set for things like framing, electrical, plumbing, insulation, internal and external finishes, and so forth. Building/zoning codes and other constraints, like homeowners association agreements, can also assert controls on how the building interacts with the rest of the community.

The home builder does not set the building codes or the inspection and test methods. The inspectors have a basis for evaluating creative alternative implementations while preserving safety for the individual and value to the community. These codes and building rules must be structured to be loosely coupled and have limited impact on other rules. However, when new construction methods, modern materials, or new aesthetics are presented, the test criteria and test methods must react dynamically and evolve.

The physical nature of a building forces us to take a step-wise approach to inspection, such as the overall design is inspected before construction begins, the foundation is inspected before the structure is built and before electrical and plumbing is installed, etc. Software intensive systems could be approached in a similar fashion. As such, the developers and evaluators would establish a partnership for setting the building codes and creating the criteria for testing and inspection.

### *Establishing the Categories of Architecture*

The T&E community must be involved in defining the "building codes" for today's complex software-intensive systems to establish the governing principles for building, implementing, and eventually testing software-intensive System-of-Systems (SoS).

The scale of the problem is growing faster than existing methods can account for. Consider that a modern automobile can have 10 million lines of "mission critical" code ("How Many Millions," 2017), and even the operating systems can easily top 25–30 million lines of code. Similarly, modern weapons systems can each have in the neighborhood of 10–20 million lines of code. Additional software is needed to integrate and share data between all platforms, sensors, and weapons to make these complex systems perform together. In addition, the very act of testing these software-intensive systems is creating significant amounts of software as well. As a result of this complexity, one or more of the following

challenges are often observed when testing and integrating large software-intensive systems:

- Integration patterns limit flexibility for incorporating new capabilities.
- System do not scale in size or diversity and have less capability than required.
- External or key interfaces do not work as specified or designed.
- Functionality is reduced from design specifications.
- Interface documentation is insufficient to effectively test system boundaries.
- Traditional Interface Control Document (ICD) specifications evolve too slowly to accommodate evolving or novel capabilities.
- The combinatorial challenge of testing every interface leads to untested interactions (Kuhn, Kacker, & Lei, 2013).

Analysis of these failure mechanisms often indicates that unplanned dynamic behavior exists among the key system elements (Capilla et al., 2014). Since the characteristic of this failure mode is unanticipated and systemic, it is unlikely that traditional analysis and engineering judgment will provide a robust and enduring solution. The situation will deteriorate as systems get more distributed, complex, and interdependent. The commercial industry is working to address the challenge of an estimated 20 billion connected devices by 2020 (Hosain, 2016). This revolution is happening over time to build and connect these devices, including finding agreements on how they all connect together. The solution to this complexity must be foundational and a fundamental aspect of architecture.

In some isolated cases, the idea of a testable architecture has been realized. For example, Architecture Analysis and Design Language (AADL) was created for the specification, analysis, automated integration and code generation of real-time performance-critical distributed computer systems (Architecture Analysis and Design Language, n.d.). AADL provides additional model-based engineering mechanisms to test an architecture prior to full product development. Outcomes of embracing testable architectures include the following:

- Early detection and debugging of unanticipated dynamic behavior
- Exercise key interfaces early in the development phase
- Enable benchmarking of key function prototypes in the system environment to provide visibility into unanticipated dependencies
- Provide for tools that can test the architecture separate from function enabling repeatable testing of early development products in a system environment
- Analysis and comparison of design alternatives in a system environment
- Discovery of emergent net-benefit capabilities that can be realized when integrating large systems and system-of systems.

The following assertions were derived from the experience of the authors and validated through research:

- Integrating systems in predictable and testable increments reduces risk and rework.
- Defining the rules and the architectural elements that enable this behavior further improves outcomes.

- Analysis of the different documents, standards, processes, models, software libraries, and physical components led to a definition for how to implement a system.

These core concepts were then distilled and mapped to a set of architectural elements from which systems can be informed, specified, designed, and implemented (Allport, Hunt, & Revill, 2016).

Table 1 presents a simplified view of that analysis, where the input in the left column is the software, interface, or hardware specifications that the acquisition community currently leverages in execution and implementation of cyber-physical systems. Those were grouped and identified the input's core architectural tenets to realize the reference architecture categories. As additional input, consideration was given to current architectural tools, standards, and best-practices to ensure that architectural content in an identified category could be captured and documented, and most importantly, tested.

**Table 1. Core Architectural Elements**

| Cyber-Physical System Concepts Leveraged in Execution & Implementation | Core Architectural Tenets | Reference Architecture Category |
|---|---|---|
| Interface Control Documents (ICDs)<br>Interoperability Profiles (IOPs)<br>Tactical Data Link Specifications (TDL) | Interfaces<br>Messages | Functional Architecture |
| CPU & Hardware Selections<br>Network & Communication Fabrics<br>Hardware Input/Output (IO) | Deployment | Hardware Architecture |
| ICD Documentation<br>Configuration Files<br>Intrinsic Knowledge of Meaning | Knowledge<br>Information | Data Architecture |
| Infrastructure Software<br>Operating Systems (OS)<br>Middleware Libraries<br>Software Development Kits (SDK)<br>Model-Based Engineering Code Utilities | Applications<br>Infrastructure | Software Architecture |
| Commercial Standards<br>Defacto Standards<br>DoD Specifications & Requirements | Standards | Functional Architecture |
| Acquisition Process<br>Contracting Actions<br>Requirements & Specifications | Business Model | Governance |

The reference architecture categories serve as the building code categories for specifying, designing, and implementing systems and testable elements in the architecture. Each of the identified reference architecture categories are defined as follows:

- Functional/System Architecture—This architectural segment is closely tied to the business goals of the system and includes statements about what a device or service "does," what it "provides," and what it "needs." Testable KPPs are usually defined against the functions and are implicitly coupled to the implementation requirements. Traditional ICDs document and define messages and interface syntax as aspects of an interoperability requirement on a unit of function. These often implicitly couple a function's deployment and current intended use into its specification. Various model-based engineering tools and standards exist for documenting interfaces of a system. The challenge is to ensure that the documentation and design clearly decouples software, from data, from function of an interface specification.
- Software Architecture—This architectural segment focuses more on how a function should be implemented in code and logic. It covers how the software infrastructure, computational support interfaces, operating systems, middleware technologies (Hohpe, 2004), and display technologies are used and integrated. A software architecture defines the boundaries between components of functionality, the granularity of those components, how those components communicate, and how the resulting software is deployed and managed. Key interfaces are identified, and mechanisms to test and decouple the interfaces are often elements of a software architecture. The challenge is that software architectures are not crafted as enduring designs and many times end up as a *defacto* system architecture coupling one system's implementation specifics to every other software service in the system.
- Data Architecture—This architectural segment is focused on documenting the content *and* meaning of data. Data is not just what is exchanged between functions and comprises more than the messages. Levels of interoperability (Tolk & Muguira, 2003) define not only the structure (syntax) of the data but also the context and behavior (semantics) as well. Traditional documentation of data has captured syntax, but semantic and meaning of the data is implicit when considering model-based engineering practices. Recent standardization efforts and activities have clearly delineated the data architectural properties necessary for a reference architecture (The Open Group, n.d.). This includes the ability to identify the syntax (structure of interfaces and messages), the traceable context (semantics of the data itself), and the behavior of the data as presented through the documented interfaces. The challenge in data architecture is decoupling the documentation of the meaning from the presentation and dissemination of messages. Often what is actually exchanged carries additional meaning and understanding that is not overtly stated or captured. Not capturing this meaning in MBE formats make testing at scale certain to generate integration errors and is the source for many of the non-desirable emergent behaviors.
- Hardware Architecture—This architectural segment covers specifications on physical computer hardware, network fabric and I/O signaling mechanisms, hardware mounting, power and handshaking protocols, connectors' wiring specification, and the like. The T&E community has a long history of

successfully and independently testing hardware. The challenge addressed in architecture is to test and ensure that software, functions, and data are sufficiently decoupled. While it is sometimes advantageous to directly talk to hardware for performance, the T&E community needs testable architectural mechanisms to isolate and decouple software and function from the hardware.

- Governance—This is a critical component in a reference architecture and details where and how the various levels of the architecture will be assembled, deployed, evaluated, and tested for conformance. The architectural categories that need to be put together for testing are as important as the testing architectural products of the individual categories.

The relationship between these architectural segments provides additional testable architectural attributes. When coupled with a functional architecture, the data architecture ensures the information flowing across to peer-level modules will be correctly interpreted when new elements are added. The software and data architecture boundary ensures that information bringing exchange within software libraries is fully documented and understood. The software and hardware architecture boundary establishes the required decoupling and interface abstraction required for portability and extensibility of the implemented functions. These boundaries are especially critical when components that require interaction are crafted by new suppliers or third-parties.

Tying the architecture categories together with a data architecture in this way reduces program risk by easing integration of replacement or new capabilities by adding clearly documented semantics and meaning—something that is lacking in today's MBE tooling.

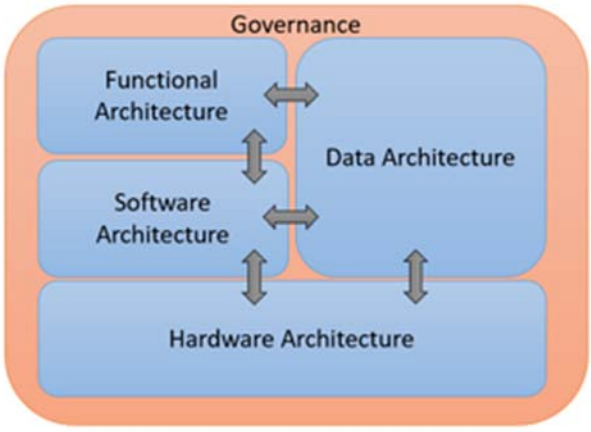


**Figure 4. Data Architecture Prominence in System Architectures**

These formalized concepts will result in individually testable architectural rules, testable relationships between the rules, designs that can be evaluated against the rules prior to implementation, and final products that are testable throughout the development life cycle. In order to accomplish this, standard arrangements of standards are needed to build and group the governing set of rules. A growing and powerful practice for achieving this is to use Technical Reference Frameworks (TRF; Schmidt, 2016).

### *Characteristics of a Technical Reference Framework*

TRFs define implementation-agonistic design environments and patterns that establish a common set of practices for use in a specified context. In effect, a TRF is a

standard for how to use a set of standards to achieve a class of designs. Program managers and their architecture teams can choose TRFs to apply their product requirements against enterprise business drivers, with the goal of creating reusable components and to establish opportunities for any practitioner that can access the environment and add value.

A minimum of three TRFs are needed to craft the full range of military mission systems and for developing reference architectures are shown in Figure 5 (Lethart et al., 2016). Note that the TRFs overlap and transition across the dimensions of criticality and scale. Product requirements may dictate the use of more than one TRF in the development of the reference architecture. For example, a system may require TRF1 for control of a vehicle, TFR2 for command functions, and TRF3 for data analytics of sensor information. TRFs are not aligned with products and systems platforms, but rather the physics-based drivers that guide and constrain how systems get implemented and connected. The three TRFs are summarized as follows:

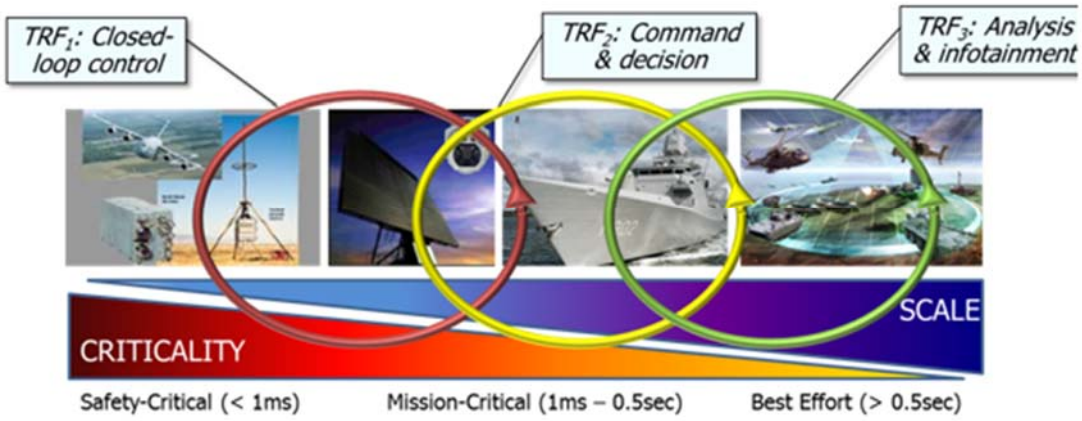


**Figure 5. Technical Reference Frameworks and Time Domains**

Safety Critical (TRF1): This TRF addresses the most critical requirements for the safe and continuous operation of the system or platform, as well as the most demanding design requirements such as personnel or weapon safety. Safety critical requirements are the ones that must take precedence if there is a conflict with other technical aspects of product. Testing these products require high degrees of timing precision and is often coupled to real-world dynamics. Implicit in the design of high availability systems/functions are forms of internal redundancy, unit duplication, direct control and polling of separate solutions, and dedicated allocation and management of resources. However, in the context of TRFs, those patterns remain implementation-agnostic. When applied, the segments of the system designed to TRF1 will meet the highest level of criticality.

Mission-Critical (TRF2): This TRF is applied to functions that comprise the mission capabilities of the platform. Timing and scale are the prime drivers within TRF2. The purpose of TRF2 is to apply modular, data-centric, loosely coupled solutions (e.g., using inversion of control patterns) to create architectural elements that satisfy performance requirements, with stringent end-to-end timing and reliability quality attributes forming key design decisions. Subordinate requirements, such as scale, regulatory compliance, and security are applied in a recursive fashion until all requirements are met.

When applied, TRF2 will manage performance of designs that are highly time sensitive, but not safety critical.

Analysis and Support (TRF3): This TRF is applied to portions of the design that has low criticality, i.e., they may not need to operate on strict time deadlines and or be hardened to survive in harsh environments. Like TRF1 and TRF2, the elements of TRF3 must adhere to the enterprise architecture model as quality attributes of an integrated system or system of systems reference architecture are created. Program managers generally would use this TRF for products that address capabilities associated with analysis, support, and infotainment applications. This analysis is guided by TRF3 patterns, which often involve virtualization, containers, and resource pooling/sharing.

The architecture team, a collaboration between the program office and the T&E community, should evaluate system requirements and assign the appropriate TRF(s) to guide the development and of their reference architecture. The reference architecture then guides and supports continuous testing throughout the development life cycle.

### *Testing a Design Using a Reference Architecture*

Continuing with the building code analogy—an electrical inspector need only worry about the electrical concerns of a project. Whether the structure is business or residential, there are existing guidelines which dictate practices for wire gauge and placement of electrical outlets. Inspections can be performed in phases as construction proceeds, and if an inspection fails at any point, the errors must be remedied before work can proceed. Additionally, there are cross-cutting specifications when electrical passes through framing or is near plumbing and water fixtures.

Instead of testing to a common implementation specification, the architectural-rigorous approach tests against a set of design tenets defined in the reference. Testing allows determination of these independent design elements. The disparate components of the product can then be properly integrated. For example, a builder does not have to wait until the house is fully built to buy faucets. There are specifications that govern the interface between the faucet and the counter and allow one, two, or three holes at various sizes in the counter. These same principles apply with TRFs. Flexibility is preserved with the use of the interfaces between and in the reference architecture categories, without resorting to being forced into reusing legacy implementations that add fragility to the end product.

Each previously introduced aspect of the overall reference architecture, software, hardware, functional, and data, has its own set of test points, tools, and MBE-based documentation practices. This needs to be performed as a carefully considered deconfliction of the related standards and specifications.

The location of these test points establishes considerations for the automation afforded by MBE-based approaches, to include following:

- The T&E Community being engaged in defining and *maintaining* the architectural elements enables testing early and often.
- The product designs can be tested against the architecture well before they are integrated into the system, e.g., testing the faucet design without knowing where it will be installed.
- The ability to test at all levels of the design is established in the context of the architectural elements, e.g., testing whether a home has enough bathrooms doesn't require detailed design of the entire house.

- Creativity in the solution space is preserved by testing a design against architecture principles, vice against a specified implementation.

In short, each reference architecture category is required to be documented with its set of building codes. This defines architecturally where the test points are well before a Program of Record starts. Certainly more can be covered and detailed, but for the purpose of this paper, a set of key testable architectural elements and patters from each reference architecture category are highlighted.

***Software Architecture***

Many books have been written about good software design. They agree on the fundamentals, but then diverge in different directions to add specialized guidance that may be less acceptable depending on what is being implemented. While a TRF is a selection of design-appropriate constraints for the system, there are a few common elements of software architecture that the majority of systems leverage. Highlights include the following:

Design for Orthogonality: Functional units of software are built to perform a specific function and can be swapped with other implementations that perform that same function (with new or improved features). When functional units do one thing, they are constrained to not be doing something else. Despite the obvious nature of that statement, behavior leakage is inadvertently built into software all the time. Products designed to do a single thing can be tested and do not require the rest of the system for a valid result. Occasionally, these tests can be performed in a simple test harness. If the software design is sufficiently granular and the software decoupled, this can allow for early testing.

Minimize Coupling—Interface-Based Designs: Software changes. Implementations change. Designs that focus on a consistent interface are far more resilient to change. This principle is very similar to the notion of orthogonality, but refers to how the software is built rather than isolating functionality. The amount of coupling is an engineering tradeoff that should be most addressed overtly at the end points of decision strings or edges. The location of those edges are a function of the TRF selected and can be at a software library, system executable, or entire virtual operating system.

Test-Driven Design: Another popular methodology is test-driven design. Before any implementation software is developed, the test cases for the corresponding requirement are developed. Then, the code is written to pass the test. This ensures that, at the very least, the requirement is met. The challenge is that designs of tests based on current understanding of the product and how it is expected to function, and those implicit assumptions, are part of the tests and developed software—for example, deciding that the test for a bathroom counter will test for three holes. Unfortunately, the right level of specification is driven by where and how integration flexibility is required, hence the need for the TRF to define the rule sets.

It is not necessary to implement a full test-driven design process in order to benefit from these principles. The value added is that software is tested against an architecture as it is developed, ensuring the design and deployment requirements of the TRF are met. Current MBE-based software engineering tooling provides requirement traceability, test artifact generation, machine parseable documentation of deployment and system topologies, code coverage, other analysis tools, and more. Each serves a function, but

without the traceability to a reference architecture "rule" there isn't a consistent application of their utility.

### *Hardware Architecture*

The T&E community has a long and successful history testing hardware. While this paper is focused on the current software complexity challenges facing today's integrators, there are good examples and lessons from the hardware perspective. The JTAG (IEEE, 2009) standard is just such an example. It provides the physical test point specification as well as the data and signaling IO that the interface supports. The JTAG community didn't invent new technology, but rather assembled a standard of standards (serial communication, power, connectors, etc.) to support their use-cases and defined data in the hardware test domain.

There are parallels in the JTAG standard and what this paper is proposing. Clearly defined separations between the hardware, software, and functional architectures provided the flexibility and endurance of the standard over the past 20 years. How the data is exchanged and packaged over the interface is well understood, but what the data means from a particular device is often documented separately. This highlights what has traditionally been the gap in architecture specifications. The syntax can document a system's interface on MBE-based tooling, but little is done other than human-interpreted prose to document the semantics. This makes every integration a process of discovery of the meaning of the data. Fundamentally, it is the data architecture that has been the missing testable piece.

### *Data Architecture*

Data architecture for interfaces is the newcomer to the conversation. Data isn't new, and database administrators have had an architecture to describe and document data's meaning for quite some time. But about data in motion? In the past, developers have been allowed to simply "create a new message" to communicate system information and state. This approach is sufficient so long as systems remain relatively small and not connected to too many external systems.

Integrating with external systems requires documentation. It requires understanding of the data's format as well as the data's meaning. To date, most development teams have relied on paper-based ICDs and some MBE-based representations of the ICD. While these documents are fine for capturing syntax (the structure: units and data type), they fall short on capturing the semantics, or, what the attributes actually mean. By adopting a rigorous approach to data architecture, the syntactic and semantic rigor in a machine readable and machine-understandable data model can be captured. It is the machine-understandable part that enables MBE-based methods for automating testing data and the interfaces that exchange it. Much like measuring and testing a database against the normal forms (Kent, 1983), testing can instead use the semantic documentation of interfaces for completeness and machine-based utility.

A powerful new integration tool is created once the exercise of capturing this information has been performed. The meaning of data is also captured when the semantics of data have been documented in a machine-understandable format. This means that a computer is able to mathematically process equivalence relationships with absolute certainty, not using stochastic processes. *A priori* integration and analysis of the data exchanges can be performed when starting the design process armed with data architectures. These systems then can be related to each other and analytically determine the overlap/gap of the integration using machine-understandable documentation (Hunt & Allport, 2016).

Data architectures test the structure of the documentation and make explicit the meaning of the content. The definition of the meaning (semantics) of the data in the interface in MBE-based formats allows testing of the interfaces with a defined syntax of their content. A data architecture supports a rigorous functional architecture.

### *Functional Architecture*

Finally, the functional architecture needs to be explored. This reference architectural category is interesting in that many aspects of it get coupled inadvertently with other architectural specifications. For example, the interface is designed to accommodate the hardware's limited bandwidth, address implications of limited compute resources, enforce a singular use of the data, or bake in signaling protocols that dictate implementation patterns. The first decoupling aspect of a functional architecture is to separate interfaces from messages. Often these are implemented as the same thing, which results in significant coupling in order to have an architecture that manages and treats the interfaces and messages as separate testable specifications. Further decoupling includes the functional decomposition of the system itself. This decomposition details each component's role in a system. In order to test the decoupling of the roles in a functional architecture, there are several approaches and techniques (a TRF will have to specify what applies for its domain) that can be leveraged. This is certainly not an exhaustive list, and the interaction with the other architectural would need to be defined.

- Functional Flow Block Diagram. The diagrams define the step-by-step flow of the logical order of execution in the system. In UML, these types of diagrams can be manifested as sequence diagrams. Other standards exist for documenting this implementation detail.
- $N^2$ Chart. These are used in software-intensive system to calculate the coupling between the inputs and outputs of the identified functions (Hitchins, 2003). A decoupled functional architecture limits the dependencies across components in the system and makes it easier to predict impacts of updates to a component in the system.
- Structured Analysis and Design. This methodology can be leveraged to describe systems as a series of functions, with identified inputs, outputs, and supporting data and mechanisms for the function's action. Again, it provides a way to quantify (test) the robustness of a design.

### *Testable Architecture Summary*

In summary, the data architecture provides the binding and traceability between the different architectural segments. The architectural specifications needed to build a design with increasing levels of specificity can link the implementation all the way back to the architectural goals inherent in the applied TRFs. Each level adds detail about the location of the interface test points, the data and its meaning that is exchange over the interface, and methods and implementation patterns that minimize coupling of software. In the end, software is just a specification that is compiled into many very specific instructions for a processor. The specification of a system and its interfaces can be treated with the same level of testable rigor as a software program as long as the right architecture is in place. The assembled systems can then be tested with compliable automation versus human-powered actions.

## The Consequence of Testable Architecture

System and system-of-system testability is enabled by an open architecture. An open architecture is achieved when the rules-of-construction are clearly documented,

deconflicted, and documentable in model-based engineering processes per the selected TRF specification (Figure 6). When a program adopts an architecture specified to this level of rigor, the infrastructure itself becomes an explicit and a separate component of that system. The open infrastructure changes the nature of integration as well the relationship between the T&E and development communities.

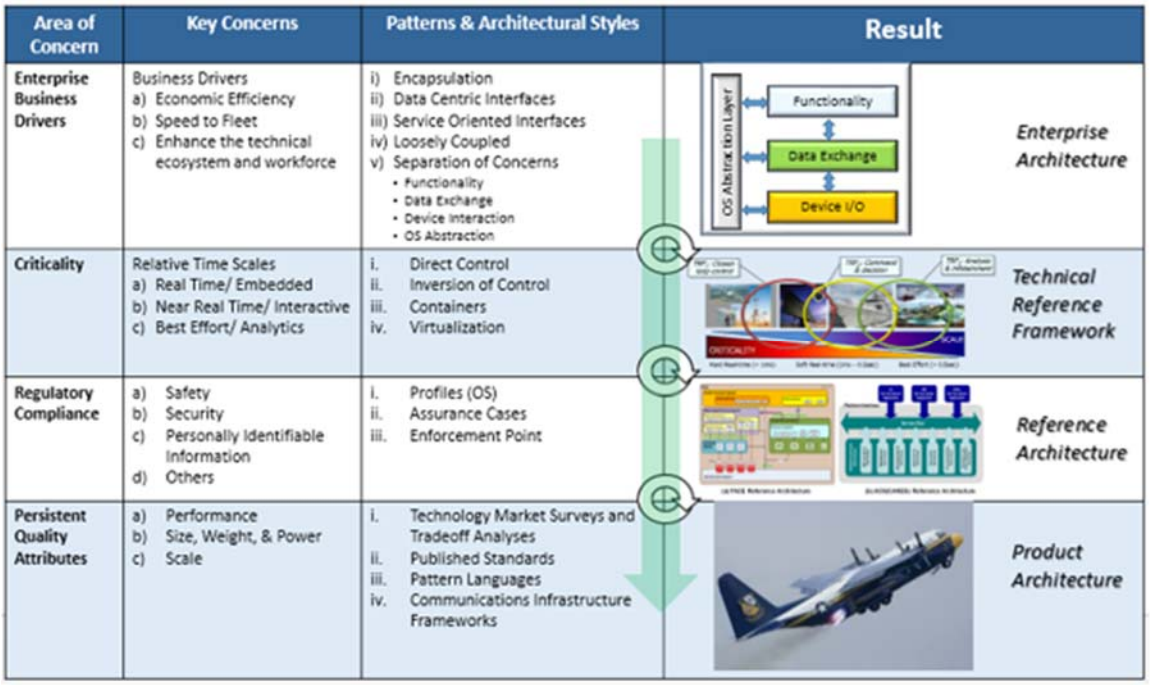


**Figure 6. Realizing a Product Architecture Through a Series of Testable Steps**

The infrastructure can be maintained and provided separately from the platform functional software. Many companies could build modules or systems to the infrastructure and any qualified vendor could then integrate a component and have it function in the system. As an example, the same plumber that built the home is not needed to replace a faucet. Your home is in fact a testable, open architecture, from the design, through the construction, and the following years on maintenance and updates. An open architecture requires an open infrastructure, an open acquisition business model, a technical and operational roadmap, and an organization that can support and maintain these items.

An open infrastructure has three primary characteristics: The first characteristic of an open infrastructure is that it has an open data model that is rigorously defined, described, and fully discoverable. The data model must be completely published, and based upon an abstraction that is broad enough to define the full domain of the system. An abstract data model enables a model to achieve the full breadth of possible implementations, while also defining repeatable interoperable mappings between these possible implementations.

The second characteristic of an open infrastructure is that it is based on open standards and is flexible. Open standards do not limit differentiation, innovation, or competition and they ensure a commodity infrastructure. Flexibility is as important as open standards because there is no one technology or application programming interface (API) that is sufficient for the range of behaviors necessary for a complete, enterprise-level infrastructure. This flexibility is achieved through the use of architectural patterns at the service and interface model. The patterns specify expected behavior while not over-constraining the communications design and build of the infrastructure architecture.

The third characteristic of open infrastructure is that the infrastructure can be developed, acquired, and maintained independently of the functionality of the system. This enables functionality to be acquired independently of the system infrastructure and ensures that capabilities are delivered without subsystem and special-purpose infrastructure dependencies.

Several organizations are achieving these ends. For example, the Army has been doing so for its next generation ground control software (Bellamy, 2014). By defining their software architecture, the decoupling between the hardware and software architectures, and elements of the data architecture, they are laying the groundwork for a generation of incrementally testable components and software.

### *Cultural and Organizational Impacts*

The building code analogy is a powerful example of a robust design and production market where creativity is highly prized, while overarching public good is managed. Architects, standards bodies, contractors, inspectors, and consumers work together to ensure new and exciting products are available to the customer.

Similar to the relationship between the inspectors and the builders, the T&E community must preserve its arms-length relationship with the development community. In this way, they can ensure that products are built with the requisite capability and inherent flexibility to grow over time. To facilitate that shift, the test tools and capabilities must be grounded in making sure that systems have the right architectural features as well as making sure that the unique military capability is delivered.

This change in relationship places the T&E role much earlier in the development cycle and in partnership with the development and operations efforts. They must be a part of setting and evolving the standards going forward and ensuring that the test products address fundamental architectural principles, versus purely on operational capability.

## Conclusion

The historic path of product development will lead to accelerating growth in complexity with unsustainable increases in development and test time and cost. An approach to crafting software intensive products needs to change to address complexity of capability while simplifying the way those products get built, tested, and fielded.

TRFs can be used to establish the “building codes,” or architectural patterns, for the product based on what the design needs to accomplish. As a team, the operators, program manager, development engineers, and test & evaluation experts can select the rules of construction during the early stages of product definition. This becomes the reference architecture and the T&E community sets the stage for how the product will get tested before it gets built—establishing the testing life cycle for the product. Testing starts with validating, early in the product development life cycle, that the architecture of the design will support the intended performance of the requirements. T&E engages early and stays involved throughout development, finding problems as early in the cycle as possible, correcting them where it is efficient and effective, driving down cost, lowering risk, and increasing robustness.

A natural consequence of these practices is to end up with a new and separate component of the design: the infrastructure. Furthermore, by infusing that infrastructure with configurable variation points, a product line architecture is created that can be used to quickly instantiate alternative downstream implementations—a critical enabler for enterprise reuse. In addition, by using TRFs that are widely practiced, any qualified vendor can create

capabilities that can be added to the product line. Lastly, the tools to the trade of integration and test are known and practicable by that same community of practitioners, such that the role of integration or testing can be risk-prudently performed by alternative vendors.

## References


Allport, C., Hunt, G., & Revill, G. (2016). Rethinking system integration. *Electronic Engineering Journal*. Retrieved from http://www.eejournal.com/archives/articles/20160609-interoperability

Architecture Analysis and Design Language. (n.d.) What is AADL? Retrieved from http://www.aadl.info/aadl/currentsite/

Bellamy, W., III (2014, September 8). New software framework coming to Army UAS ground stations. Retrieved from http://www.aviationtoday.com/2014/09/08/new-software-framework-coming-to-army-uas-ground-stations/

Capilla, R., Bosch, J., Trinidad, P., Ruiz-Cortés, A., & Hinchey, M. (2014). An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. *Journal of Systems and Software, 91*, 3–23.

DoD. (2017). Systems engineering. In *Defense acquisition guidebook* (Ch. 3). Defense Acquisition University.

Deputy Assistant Secretary of Defense for Developmental Test and Evaluation (DT&E). (2016). *Developmental test and evaluation annual report for FY2015*.

Deputy Secretary of Defense for Systems Engineering. (n.d.). Retrieved from http://www.acq.osd.mil/se/http://www.acq.osd.mil/se/

Elfriede, D., Rashka, J., & Paul, J. (1999). *Automated software testing: Introduction, management, and performance.*

Guertin, N., Schmidt, D. C., & Sweeney, R. (2015). How the Navy is using open systems architecture to revolutionize capability acquisition. In *Proceedings of the 12th Annual Acquisition Research Symposium*. Monterey, CA: Naval Postgraduate School.

Guertin, N., & VanBenthem, P . (2016). Modularity and open systems architecture applied to the flexible modular warship. *ASNE Journal*.

Hitchins, D. K. (2003). *Advanced systems thinking, engineering, and management*. Artech House.

Hohpe, W. (2004). Enter*prise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley.

Hosain, S. Z. (2016, June 28). Reality check: 50B IoT devices connected by 2020—Beyond the hype and into reality. Retrieved from http://www.rcrwireless.com/20160628/opinion/reality-check-50b-iot-devices-connected-2020-beyond-hype-reality-tag10

How many millions of lines of code does it take? [Graph] (2017, February 8). Retrieved from http://www.visualcapitalist.com/millions-lines-of-code/

Hunt, G., & Allport, C. (2016). Disambiguate data with documentation. Future Airborne Capability Environment Technical Interchange Meeting. The Open Group.

IEEE. (2009). IEEE standard for reduced-pin and enhanced-functionality test access port and boundary-scan architecture (1149.7 Standard).

Jamshidi, M. (2008). *System of systems engineering: Innovations for the twenty-first century.*

Kent, W. (1983). A simple guide to five normal forms in relational database theory. *Communications of the ACM, 26,* 120–125.

Kuhn, Kacker, & Lei. (2013). *Introduction to combinatorial testing*. Chapman and Hall/CRC.

Lethart, R., Porter, A., Schmidt, D., O'Hare, M., Crisp, H., & Laird, B. (2016). *Capability-based technical reference frameworks for open system architecture implementations*. Presented at the SEDC 2016 Conference. Retrieved from http://www.sedcconference.org/capability-based-technical-reference-frameworks-for-open-system-architecture-implementations/

Micouin, P. (2014). *Model based systems engineering: Fundamentals and methods.*

National Defense Authorization Act. (2017). Section 805.

The Open Group. (n.d.). FACE technical standard. Retrieved from http://www.opengroup.org/face

Rahman. (2014). DevOps, continuous delivery (CD) and agile—Is this going to change the world of software engineering? *Continuous engineering for dummie*s. IBM, Wiley.

Richardson. (2016). *Design for maintaining maritime superiority*.

Schmidt, D. C. (2016, July 11). A naval perspective on open-systems architecture [Blog post]. SEI Blog. Retrieved from https://insights.sei.cmu.edu/sei_blog/2016/07/a-naval-perspective-on-open-systems-architecture.html

Tolk, A., & Muguira, J. A. (2003). The levels of conceptual interoperability model (LCIM). In Proceedings of the *IEEE Fall Simulation Interoperability Workshop*. IEEE CS Press.


**Acquisition Research Program**
**Graduate School of Business & Public Policy**
**Naval Postgraduate School**
**555 Dyer Road, Ingersoll Hall**
**Monterey, CA 93943**

www.acquisitionresearch.net