ABSTRACT

Title of Document: A MODEL FOR ESTIMATING THE COST TRADEOFFS ASSOCIATED WITH OPEN ELECTRONIC SYSTEMS

Zev Schramm, Master of Science, 2013

Directed By: Dr. Peter Sandborn, Professor of Mechanical Engineering

An open systems approach (OSA), especially when used in conjunction with modular architecture, reuse, and harnessing of existing (COTS or proprietary) technologies, is commonly associated with cost avoidances resulting from: more efficient design; increased competition among suppliers; more efficient innovation and technology insertion; and modularization of qualification. However, OSA strategies require investment and may increase risk exposure. To determine if openness should be pursued, and to what degree, a quantitative model assessing the costs associated with openness is required.

Previous attempts to measure openness rely on qualitative measures, and cannot be used to estimate the life cycle cost impacts of openness. The model developed in this thesis quantitatively determines the effects of openness on life cycle cost. The life cycle cost difference between two implementations with differing levels of openness was calculated for a case study of an ARCI sonar system, providing insight into the value of openness. The case study performed in this thesis provides the first known quantitative support for Abts' COTS-LIMO hypothesis that increasing CFD increases cost avoidance. However, these results challenge Henderson's implicit assumption that marginal openness is always positive (increasing openness is always beneficial).

A MODEL FOR ESTIMATING THE COST TRADEOFFS ASSOCIATED WITH OPEN ELECTRONIC SYSTEMS

By

Zev Schramm

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Masters of Science
2013

Advisory Committee:
Dr. Peter Sandborn, Chair
Dr. Linda Schmidt
Dr. Patrick McCluskey

Acknowledgements

This thesis would not have been possible without the gracious support of many individuals. First and foremost, I would like to thank my mentor and advisor, Dr. Peter Sandborn, who has spent countless hours patiently providing advice, direction, and encouragement. I have great appreciation for Gregory Twaites, Nathan Hastad, and the entire General Dynamics team, who have helped by sponsoring my research and by acting as a sounding board for my ideas.

I would also like to thank Dr. Linda Schmidt and Dr. Patrick McCluskey, for taking the time to serve on my thesis committee (and Dr. Bilal Ayyub for offering to do so). Additionally, I must express my gratitude to Mrs. Terry Island and Mr. Amarildo Damata for their help throughout my undergraduate and graduate studies at the University of Maryland.

Finally, I would like to thank my wife, Michele, for her love, support, and understanding these last several months and years. Michele, you are my angel and my inspiration. You give me hope and strength in times of adversity and of success. I love you very much.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1: Motivation

Historically, when large, complex electronic systems[1] were created, critical functionality was provided by bespoke components that were designed for the specific conditions and requirements. This was especially true for military systems, where state-of-the-art and high-performance components are necessary for the successful and safe completion of assignments (mission critical and safety critical) [1]. The use of custom-made components (hardware and software) results in long development times and high development costs, as significant effort is expended to recreate functionality that is, increasingly, commercially available [2, 3].

In the last few decades, technological advancement has proceeded at break-neck pace, with computing power increasing exponentially, while hardware simultaneously shrinking in both size and weight. This has allowed electronics to become more general, so that hardware and software components can be designed once, and then used in many different applications [1, 4]. These advancements have

[1] Throughout this thesis, the term "system" will be used to mean any high-level "system of systems", made up of some number of subsystems and components. A system may be a ship, airplane, or radar. The term "component" will be used to refer to any lowest-level part, be it hardware, software, or some combination of the two, which is designed or procured as a single unit. The term "enterprise" will be used to refer to any entity that defines and maintains a system design, and then employs one or more instances of that system to fulfill operational requirements. An enterprise may maintain designs and provide operational support for several types of systems at the same time with similar or diverging purposes. For example, Rolls Royce offers several types of aircraft engines, while General Dynamics designs aerospace, naval, and land-based systems.

increased the viability of using Open Systems Architectures (OSA) in general, and a Modular Open Systems Approach (MOSA) in particular [5, 6].[2]

Generally, it is taken for granted that the use of OSA and MOSA principles is a way to decrease the total life cycle cost of a system. Leveraging existing, open technology, including commercial off-the-shelf (COTS) components, avoids much of the costs associated with designing a component from scratch, and the time required for development or refresh of the system can be greatly reduced [7]. This allows for faster, less expensive, and more frequent design refreshes, which helps mitigate the effects of obsolescence and can lengthen the life of the system. Frequent refreshes allow for the insertion of new or improved technologies, allowing incremental improvements to the system during its use. This is sometimes referred to as "spiral development" [6, 8]. Use of well-defined standards promotes smooth interfacing both within and between systems. Use of common component types fosters competition between suppliers, which can reduce the component's procurement cost. Reuse of components (within and between systems) eliminates redundant components, and allows for economies of scale, further reducing costs.

However, the use of an open methodology represents a tradeoff because there are costs associated with using an open methodology. For example, COTS components may not meet the required size, weight, or performance specifications; those that do may be prohibitively expensive. Further, use of commercial components

[2] OSA is sometimes used to mean Open Systems Approach, the set of principles, and the deployment thereof, used to design systems with an Open Systems Architecture. Likewise, MOSA is also used to mean a Modular Open Systems Architecture, an architecture that is designed in accordance with MOSA principles.

to build a subsystem may require the use of generalized components with unnecessary additional functionality, adding to the cost, size, and complexity of the subsystem. A specially designed subsystem may be simpler and less bulky. Additionally, unnecessary functionality and complexity increase the cost of qualifying the component and system, and could also increase the effective failure rate [9, 10].[3] Moreover, by using of commercial instead of proprietary components, the enterprise loses some control over the supply chain, unless the enterprise has a particularly large demand for the component [11]. COTS components are also more volatile than proprietary components, receiving minor[4] updates more often, and becoming obsolete more quickly. This makes it desirable to refresh MOSA designs more frequently [12, 13]. In addition to added refresh costs, refreshing more frequently may lead to an increase in the number of fielded configurations, with an accompanying increase in outlays for logistics to track and support each of these versions. These factors mean that there are situations in which a more open system could end up being more expensive than a "closed" one. It is therefore important to be able to quantify openness and to predict the total costs avoided and added due to its openness.

[3] For example, the initial flight of the Ariane 5 rocket suffered catastrophic failure due to unnecessary functionality in a software component that was reused from the Ariane 4 rocket [10].
[4] A "minor" change or update is one, such as a bug fix or manufacturing process change, that does not significantly affect the use of the component. A "major" change affects the component's use, such as an interface update or a change in technology, is modeled as an obsolescence event.

1.2: Modular Open Systems Approach (MOSA)

MOSA is a combined business and engineering strategy that draws on a modular design and the use of open standards when developing a new system or updating an existing one [6]. Modular design is an approach in which a system's required functionality is divided amongst its components or subsystems (modules), such that each module can be deployed independently. The United States Navy's first attempts at using modular systems began with the Seas Systems Modification and Modernization by Modularity (SEAMOD) program in 1975 and the Ship Systems Engineering Standards (SSES) program in 1980. These efforts were not realized, in part because limitations on computing power, longer component procurement lives, and a lack of popular, well-defined open standards meant that there was little impetus or desire to challenge the status quo [14]. By 1994, these obstacles had been diminished, and the Open Systems Joint Task Force (OSJTF) was created to establish and encourage the use of MOSA. In 2003, the Department of Defense (DOD) published a directive, DODD 5000.1, which required that "a modular, open systems approach shall be employed, where feasible" [15, 7].

Design modularity takes many forms. Use of interchangeable parts dates back to at least the 18th century, when Honoré Blanc and then Eli Whitney demonstrated that by standardizing the dimensions of gun components to a strict tolerance, guns could be assembled and repaired with stocked components, instead of requiring a specially made component [16]. This insight, crucial to production during the industrial revolution, is the concept of *maintenance modularity*, which allows individual components or subassemblies to be interchangeable, regardless of their

manufacturer, the system they are installed in, or their location within that system. Closely related is the idea of *production modularity*.[5] Production modularity allows for complex subsystems to be constructed and tested independently in specialized locations and then brought together for final assembly. This allows for shorter assembly time [14]. *Mission modularity* is achieved by designing interchangeable subsystems that serve different functions. The system can then be quickly reconfigured depending on the requirements of the current mission. This was used for the Littoral Combat Ship (LCS), so that it could complete diverse missions. Its dedicated mission modules include anti-submarine warfare (ASW), anti-surface warfare (SUW), and mine countermeasure (MCM) [5]. When the same component or specialized module is used in several different types of systems or platforms, it may be called *component sharing*. Component sharing reduces the number of component types used, simplifying logistics.

The use of a modular, open systems approach is assumed to reduce costs by increasing the interoperability, maintainability, extensibility, composability, and reusability of the system [17]. Each of these goals is defined below.

1.2.1: Interoperability

IEEE defines interoperability as "the ability of two or more systems or components to exchange information and to use the information that has been exchanged" [18]. For example, different types of aircraft must all be able to interact

[5] Also called construction modularity

using the same radio and air-traffic control systems. The concept of interoperability can also be applied to different versions of the same component – if one workstation is upgraded to newer software, it should still be able to communicate properly with the other workstations. Interoperability is achieved by using standardized interface protocols in place of proprietary ones for both hardware and software [6, 19]. Well-defined interface standards allow components that were designed separately to work harmoniously, even if one or both of them have been repurposed from a different application. Modularity is crucial to allow for the different components to be connected together in new configurations. An example of interoperability is the standardization of USB drives. USB drives are modular, so a computer with a USB drive connection is compatible with any USB device, and each USB device can be used with any computer (see also Composability). For the most part, this interoperability transcends the hardware, software, and operating system used. This reduces the number of specialized ports a computer needs, and allows for greater flexibility of setup and choice of components. Additionally, since USB is an open standard, anyone can design a USB device, and there are many vendors to choose from when purchasing a USB device.

1.2.2: Maintainability

Maintainability is the ability to easily locate (diagnose) and correct defects, and the ability to operate in a given environment and performance level without an undue number of failures. This applies to hardware and software at both the

component and system levels. Failed hardware should be easily replaceable – male/female ports or screw connections are preferable to soldered ones. For software, maintainability deals not only with the initial quality of the code, but the quality of improvements and updates made to it [20]. Additionally, software should not be hardware dependent. Middleware should be used handle and hardware-specific features. This allows the same software to run on many different hardware arrangements, with only superficial modifications [2]. In general, using open systems increases maintainability[6] because widely used and accepted standards are usually well understood – any design issues have had time to be resolved (or at least documented) by the community using that standard. This knowledge base is invaluable, and can be used to help diagnose faulty components, or to identify components that, while theoretically interoperable, do not "play nicely" together. If new bugs are found, the speed and skill with which it will be resolved is improved, because there is a larger community supporting the standard [21]. Designing and using a proprietary system, on the other hand, necessitates going through the debugging and troubleshooting process from the beginning, without outside assistance. Modularity also helps improve maintainability, as it allows failed components to be quickly swapped out for operational ones with the same function, even if the new components are not the same as the original. For software, modularity allows a bug-ridden or inefficient subroutine to be replaced with a new one that is of

[6] Use of open components, particularly COTS components, may increase operation and support costs because of the need to license the IP, subscription fees for maintenance updates, and unforeseen incompatibilities between two components that must be resolved by selection of a new component or redesign of the interface between them [59].

higher quality. Modularity is likewise useful for managing obsolescence and for allowing incremental upgrades. These ideas connect directly to the goals of Extensibility and Composability.

1.2.3: Extensibility

Extensibility measures the degree to which new technologies can be quickly and efficiently inserted into an existing system. The new technologies may allow for increased performance of an old function, or may introduce entirely new functionality. Extensibility is also closely related to scalability, the ability of the system to handle a greater work volume or a greater number of inputs. Scalability is effectively a measure of extensibility of performance, particularly when the added technology is further instances of the original subsystem. As explained above, extensibility can be accomplished through the use of standardized interfaces and modular architecture so that a new component with the same interface standard and similar form factor can replace an older component. Extensibility may be applied to individual components or a cohesive subsystem. An example of extensibility due to use of open standards is the ability to upgrade the DRAM module on a computer from 1 GB to 2 GB.

1.2.4: Composability

Composability is the ability to use (and reuse) systems or subsystems independently of one another. In order for subsystems to be composable, they must be

modular and self-contained so that the subsystem can be deployed without any other accompanying items [22]. Additionally, the interface standard used must be well defined and consistent across all subsystems. Ideally, each component should act as a "black box" which has certain inputs and outputs as defined by its interface standard – the exact mechanics of its inner workings should be irrelevant to the surrounding components. Composability allows an enterprise to select from components from a library and assemble them into different configurations to create different systems that can fulfill unique sets of requirements [22].

1.2.5: Reusability

The capacity to take an old system and apply it to new uses is the measure of its reusability. Reusability applies within a single system design (using the same microprocessor or chip for all boards in System A); across unrelated designs (two or more independent systems designed and produced by the same enterprise); and even between enterprises [4, 23]. Like composability, reusability is facilitated by the use of open standards and modular architecture. Reuse allows for design requirements to be met with existing components, without the significant investment that would otherwise be required to create a new design, qualify a new component, or qualify a new vendor. Reuse also reduces the number of unique components that must be tracked and supported by an enterprise [24].

Modular, open architectures are made up of one or more subsystems that interact via a modular backbone. When possible, well-defined, publicly available

(open) standards are used, particularly for the system's "key interfaces". Key interfaces are interfaces that are used to pass the most important information between functionally adjacent components or subsystems [6, 17]. An interface may also be designated as a key interface if it connects a technologically volatile subsystem to a more stable one [7]. Defining standards for these interfaces helps to isolate them from one another, protecting against the ripple effect [25]. Modularity is a vital component of this strategy, because a modular architecture allows for the replacement or redesign

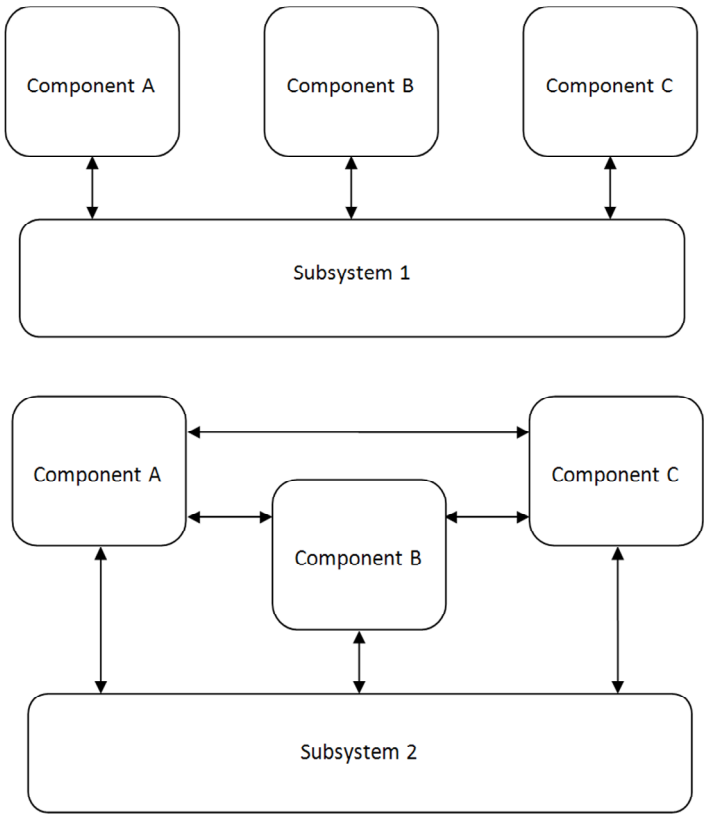


Figure 1: Subsystem 1 uses a modular architecture, and the components do not interact directly. A change to one component is therefore isolated to that component, which may be requalified independently. Subsystem 2 does not use a modular architecture, and a change to one component affects all components. Updating one component requires all components be updated and requalified. This is called the ripple effect.

of one subsystem with minimal impact on the surrounding subsystems (Figure 1). This enables subsystems to be re-qualified independently [26]. As such, refreshes

may be conducted more frequently, which allows for improved performance and increased functionality. The modular architecture also increases the system's scalability and extensibility, since new modules (or additional instances of existing ones) may be added more easily [27]. By utilizing open standards, an enterprise can leverage existing technology and harness commercial off-the-shelf (COTS) components, instead of developing and maintaining the technology itself [8, 2, 24, 3].

1.3: Openness and Open Architecture

1.3.1: Defining Openness

The terms "*open*" and "*openness*" are used in a wide range of fields, and have a broad definition. In general, openness refers to the characteristics of availability, transparency, and not hiding or obstructing access to information. When applied to objects such as architectures, systems, or components, *openness* is a measure of the *degree* to which an object conforms to a set of defined *standards*, and the openness of those standards. A *standard* is a formal definition of the methodology and structure to be used by a component or system. An interface standard is a standard that focuses on how a component interacts with its surrounding components (as opposed to the internal operation of the component).

The exact degree of openness of a standard or system depends on the context in which it is being used. The word "degree" is used because openness is a loosely defined metric that is best measured on a comparative scale. While idealized "fully open" and "fully closed" standards and systems may be conceived, most standards

fall somewhere in the middle of this range. Few, if any, completely open systems exist, because all real-world systems use architectures, components, and standards that are, at some level, based on the intellectual property of one or more enterprises that are not the product designer, manufacturer, or supporter.

The relative openness of two different standards depends on the definition of openness used. Some definitions of openness put requirements on the method used to create the standard – open standards are those that are developed using a collaborative or quasi-public effort, where all parties who will use the standard have a say in its creation [28, 29]. By this definition, de-facto standards that were initially created by a single entity are mostly closed, even if all of the details were subsequently published and made available to the general public.

Other definitions of openness focus on the "availability and accessibility" [21] or interoperability [30] of the standard. Hanratty, et al, for example, define openness as standards, systems, or components that can be "supported by the marketplace, rather than being supported by a single (or limited) set of suppliers" [9]. By these definitions, a de-facto standard may be made open by publishing its exact specifications, and making it available for use by the general public on reasonable and non-discriminatory (RAND) terms [31]. (Outside of the US, this type of license is commonly referred to as a fair, reasonable, and non-discriminatory, or FRAND license [32, 33].) Though licensing under RAND terms has legal implications, the terms "fair" and "reasonable" are amorphous and ill-defined. There is little legal precedent, and the meaning of these terms is heavily dependent on the organization and situation in which it is used [34, 32, 33, 35]. This means that although two

standards are both available under RAND licenses, they may have different levels of openness.

Though reasonable royalties are generally a barrier to entry, some require the standard to be entirely free to use, without any royalties or licensing fees, in order to be considered accessible. This is primarily due to the influence of the "Open Source" software movement. Much open source software is developed and used by individuals or small private enterprises, where any fee, no matter how small creates a barrier that makes the standard inaccessible [21]. For the purposes of this thesis, where the primary focus is on systems designed and maintained by larger, more traditional enterprises, the former characterization is more useful. We will use the following definitions:

Definition of Open Standard

An Open Standard is any well documented, publicly accessible, and unrestricted standard that is either: a) Royalty Free (RF), or in some cases b) may be licensed for a small fee (RAND).[7] The openness of a standard may be increased by increasing the ease of access for external parties to access its definition. This may be done by publishing it, limiting or removing licensing requirements, and reducing associated fees.

[7] For simplicity, the term RF is used here to mean license that has no associated usage or IP fees, and RAND to mean licenses that have any type of fee, however minimal. In actuality, some RAND licenses allow for usage without fees.

Definition of Open Architecture

An Open Architecture is any architecture that predominantly or exclusively employs open standards to define the interfaces between its components. The openness of an architecture may be increased by switching to standards that are more open, and by utilizing fewer standards overall.

To be considered well documented, the Standards Defining Organization (SDO) must provide enough detail of the specifications so that any other interested parties can use the standard to design competing interoperable components. This means that all details must be published, and no features hidden or withheld. Additionally, the SDO should provide a process for revising the standard to incorporate any fixes or emendations [36].

A publicly accessible standard is one that is equally available to anyone who wants to implement it. For fully open standards, no restrictions may be placed on the use of that standard, including limitations on its reuse, modification, or extension. However, the SDO may require that any modifications or extensions be published and licensed under the same terms as the original standard [37].

In order to increase the accessibility of the standard, and thereby foster competition, the standard must be available either for free, or for some minimal cost [38]. In recent years, many organizations have required standards to be completely free to distribute, implement, and utilize (RF licensing) in order to be considered open, and have excluded more general RAND licensed standards from their lists of open standards. Some have allowed nominal fees (to cover the cost of publishing) to

be charged for copies of the standard's documentation, with the condition that the purchasing entity has full rights to copy and distribute it freely. Exception is also made for optional certification services, which the SDO may provide at any charge [37].

Though the precise meaning of a RAND (or FRAND) license depends on the intent of the SDO, their general intent is apparent. The standard in question must be made available fairly, in a manner that is not anti-competitive. This includes prohibitions against practices that place undue limitations that could block an interested party from licensing the standard [39]. The license must be available for a reasonable royalty, where 'reasonable' is determined by several factors, including the cost to develop the standard, the value of the standard on the open market, the existence of alternative standards, and the royalties charged by other SDOs for comparable situations [40]. The royalty and license must also be non-discriminatory – the SDO may not, for any reason, charge different royalties to different groups, require unpaid licenses to others' intellectual property (uncompensated grant-backs), or refuse to license any party that accepts the terms of the RAND license [40]. For our purposes, publicly published standards that use RAND licensing are still considered open, even though the licensing fees make them less accessible than RF licensed standards.

1.3.2: Open Source

Care should be taken not to conflate openness or Open Architecture with the principles of Open Source. While the terminology and some of the underlying ideas are similar, the two are different things. Openness as defined previously is applied to both hardware and software components, and is measured on a scale from fully open to fully closed. While the many notions of openness agree in guiding principles, there is no single definition. As such, a Standard that was initially developed by a single entity and subsequently published for a minimal fee under a RAND license would receive a high openness score under our definition, but would score relatively poorly by another definition.

The term "Open Source" was coined by Christine Peterson in 1998 after the release of Netscape's source code and in conjunction with the formation of the Open Source Initiative (OSI) [41]. As such, the term refers almost exclusively to software,[8] and has been authoritatively defined by the Open Source Definition [42]:

1. Free, unrestricted redistribution, including when the software is sold as part of a larger work
2. The Source Code must be accessible and unobscured
3. Modification and extensions must be allowed so long as they are distributed under the terms of the original software

[8] The Open Source philosophy predates the computer era, and is still present today in non-software fields. As a result of the proliferation of the Open Source movement, its terminology has begun to be applied to other fields.

4. The license may restrict modification of the source code on the condition that "patch files" which modify the code when it is compiled are allowed. Any extensions may be required to use a different name or version number.
5. No discrimination of persons, groups, or fields of endeavor
6. The license must be distributed with the software – anyone who receives the software are free to edit and distribute it without additional licensing
7. The license must not be product specific
8. The license must not restrict other software
9. The license must be technology neutral

According to this definition, a software item either qualifies as open source, or does not - there is no concept of "partially open". This binary approach means that many standards that were once considered to be relatively open (using the definition of openness put forth in the previous section, and similar definitions) have been reclassified as not open, and has led many SDOs to make their definitions of openness more rigorous. A key example is the exclusion of standards that have any usage or IP fees. For traditional businesses, small fees are not a significant impediment, and so RAND licensed standards may be considered open. The Open Source community, on the other hand, is defined by the voluntary collaboration of many developers, and a final product that is offered to the public free of charge. In this situation, any fee, no matter how small, becomes an insurmountable barrier to

entry. However, the Open Source community does recognize a fee-to-use standard as open provided that its fees are waved for Open Source projects.

1.3.3: COTS and Openness

Though, as discussed, there are benefits inherent to using and open approach, an important benefit of using open standards is that it enables the enterprise to harness off-the-shelf items. Many of the benefits mentioned are enabled by the use of off-the-shelf items. Other benefits, which exist without the use of off-the-shelf items, are still enhanced by this capability [6].

Commercial Off-The-Shelf, or COTS, is a broad term used to describe commercially available, general-purpose hardware and software components. COTS components are, by their nature, open to some degree. Many COTS items are fully open – based on an open standard, with similar components available from multiple sources (USB based hardware and software, for example). Other COTS components, though based on proprietary standards, are also open to some degree, because the standard is made available for others to use. This is true of Apple's proprietary connectors, for example the new "Lightning" connection port. Apple maintains sole control over the standard's definition, and strictly regulates who is allowed to license it, under what terms, and for which products [43]. In such cases, the overall openness of the standard is correlated to its final availability and accessibility. The presence of COTS hardware or software is enough to indicate that a system is, to some degree,

open, because use of COTS components demonstrates that the system is not designed exclusively using inaccessible proprietary standards.

1.4: Effects of Openness on Life Cycle Cost

While the explicit goals of using MOSA are to increase the interoperability, maintainability, extensibility, composability, and reusability of the system, there are other benefits associated with using open architectures and open standards, including increased competition, increased innovation, improved quality, fewer issues due to obsolescence of system components, reduced risk, and reduced development time. Many of these benefits, which stem simply from the fact that other enterprises are using the same standards, are called network effects or network externalities [40]. Each of these benefits, which results in significant cost avoidance, has been observed by enterprises employing an OSA. However, several case studies have also noted that the use of OSA strategies has led to an increased support cost. The viability of an open systems approach depends not only on the architecture and components used, but on the number of fielded systems, obsolescence mitigation strategy, refresh plan, and the number of years the systems are to be supported.

1.4.1: Increased Interoperability

The use of open standards is crucial to fostering interoperability. It allows many different products to work together despite being created and used by different

enterprises [35]. One example of this is the NAVSTAR GPS system, where standardization and publication of the communication signal made it possible for numerous other enterprises, including private companies and foreign governments, to utilize the system. Similarly, standardization of the hardware and receiver interfaces made it easy to implement the system across more than 30 different aircraft platforms. This increased commonality also helped to decrease the number of fielded configurations, and reduce the logistics footprint, avoiding significant costs [44]. The benefits of interoperability were also seen by the Joint Precision Approach and Landing System (JPALS). In this case, standards were coordinated with the FAA and other international bodies to ensure that civilian and military aircraft from many different countries are able to know each other's precise positioning, and land safely, all around the world [45]. This increases safety, supportability, and readiness, all while reducing costs.

1.4.2: Increased Maintainability

The use of open standards also increases the maintainability of a system by simplifying and normalizing maintenance procedures. When COTS components are used, some of the support responsibilities may be shifted to the component manufacturer [8]. This is especially true in the cases of software and throw-away hardware (or hardware covered by a warranty). The reuse of standards and components means that fewer manuals, less training, and less specialized test equipment are required [46]. Fewer types of spare components must be tracked,

purchased, and stocked, reducing the logistics footprint. This benefit of standardization was clearly during the design of the *Virginia* class of submarines. While previously designed classes of submarines had seen the number of parts proliferate, with the *Trident* class using 28,000 components, the *Los Angeles* class using 29,000 components, and the *Seawolf* class using 45,000 components, the *Virginia* class used only 15,00 components [24]. Standardization also allowed for a 32% decrease in testing equipment, avoiding $50,000 in costs per ship. Overall, a $27 million investment in openness and standardization was projected to avoid $789 million in costs [24].

1.4.3: Increased Extensibility

Use of open standards allows a system to benefit from improvements and new innovations from the public marketplace without any up-front investment costs. These enhancements can then be adapted and installed much more quickly and efficiently than could be done for a proprietary system. By using the Weapons Systems Common Operating Environment (WSCOE) and Portable Operation System Interface (POSIX) defined by IEEE, the Weapon System Technical Architecture Working Group (WSTAWG) was able to augment the Predator Unmanned Arial Vehicle (UAV) for use with the Hellfire missile in just over one month. This was made possible by porting target tracking software from a Line-of-Sight Anti-Tank (LOSAT) missile. This reduced development time considerably, and avoided 75% of the typical software development costs [7].

1.4.4: Increased Reuse

Reuse within a design avoids costs by allowing for redundant components that serve similar purposes to be eliminated, reducing the number of unique components that must be qualified, purchased, and stocked in order to produce and maintain a system. Further cost avoidance is achieved by decreasing the logistical footprint of the system by reducing the number of components for which part libraries must be maintained, tooling and infrastructure supported, spares stocked, and technicians trained [4, 47]. Reuse between systems occurs when a component from an earlier application is integrated into the design of subsequent systems. This greatly reduces the time and cost to design the systems, and effectively lessens the over costs per system associated with qualifying components, maintaining component libraries, training technicians, sustaining maintenance and support operations, and other non-recurring costs by amortizing them across several projects. This benefit was seen after designing the Virginia Class of submarines. 45% of the components used in the Virginia design were reused in creating the USS Jimmy Carter Multi-Mission Platform, and an additional 64% of the Virginia components were reused in designing the SSGN Class of submarines. Combined, this resulted in costs avoidances of over $150 million [24]. High levels of reuse also allows for maximum benefit due to the effects of learning and by leveraging economies of scale [48, 49].

1.4.5: Increased Competition

Open standards allow for many different suppliers to produce similar and even interchangeable components that can perform the same function. This empowers an enterprise designing a complex system to choose more freely between multiple suppliers. Since components from several sources are interoperable, the enterprise may switch between them without incurring significant cost. This would not be the case if proprietary standards were used, where the high costs to switch to a different standard would form a barrier, locking the enterprise into the original source. The freedom to switch between suppliers helps increase competition between them. If the enterprise can get a component with similar functionality from another source for lower cost, there is little to prevent the enterprise from switching sources. This incentivizes the different suppliers to offer their component for a lower cost, or to otherwise appeal to the enterprise by offering other benefits, such as increased levels or types of functionality, or an increased warranty [35]. The benefits of increased competition are apparent in many case studies, including the A-RCI sonar system [2], mechanically attached pipe fittings [50], and standardization of aircraft batteries and related components [49]. In this last case, it was estimated that the availability of more than one supply source reduced costs as much as 25 to 30%.

1.4.6: Increased Innovation

Open standards increase innovation for several reasons. First, increased competition, as mentioned previously, spurs component manufacturers to distinguish

their component from the rest of the field. This can be done by increasing its performance, decreasing its form-factor, or developing some new valuable functionality [35]. Additionally, for the SDOs and product suppliers to make a profit, the standard they promote must be long-lived, so that they will have sufficient time to recoup the initial development costs. This gives them incentive to produce new innovations that distinguish their standard from other available standards. In the case where one open standard controls most of the marketplace, innovation is still encouraged because the standard's popularity guarantees its longevity, so developers see less risk in investing additional time and money to improve the standard or introduce a new product [35, 40]. This helped encourage the spread of the Internet and the World Wide Web, thanks to the open standards of TCP/IP and HTML respectively [21].

1.4.7: Improved Quality

The definitions of many open standards are set by the SDO after consulting with many experts, and taking input from many of the key market players. This allows them to pool their experience, and produce a better standard than any one enterprise would have independently [35]. Use of open standards allows for higher quality by increasing competition between suppliers. The required functionality may even be provided by components designed and supplied by another enterprise that specializes in the functionality required.

1.4.8: Mitigating Obsolescence

Diminishing Manufacturing Sources and Material Shortages (DMSMS) obsolescence occurs when components required to support a system become obsolete or are otherwise unprocurable. In the last few decades, DMSMS obsolescence has become a significant issue because of fast turnover and short procurement lives in the modern component market. This means that the support life of some systems is many times longer than the procurement lives of their constituent components [51, 26, 52]. If spare components are unavailable, it may be impossible to continue to support an existing system. This may cause a crisis for complex systems, since the long-development cycle for such systems means DMSMS obsolescence issues may be present before the first system is even fielded. A stark example of this phenomena is a sonar system detailed in Singh and Sandborn [51], which faced obsolescence of 70% of its commercial components before it was initially deployed (Figure 2).

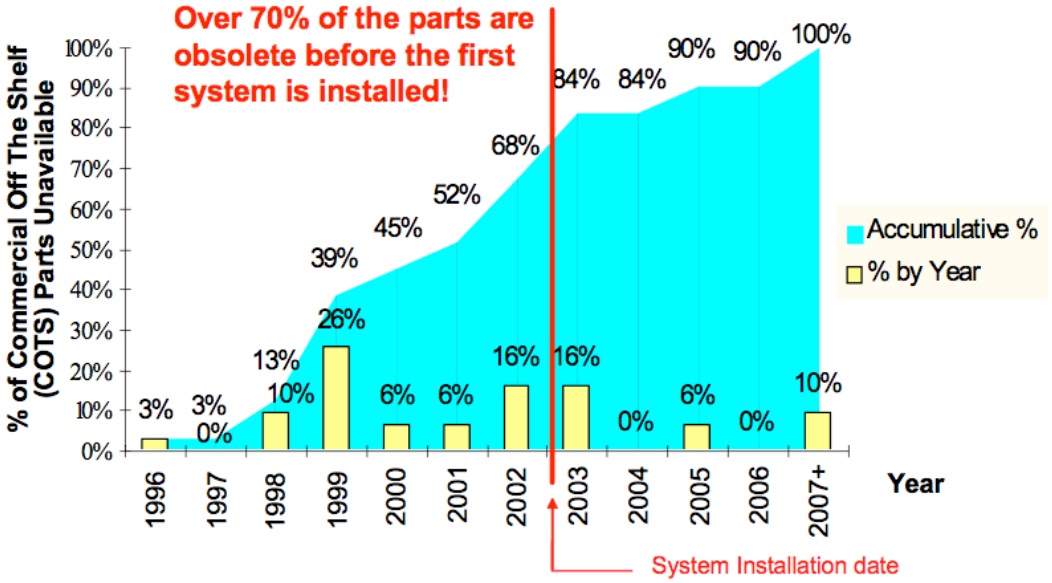


Figure 2: COTS Obsolescence faced by a sonar system during the first decade of its life cycle (from [51])

Historically, DMSMS issues have been treated reactively, by either refreshing the system design to use a non-obsolete component (replacement); bulk purchasing enough instances of the component immediately prior to its obsolescence to last until the system's end of service (EOS) date, and then hoarding the components until they are used (life-of-type/lifetime buy, or LTB); or using some combination of these methods (bridge buy) [52, 26]. More recently, strategic DMSMS management techniques have been developed where the support plan for system includes pre-defined refresh dates at which components that are obsolete or nearing obsolescence are designed out of the system [51, 53]. Each of these methods significantly adds to the total cost of ownership (TCO) of the system, and that cost increases as the number of obsolescence events and number of support years after obsolescence increases.[9]

Additionally, obsolescence can be a symptom of a larger issue – the system uses outdated technology, which means that the system's performance, functionality, and even form factor lag significantly behind the current state-of-the-art. This is an important consideration in commercial and consumer systems for remaining competitive and retaining market share; it may also be crucial in defense applications, because the technology used is both Mission Critical and Safety Critical issue; any enhancement could mean the difference between victory and defeat [8, 25]. In order to keep system technology current, it is necessary to perform frequent design

[9] The viability of each of these options depends on their relative costs, which are effected by a number of factors. As a rule of thumb, redesign is good in situations when qualification costs are low, holding costs are high, and when the EOS date is far away (or may be pushed back with little notice). LTB is good for situations when qualification costs are high, holding costs are low, and when the EOS date is near and unlikely to change.

refreshes. However, design refreshes are an expensive, time-intensive process. In some cases, a design refresh also requires a partial or complete requalification or recertification of the system to ensure it meets the requirements for performance and stability. The use of a modular open systems approach (MOSA) is seen as a way to decrease both the time and cost of design refreshes [8, 54, 2].

1.4.9: Reduced Risk

Without an open standard, there is the danger that the enterprise can choose to use and invest in a technology that may become marginalized if the rest of the market moves in a different direction. When this occurs, the technology will quickly become obsolete [35]. Open standards, particularly those that are well-established, hold a large market share, and are not dominated by alternatives, are assured some amount of longevity. This allows an enterprise to invest in infrastructure and any required training with less risk. Additionally, by exploiting previously used, established components, many development risks may be reduced [47, 54].

1.4.10: Reduced Development Time

The use of open standards leads to reduced development time because it enables the use of unified platforms, increasing the efficiency of research and development [40]. Reuse of components and subsystems from previous systems saves the time (and cost) associated with developing those components. This was seen in the creation of the USS Jimmy Carter Multi-Mission Platform and the SSGN Class of

submarines, both of which reused considerable portions of the Virginia Class submarine design [40]. This benefit is also seen when using COTS components – even if the enterprise doesn't have first-hand experience with them, they are able to integrate them into the system without designing them from scratch.

1.4.11: Enterprise Benefits

As noted above, interoperability and reuse allow an enterprise to repurpose components and subsystems designed for previous systems to increase design and maintenance efficiency, interoperability, and innovation. Beyond this, using an open systems approach across several projects allows an enterprise to increase its return on investment by allowing several projects to benefit from the same infrastructure. For example, the Army Materiel Command found that by consolidating all standards information into a single library, the quality and availability of the information could be improved while one third, or more than $1 million, of the costs could be avoided annually [55].

1.4.12: Increased Costs

Increased cost due to openness may occur if the regulations governing the system's support prevent the full benefits of openness from being realized. For example, using open systems and implementing a faster refresh cycle may not be practical in situations where recertification is expensive or partial recertification is not possible. Historically, this was sometimes the case for safety critical applications,

including military systems and nuclear plants. This must be addressed by appropriate regulatory changes. Additionally, while using commercial components removes much of the design and support burden, it also decreases the level of control the enterprise has over its specification and production. The enterprise may not be notified of minor product or process modifications, which may result in regulatory or certification issues.

Use of existing (reused or COTS) components may also add cost. If a 'perfect' component does not already exist, it may be necessary to use a more expensive component with higher performance or greater functionality than required. This extra functionality could increase the cost and effective failure rate of the component. In other cases, it may be necessary to modify the COTS or reused component for use in the new environment. Possible modifications include physical modifications to a hardware package so it fits in the required space; uprating hardware components for operation beyond the manufacturer's specifications [56]; burning-in a component to remove "infant mortality" failures and improve reliability [57]; and functional changes to software, either by altering the component itself, or through the use of a "wrapper" or API (Application Programming Interface).

Other costs associated with implementing open designs include the costs of building and maintaining a component library for the product to be reused from, and from encouraging and enforcing the use of appropriately open design paradigms. Finally, while use of openness helps to prevent component obsolescence from occurring, and allowing for more affordable handling of obsolescence when it does occur, there is still a risk of standard obsolescence. If the enterprise selects the

"wrong" standard, and the market shifts away from the standard selected, significant costs must be incurred, not just to replace the standard, but to replace all the components which use the standard as well. These costs may outweigh all the other benefits gleaned from using open systems. Unlike the situation when using the enterprise's own proprietary standard, the components and standard may become completely unsupportable, even by the enterprise or a third party, due to lack of access to the necessary IP. This risk may be mitigated by careful selection of the standards used performing a "market watch" to track market trends and be able to better predict obsolescence events or by only using completely open standards. However, great care should be taken in selecting standards to be used. In some scenarios, *openness may not be the best option*. Though open standards should be favored over proprietary ones, the final decision must be made based on maturity, acceptance, and the ability to adapt to meet future needs [6].

Knowledge of the principles and applications of MOSA allows for an understanding of the theoretical situations in which it is most valuable. However, it is also important to be able to ascertain the exact value of an open approach, and differentiate between situations when the use of open systems is beneficial, and situations in which it is detrimental. To answer this question, it must be possible to calculate the investment and the return associated with implementing an open design.

1.5: Measuring Openness and Life Cycle Costs: Existing Work

While many previous efforts have been made to measure openness, almost all of these efforts make the implicit assumption that openness is always beneficial. Additionally, most rely on highly qualitative analysis of a system, which results in low accuracy and poor repeatability. Others are highly specialized tools, only applicable in specific situations, and not valid for more general cases. Finally, the results are often given as an intangible "openness score", which leaves many questions unresolved. The metric can be used to tell which of two systems is more open, but does not provide enough information or resolution to make a business case. How much extra investment is necessary to achieve a higher level of openness? What is the expected return on this investment (ROI)? Should further openness be pursued, or is the current level sufficient? Does the number of fielded systems affect this return? Is there a minimum or maximum length of time that the systems need to be supported in order for openness to pay off? If so, is the result sensitive to life extension?[10] These questions, largely unresolved, are vital to any enterprise considering an open approach.

[10] A life extension occurs when the end of support date is pushed back, and the system must be supported for more years than originally anticipated. Life extension may occur with only a few years of advance warning. This may result in shortages of obsolete (stockpiled) components.

1.5.1: MOSA and MOSA PART

The Modular Open Systems Approach (MOSA) was initially developed by the Open Systems Joint Task Force (OSJTF), which was established by the Department of Defense (DoD) Office of the Undersecretary of Defense for Acquisition, Technology, and Logistics (OUSD (AT&L)). MOSA is a combined business and engineering strategy that calls for the use of widely supported commercial or open standards to be used when developing a new system or updating an existing one. In addition, the enterprise should design for change by using modularity and functional division. This enables spiral development [6].

The OSJTF suggested that enterprises apply MOSA using a five-part strategy [6, 17]:

1. Establish an enabling environment.

   This refers to setting a tone, at the enterprise level, to support and encourage the use of open systems, and creating a MOSA roadmap. This could mean requiring a certain level of openness in designs, training managers and designers about MOSA principles, performing market research, and maintaining a component library. The enterprise should also actively work to remove any barriers to openness.

2. Employ modular design.

   All designs should use a high level of functional division so that any design changes are compartmentalized. This prepares for future design changes, which are largely unavoidable.

3. Designate key interfaces.

   Designating key interfaces allows the enterprise to focus attention strategically, and concentrate on the most volatile, unreliable, or fundamental interfaces.

4. Use open standards for key interfaces.

   While use of open standards may be valuable in general, their use for the key interfaces is vital to ensuring interoperability and interchangeability of components.

5. Certify conformance.

   In addition to encouraging the use of MOSA, it is important to verify that the final design is, in fact, open. This includes confirming that the standards and COTS components selected are interchangeable and not vendor specific.

To certify that both the system and enterprise have properly implemented MOSA principles, the OSJTF developed the MOSA Program Assessment and Rating Tool (PART). The MOSA PART was adapted from the Office of Management and Budget (OMB) PART questionnaire, which was created to evaluate and improve the performance of a wide array of programs across the federal government.

The MOSA PART is a series of questions divided into two sections, business indicators (11 questions) and technical indicators (13 questions). Each question asks about the extent to which a certain principle, ability, or practice is used. The questions, included in Appendix A, are answered by 1) noting if the principle, ability, or practice is planned, already achieved, or not applicable; 2) selecting the extent to

which the principle, ability, or practice has been implemented (none, little extent, moderate extent, or large extent); 3) providing a rationale or explanation for that selection; and 4) supplying any supporting evidence or data, as appropriate. Once the questionnaire has been completed, an assessment of MOSA implementation is given, with each of the five principles (enabling environment, modular design, key interfaces, open standards, and conformance) given a score between 0 and 100%. A combined rating is also given. Scores between 80% and 100% are considered "exemplary", between 60% and 80% "satisfactory", and between 40% and 60% "marginal". Scores below 40% are considered "unsatisfactory".

While the MOSA PART can be used to quickly give an approximation to a system's openness, it lacks guidance on how to accurately evaluate the principles and practices it intends to measure. It is subjective and qualitative, and the final results may be highly dependent on the optimism with which the survey is completed. For example, consider the question "to what extent has the program designated key interfaces?" The tool and its documentation give no instruction on what constitutes a "moderate" level of designating key interfaces as opposed to "little". Depending on the responder's experience, outlook, expectation, or mood, the same evidence could be used to support either response, demonstrating a complete lack of inter-rater reliability. This is true of all 24 questions, and is no small issue – answering "little extent" for all questions results in a score of approximately 33% (unsatisfactory), while answering "moderate" across the board gives a score of about 67% (satisfactory). A borderline system could fall anywhere on this range, with the system's final score effected more by the person or group filling the survey than by

the system itself. This lack of repeatability and precision calls the validity and accuracy of the analysis into question as well.

The openness calculation itself is problematic. Each question is theoretically weighted equally, on a linear scale – a value of 1/3, 2/3, or 1, for little, moderate, or large extents, respectively. Each of the questions is correlated with a different one of the five principles, and the results of that question group averaged to give the score for that principle, with "planned" and "achieved" scores tabulated separately. The five principle scores are then averaged to give the combined rating. However, each question can be marked as "not applicable", removing it from the calculations. Although in theory each question is weighted equally, in practice some questions hold more sway than others. The eleven business indicator questions are averaged to obtain the "enabling environment" score, while only two questions ("to what extent has the criteria for designating key interfaces been established?" and "to what extent has the program designated key interfaces?") are used to calculate the "key interfaces" score. If one of these questions is marked as not applicable, due to lack of evidence, for example, the other question becomes the sole basis for the "key interfaces" score. Furthermore, that single question's answer weighting increases such that it becomes 20% of the combined score. In comparison, questions in the business indicator are only weighted as 1.8% of the combined score.

Finally, the assessment's rough output, satisfactory or not, tells us nothing about the investment already committed to openness, or the costs to attain a higher MOSA score. Though it can measure conformance to the five MOSA principles, it

has no capacity whatsoever to capture the cost avoidances associated with using a MOSA approach.

1.5.2: OAAM and OAAT

The Naval Open Architecture Enterprise Team (OAET) initially developed the Open Architecture Assessment Model (OAAM) in 2005 to "define, measure, and illustrate the relative levels of openness" [17]. Like MOSA PART, the OAAM, described in Appendix B, measures the openness of a system using both business and technical characteristics. The resulting business and technical scores, a value between 0 and 4 taken from the description that best matches with the system's situation, are then plotted to attain the overall openness characterization, a value of low, medium or high. The OAAM is simple and straightforward, but lacks depth and resolution. The OAET therefore created the Open Architecture Assessment Tool (OAAT) to expand and build upon the OAAM. Initially, the OAAT took cues from MOSA PART, using a set of questions to assess the openness of a system. The OAAT was then revised to incorporate the MOSA PART questionnaire itself.

Like MOSA PART, the OAAT uses a series of questions to evaluate an enterprise and system on both business and technical aspects. The business section, called the programmatic section by the OAAT, consists of two sections. The first section consists of eleven questions that are taken directly from the MOSA PART's business indicators. The second section, with thirteen questions, covers OAAT specific items.

In addition to the MOSA PART technical indicators (thirteen questions), the OAAT technical section contains questions designed to align with the five goals of using an open systems approach: interoperability (six questions), maintainability (two questions), extensibility (three questions), composability (two questions), and reusability (four questions). All of these questions are detailed in Appendix A.

The OAAT scoring algorithm is similar to, but more advanced than, that used for MOSA PART. Each response is given a score between zero and five. When the scores are averaged, however, not all questions are given equal weight. Important questions are designated as "key," "litmus," or both. "Key" questions are weighted between 3 and 4 times stronger than other questions, and an answer is required (the "not applicable" response is disabled). "Litmus" questions can limit the maximum score a system can receive, even if the responses to all other questions would suggest a higher score. This makes the OAAT less susceptible to large variations in results due only one factor. It also helps to ensure that results are a valid representation of reality, and are not misrepresented because of the answers to less important questions. Additionally, the OAAT specific questions come with guidelines for each question that define approximate values for "limited extent", "some extent", "extensively", and "very high extent". This greatly improves inter-rater reliability, and improves the precision and repeatability of the results. However, these methodological improvements are only applied to the questions unique to OAAT. The questions taken from MOSA PART do not have "key" or "litmus" designation, and as before, the evaluator is given no guidance as to the definitions of "little," "moderate," or "large" extent.

Like MOSA PART, OAAT cannot account for the cost and investment associated with implementing and using an open systems approach, and is unable to measure the value of the benefits thereby attained. It cannot, therefore, be used to make a business case for the continued or expanded use of openness.

1.5.3: AFRL/RYM Metrics Working Group

In 2011 and 2012 there was collaboration between a number of parties, including the Air Force Research Laboratory's (AFRL) RYM subgroup, MacAulay-Brown, General Dynamics, and Lockheed Martin, to develop a set of metrics to evaluate and judge the openness of an architecture. Focus was given to selecting metrics that were broad enough to a general case, and quantifiable, so that the measurement would be highly repeatable and reliable.

The result of their efforts was called the MOSA Metrics Calculator [58]. It asks a series of simple questions, which are given in MOSA Metrics Calculator, about each component in an architecture. The results are essentially averaged based on which of the five MOSA principles or tenets they are applicable to. The results are then presented on a radar plot showing how well each of these tenets is attained. A sample plot is shown below in Figure 3.

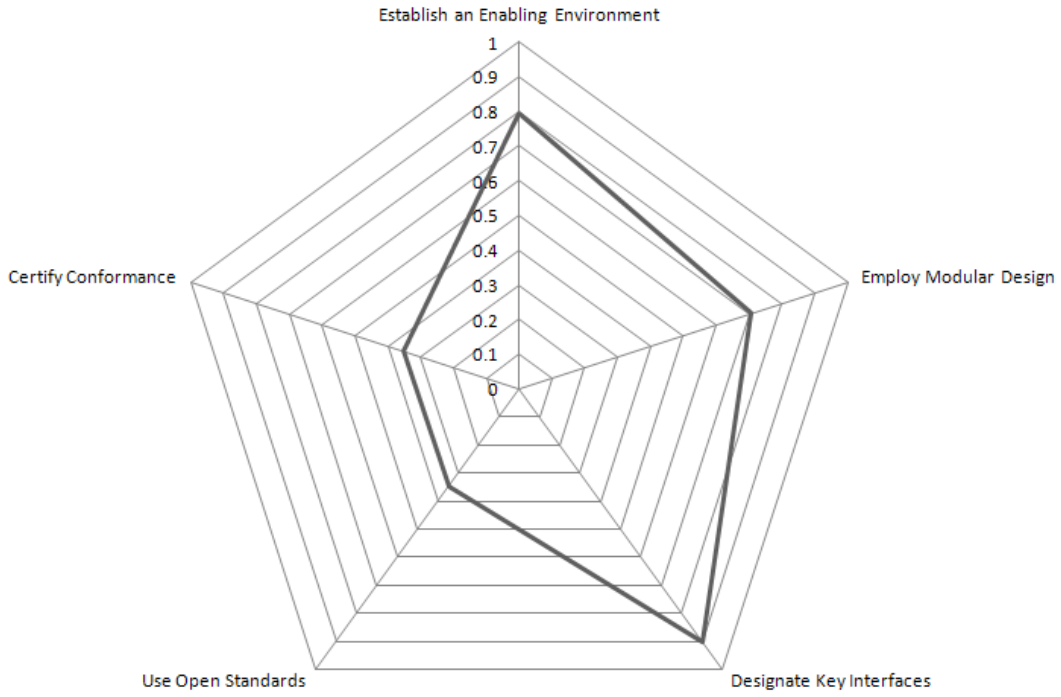


Figure 3: MOSA Tenet Scoring Chart. Example results from the MOSA Metrics Calculator [58]

The MOSA Metrics Calculator improves upon the MOSA questionnaire by ensuring that all metrics are quantifiable, and by using a calculation methodology that is less susceptible to random fluctuation. However, like the previous measurement tools, it cannot be used to make a business case for previous or continued use of an open systems approach.

1.5.4: Open PDQ MOSA Openness Metric

Another approach to measuring openness comes from Peter Henderson, of PMH Systems and the University of Southampton. In several papers (and parts thereof) made available on his website, he goes through the definitions of MOSA and

openness, lists the benefits of and hindrances to attaining openness, and gives guidelines for architects or project managers who hope to implement an OSA. In making a case for open systems architectures [58], he uses a stochastic model to demonstrate that the development cost of a system or module can be reduced significantly by increasing its openness coefficient. The same is true of development time. Unlike many other analyses, he is able to give concrete estimates of the cost and time reductions due to the system's openness. For example, increasing the openness coefficient from 0.2 to 0.4 avoids almost two thirds of the development cost [58]. In this case, the openness coefficient is defines as the fraction of interfaces that use open standards. The results of his model show that development time and cost decrease as openness increase. Further, it demonstrates the decreasing value of marginal openness – more is to be gained by increasing from an openness score of 0.2 to 0.3 than by increasing from a score of 0.6 to 0.8. In other words, the more closed a system is, the easier it is to benefit from implementation of an open systems approach.

Unfortunately, the details of the model are vague. Henderson notes that the model is unverified, and still in development – the significance of the model presented is not the specific values presented, but in the trends and implications, which hold true for a range of input parameters [58].

Henderson's work is valuable, as it uses an easily quantified metric to define openness. The model then translates this openness measure into an approximate benefit. Though the costs that must be invested to achieve a higher level of openness are unknown, this model begins to formulate a business case for the use of openness.

However, the model has a number of shortfalls. First, the openness metric is very basic, and cannot distinguish between standards with different levels of openness. In a more recent, still uncompleted work, Henderson defines a "fuzzy measure of openness" [59], in which the score for a given interface that is dependent on the number of companies that produce the interface and the number of companies that consume it. The number of producers and consumers are each rated on a high/medium/low scale, resulting in seven levels of openness – very low, low, medium low, medium, medium high, high, and very high. Combining these openness measures would improve the resolution and therefore the accuracy of his model.

A second issue with Henderson's model is its very limited scope. The model only covers the design phase. This ignores significant costs and avoidances later in the life cycle, such as those incurred during operation and support, and design refresh or technology insertion. While the benefits at design would be similar to those during refresh, the trend during the support phase would be reversed – higher openness means higher volatility, which increases support costs. As noted previously, it also leaves out the costs incurred to achieve a higher level of openness. Without these costs, the results of Henderson's model are highly misleading. The plots produced by his model suggest that cost and time are both decreasing functions of openness. This would mean that no matter how open the system is, it stands to benefit from increasing the openness. In actuality, this is not the case. Decreasing marginal openness suggests that if one-step increases in openness require a constant cost

investment,[11] the added investment will at some point outweigh the additional benefit. A more complete and comprehensive model is needed.

1.5.5: The COCOMO Family of Models: COCOTS and COTS-LIMO

Though our interest is in the broader topic of openness, a number of cost models have been developed to estimate the costs associated with developing and supporting COTS-based systems (CBS). These models, which only cover software, were created in response to the recent popularity of using CBS to try and reduce the time and cost to develop large software systems [60, 61]. Though like the PDQ model, some of them focus on integration costs alone [62], these models are valuable because they are able to translate usage of openness, or more specifically, the use of COTS software, into estimated costs.

Some of the best developed models are based on work by Christopher Abts and Barry Boehm at the University of Southern California. Boehm is well known for his work on the Constructive Cost Model, or COCOMO. COCMO, which was originally published in 1981, and updated in 2000,[12] is an empirically-based parametric model that can be used to estimate the time, effort, and cost associated with developing a software system. It has several sub-models that allow for accurate

---

[11] Investment here includes any added costs due to openness at any point in the module's life cycle, not just before or during design. It is possible that a marginal increase in openness would require increasing (not constant) levels of investment.

[12] The initial version of COCOMO will be referred to as COCOMO 81. The term COCOMO by itself is used to refer to the updated version, also called COCOMO II. Preliminary versions of COCOMO II were available as early as 1995.

assessment in different parts of the design process. These sub-models are called the Applications Composition, Early Design, and Post-architecture model, with the latter being the most detailed, calibrated, and verified of the three. However, the COCOMO model is explicitly designed for "traditional" system designs, and cannot be used in situations when COTS software is utilized [60].

Since the release of COCOMO, a number of extensions and special applications of the model have been published, including the USC-CSE COTS Integration Cost Calculator[63] (CICC). The CICC model was then expanded to create the Constructive COTS Cost Model, or COCOTS [60]. Like COCOMO, CICC and COCOTS make their estimations based on the size of the software, measured in thousands of lines of source code,[13] or KSLOC, and user input about the type of project and its rating on various other attributes, or "cost drivers". CICC and COCOTS can be used to estimate the costs incurred to integrate a COTS software component into a larger system during the design phase, but cannot be extended to other life cycle phases or to hardware [63]. Even without these later phases, Abts found evidence that the conventional wisdom of "more COTS is always better" was incorrect. Abts aptly explains, "'not building' [does] not mean 'not developing', only developing *differently*" [13]. When using a large number of COTS components, significant effort must be expended to ensure they are all compatible, fulfill the functional requirements, and to integrate them into the larger system. COTS components are more volatile, subject to frequent updates and shorter procurement

[13] A function point based analysis can be used as well, though a language table is used to convert the number of function points to approximate KSLOC.

lives. In general, increasing the number of COTS components increases risk and volatility, meaning that in some situations, a COTS-based system could have a shorter or more expensive life than a comparable proprietary system.

Abts approached this issue by presenting the COTS Lifespan Model (COTS-LIMO), where he argued that the important metric was not high use of COTS components, but high COTS Functional Density (CFD) [13]. In other words, the amount of system functionality provided by each COTS component should be increased, while the number of actual COTS components kept to a minimum. Because of the volatility of COTS components, Abts' model suggests there is an equilibrium value for number of COTS components in a system. Below that equilibrium value increasing the CFD increases cost avoidance. Above the equilibrium value, high volatility and lack of supply line control take effect, and the system rapidly becomes unsupportable (

Figure 4). Unfortunately, though others have found data to support this proposition [12], the COTS-LIMO model was never completed.

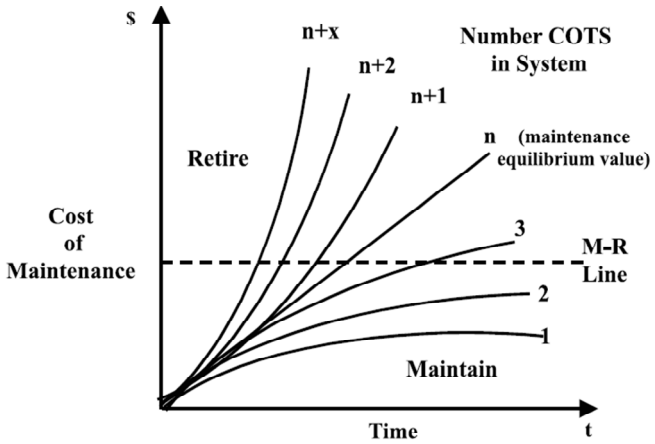


Figure 4: The COTS-LIMO hypothesis of COTS-based systems (from [13])

1.5.6: SAIC Models

Three models appear in the literature attributed to the Science Applications International Corporation (SAIC). The first, "An Economic Analysis Model for Determining Custom versus Commercial Software," was developed by Karpowich, Sander, and Verge [60, 62]. It focuses on the costs associated with integrating a COTS software component, which it gives as [60]:

$$Cost_{Integration} = Cost_{Licensing}\, N_{Licenses} + Cost_{Training} + Cost_{Glue\ Code}$$

Equation 1

where:

$Cost_{Integration}$ is the cost to integrate the component

$Cost_{Licensing}$ is the cost of the COTS component license

$N_{Licenses}$ is the number of COTS component licenses required

$Cost_{Training}$ is the cost to train the end users of the component

$Cost_{Glue\ Code}$ is the cost to create the interface or glue code

The usefulness of this model is limited because it only covers design and integration, and only covers software. Additionally, it leaves $Cost_{Glue\ Code}$ as a given value, when determining how to calculate it is of significant interest.

The second SAIC model is attributed to Stutzke [60]. Though this model was never completed, it proposed the following formula to measure the cost impacts due to COTS volatility:

$$Cost_{Volatility} = (Volatility)(Coupling)(Size)(Cost_{Screening} + Cost_{Changes})$$

Equation 2

where:

$Cost_{Volatility}$ is the added cost associated with the volatility of the component

$Volatility$ is the number of expected releases of the COTS component during the life of the system

$Coupling$ is the number of other components with which the COTS component interacts

$Size$ is the effective size of the component's interface, based on the number of procedures or functions and number of arguments used to interact with it

$Cost_{Screening}$ is the cost of screening all of the components (the COTS component, and all component affected) to ascertain the effects of the release of a new version of the COTS component

$Cost_{Changes}$ is the cost of implementing the changes to effected components

This model is limited because it only deals with software, and only addresses volatility. However, it is still useful because much of the added cost incurred by using COTS components stems from their volatility.

The third SAIC model, also by Stutzke [64], and an expansion of the previous model, also aims to quantify the costs associated with COTS volatility. In addition to the added effort needed to update interfaces after the release of a newer version of a component, which was included in the previous model, this model accounts for the efforts and costs associated with unneeded functionality, the risk of incompatibility of

the new component, and the risk of elimination of required functionality. The model argues that the effort required to integrate a new version of a component is given by [64]:

$$Effort = (L)\left(E_{Analyze}\right) + (M)(E_{Modify}) + \left(\frac{M(M-1)}{2} + M(N-M)\right)E_{Test}$$

Equation 3

where:

L is the number of links the component has with other components

$E_{Analyze}$ is the average effort to analyze a link to see if it needs modification

M is the number of links that need to be modified ($M \leq L$)

E_{Modify} is the average effort required to modify the effected link

N is the number of components, custom and COTS, in the system

E_{Test} is the average effort to test the modified linkage.

The model then makes several simplifying assumptions to reduce the number of variables, and the complexity of calculations. However, the more complex formulation provides valuable insight into what costs may be expected due to the volatility of COTS components, and how to model them.

1.5.7: Loral Federal Systems Model

While he was at Loral Federal Systems, Tim Ellis helped develop a COTS software integration cost model based on Loral's database of historical information. A key difference between this and other methodologies is that the data on past

experiences led Ellis to reject models based on SLOC, which he found led to inaccurate results [61]. Ellis's model is instead based solely on function point analysis. Only new function points are counted – function points supplied by COTS software is omitted. Instead, the cost of integrating COTS software is estimated based on [61]:

1) The number of COTS components
2) The number of interfaces to be developed (COTS and non-COTS)
3) The percentage of requirements that are fulfilled by COTS components
4) A productivity factor based on the language used and staff experience
5) A set of 17 COTS cost drivers:
    a. Product maturity
    b. Vendor maturity
    c. Configurability/customization
    d. Installation ease
    e. East to upgrade
    f. Vendor cooperation
    g. Product support services
    h. Product support quality
    i. Quality of documentation for user, administrator, and installation
    j. Ease of use for end user
    k. Ease of use for administrator
    l. Type and quality of training available for administrator and end user
    m. Administrative effort to maintain
    n. Portability between platforms
    o. Previous Product Experience
    p. Expected Release Frequency
    q. Application or System COTS package

The number of components and interfaces to be developed is used to estimate the number of work units required for the project, taking into consideration the

expected complexity, number of function points, productivity factor, and the cost drivers. Each of the 17 cost drivers is weighted on an individual scale to give a cost factor, which is used to modify the labor estimate. The exact equations used in the calculation are proprietary, but the model was calibrated using six internal projects, and was able to predict those cases with a relatively high accuracy. Its accuracy for external projects is unavailable.

1.5.8: SoCoEMo-COTS

The Software Cost Estimation Model for COTS, or SoCoEMo-COTS, is an economic model for COTS Based Development that aims to produce an estimate of the ROI that can be expected when investing in COTS software components. This is done by defining investment costs (incurred in the first year), and periodic costs and benefits (incurred in all years after the first year) [65].

Investments include a domain analysis cost, and the costs to develop any necessary reuse infrastructure. For each COTS component researched, there are the costs to assess, select, and certify the component for possible use, and the cost to insert the component into the component library. For each COTS component actually used, the investment costs include tailoring the COTS component for use in the specific application, developing any necessary glue code to integrate the COTS component, extra effort required for COTS due to their volatility. Investment also includes the costs to develop any required proprietary components.

Periodic costs include the costs to assess, select, and certify any new COTS components, and add them to the library, costs to reassess new releases of COTS components already in the library, and the costs to maintain the component library. Other periodic costs include: re-tailoring of COTS components that have a new version released; maintaining and updating glue code, including any extra effort due to COTS volatility; and the cost to maintain proprietary components. The primary periodic benefits are the maintenance and service costs avoided by using COTS components, since these are provided by the COTS manufacturer.

The model as presented here[14] provides a helpful insight into the benefits that can be attained not just by using COTS and open architecture for one project, but how an enterprise can increase those benefits by amortizing the initial investment and basic support costs across several projects. Unfortunately, the SoCoEMo-COTS model does not include any insight on how to calculate these costs – all of the cost inputs are to be "determined by expert judgment" [65].

1.5.9: RI3

A final model of note is the Risk Identification: Integration & Ilities, or the RI3 model. Unlike the other models presented, this model does not aim to measure openness or life cycle costs. Instead, it focuses on the risks associated with using new,

[14] The paper presenting the SoCoEMo-COTS model organizes the listed costs and equations in a significantly different manner, dividing each of investment cost, recurring cost, and recurring benefits into three components: domain, application, and corporate. While their general approach is useful, their specific implementation, and presentation thereof, is poorly documented and explained. For this reason, a simplified, restructured version of their model is presented.

unproven technologies. The “ilities” are stability, complexity, reliability, maintainability, integrability, and testability. Use of unproven COTS components increases the risks of penalties and delays due to changes, unsolved bugs, and availability issues. RI3 assesses these risks for level of consequence and likelihood of occurrence based on responses to a questionnaire. The tool then demonstrates which components or modules are at the highest risk, and the cause of those risks, suggesting a best approach to reduce that risk.

It is important to be cognizant of these risks, especially when trying to implement a system using COTS components. Reused and proprietary components will have lower risks than COTS components which receive major updates more frequently. These risks are related to the Technology Readiness Level (TRL) of the component. Components with low TRLs have higher risk. One of the benefits of using open and COTS-based systems is the ability to try and use newer technologies (lower TRL components) that have more and better functionality. However, using such components exposes the system to considerable risk of delays and penalties due to incompatibilities, unreliability, and unmaintainability.

1.6: Problem Statement

The use of a modular, open systems approach has the potential to decrease the total life cycle cost of complex systems. It can potentially simplify and accelerate system design, transfer many support responsibilities and costs to component suppliers, avoid obsolescence issues, and ease the insertion of new technologies.

However, there are costs and risks associated with using an open approach. Passing costs and responsibilities to component suppliers removes control over component selection, update schedules, and supply chains. Inserting new, immature technologies can lead to unforeseen compatibility issues. High volatility and frequent required updates may cause operation and maintenance costs to grow uncontrollably, and make the system unsupportable. *In order to decide if openness should be pursued, and to what degree, a quantitative model that can measure both the costs and benefits associated with a system's openness is needed.*

Though several models, such as MOSA PART and OAAT, have been proposed in an attempt to measure openness, they rely on qualitative measures, and tend to be very subjective. They also have no ability to measure the effects that openness has on life cycle cost. Other models, including COCOTS and SAIC, are able to estimate the life cycle cost associated with a system, and even take into account some of the effects of using COTS components, however, none of these models provide a complete picture of the situation. The Loral model only predicts the costs due to COTS volatility. Some models, like COCOTS and SAIC, only cover COTS integration costs, and cannot be used to analyze the rest of the life cycle. Other models, like COTS-LIMO and SoCoEMo-COTS, have been proposed, but not developed. None of these models provide a complete picture of openness, from proprietary to COTS to fully open. *There is no model that can quantify all the cost affects of increasing the openness of a system over the course of its complete life cycle.*

Many enterprises have assumed that increased openness is always beneficial. Evidence exists to demonstrate that this is not the case. While the use of an open approach can be beneficial, there are risks involved that must be taken into account. *The model developed in this thesis provides an enterprise with insight into the value of an open systems approach, and a cogent business case for choosing whether or not to invest in increased openness.*

1.7: Research Tasks

The following steps are necessary to create the proposed model:

1.7.1: Task 1

Identify relevant Openness Metrics and Cost Drivers, and formulate the connections between the two. This will be used to create a complete picture of the scope of openness impacts on electronic systems, and to guide model development.

1.7.2: Task 2

Develop a "Cost of Openness" model that calculates the cost impact of openness by comparing a specific system implementation to a reference architecture. The model must be dynamic (time dependent) and accommodate uncertainties in input parameters.

1.7.3: Task 3

Use the model developed in Task 2 to conduct a case study and associated sensitivity analysis to demonstrate the application and usefulness of the model.

Chapter 2: Cost of Openness Model

In order to assess the life cycle cost impacts of openness on electronic systems, the approach shown in Figure 5 has been used.

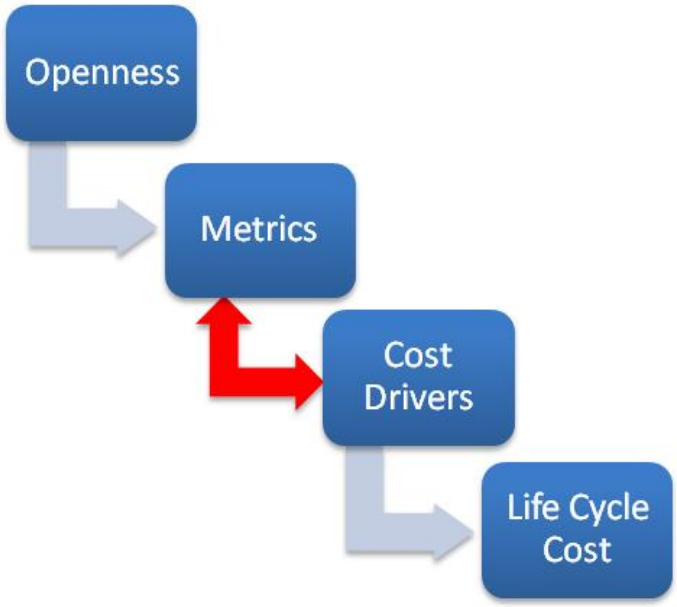


Figure 5: The model is created by measuring openness with relevant metrics, and relating those metrics to cost drivers that can be used to predict life cycle cost.

The approach begins by selecting metrics that can be used to define and measure openness. Concurrently, the cost drivers that can be used to measure and predict life cycle cost are identified. The openness metrics must then be associated with the appropriate cost drivers, so that the connection between openness and life cycle cost may be ascertained.

A list of possible openness metrics was created based on the definition of openness given in Section 1.3, and based on prior attempts at measuring openness including: MOSA PART, OAAT, and the AFRL/RYM Metrics Working Group. This

list was refined by eliminating any metrics that could not be expressed in a clear, quantifiable way. Additionally, metrics were expressed in a general form, so that any component or system can be evaluated using the same metrics. Selected metrics include: market share; volatility; the number of years it has been in the commercial marketplace; enterprise experience; expected procurement life; number of competitors; and accessibility. Procurement and licensing costs are also used. Several other metrics, such as market outlook, number of vendors, and number of applications (a measure of generality and reusability), could also have been selected, but this increases the complexity of the model, and requires more data for simulations to be run. Additionally, some of these overlap with metrics already included. For example, number of vendors may be related to market share and accessibility. All three of the unselected metrics mentioned play a role in the procurement life profile.[15] The possible metrics were further limited to remove metric for which data would never be available.

Similarly, cost drivers were selected based on previous works and experience. The life cycle costs are broken down into six categories, based on when and how the costs are incurred: initial design; non-recurring engineering (NRE) and qualification; recurring production; operation and support; refresh/redesign and implementation of upgrades; and enterprise. In the sections that follow, each of these categories is expanded into a set of cost drivers and then associated with the relevant openness metrics. Since each cost category is calculated separately using a cost-difference

[15] Procurement life is input as a distribution, and is used to predict when the component will become obsolete. This profile should represent increased risk of premature obsolescence due to market outlook, competing standards/components, and other factors.

methodology[16] (Figure 6), cost drivers that are not impacted by the openness of the design can be eliminated.

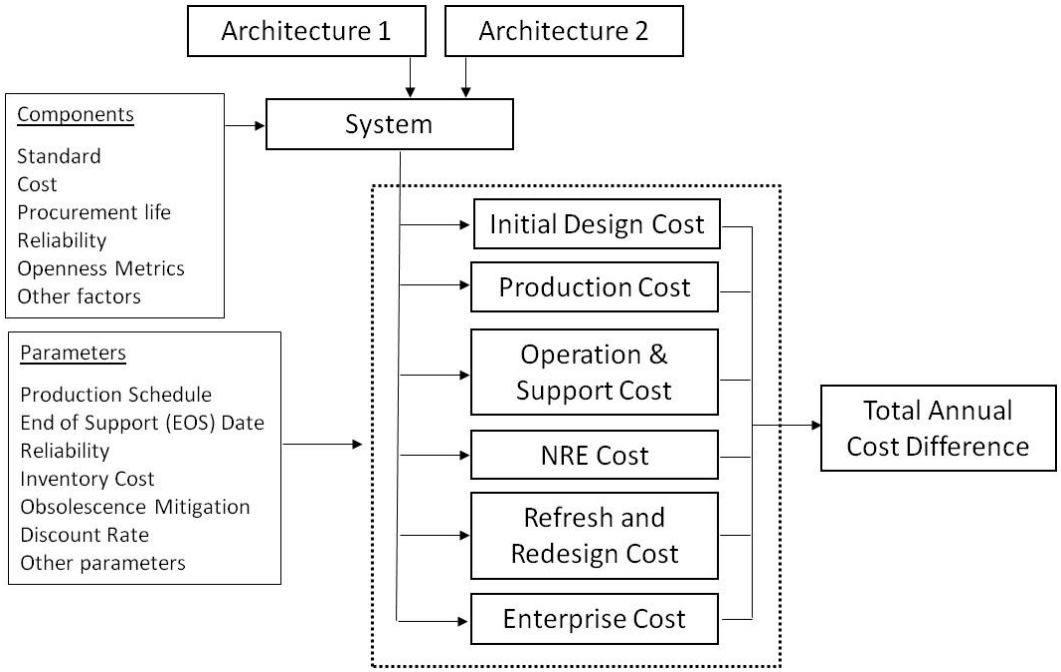


Figure 6: The model takes in two (or more) proposed architectures and various parameters. The model is divided into six distinct cost models, which are then combined to give a total cost difference.

The breakdown of costs into the major categories is given by:

$$C_{Total} = C_{Design} + C_{NRE} + C_{Production} + C_{O\&S} + C_{Refresh} + C_{Enterprise}$$

Equation 4

C_{Total} is the total cost incurred designing, building, operating, and retiring the system. C_{Design} are the costs associated with designing a new system that satisfies a set of requirements, and includes most of the costs incurred before the final design is selected, including the cost of designing the system used, as well as the costs of

[16] We are interested in the difference in cost between cases, not the absolute cost of cases.

partial or other designs considered, but not implemented.[17] Prototyping and any design overhead costs are also included in the design costs. The remaining pre-productions costs fall into the category of C_{NRE}, the Non-Recurring Engineering costs, the most significant of which are the testing and qualification costs to demonstrate that the standards, components, subsystems, and complete system meet the required design parameters for performance, reliability, security, etc. These costs may be significantly lower for enterprises that maintain a library of previously used and qualified components. $C_{Production}$ includes all costs to assemble and ship the system, including procurement, screening and/or burn-in of hardware components, assembly, and any recurring testing costs. All operation and support costs, including those associated with maintenance, sparing, obsolescence mitigation, and lack of availability fall under $C_{O\&S}$. After the system has been in use for some period of time, it may be desirable (or necessary) to update or refresh the system to ensure it is still manufacturable and supportable.[18] These costs, $C_{Refresh}$, are similar to those initially incurred in C_{Design} and C_{NRE}, except that there is a greater opportunity for design reuse, not only of some components or subsystems, but of the overall system architecture as well. The final general category of costs is $C_{Enterprise}$. These are costs that are incurred at the enterprise level, and are shared across all of the different types

[17] Not implemented, i.e., not used for the current system. The ability to repurpose a previously designed subsystem for a new application is an enterprise-level cost avoidance strategy that reduces duplicated and unnecessary effort.

[18] It is often desirable to update a system to add new functionality or improve performance. This is referred to a redesign to distinguish it from a design refresh. While refresh can be modeled with reasonable accuracy, redesign requires parameters and data that are unpredictable, difficult to measure, and, even in the best scenarios, may not exist. See Section 2.1.5 for more details.

of systems created by the enterprise. Examples are costs to maintain a part library, or any support infrastructure that is shared by more than one project.

In the sections that follow, each of the cost categories described above will be further examined and broken into specific cost drivers. The equations for calculating each cost driver are expressed as a function of the openness metrics, system architecture, and other parameters, and are formulated as the cost for one system instance in one year.

To evaluate the total life cycle costs of all instances of the system, the functions must be applied to each system instance in all years that that system instance is fielded. These functions allow the total cost of using a particular system design, open or closed, to be quantified. However, examining the total costs of one system instance does not demonstrate the costs incurred or avoided due to the system's openness. In order to accurately assess the portion of the system costs that result from openness, a comparison between two architectures with the same functionality but different levels of openness will be used. This will both simplify and increase the accuracy of the cost calculations, because costs that are similar for both architectures are omitted from the calculations. This simplification reduces the epistemic uncertainty by limiting the effects caused by unknown factors. Aleatory uncertainty, on the other hand, is not reduced by this procedure.

To handle uncertainty in the analysis, input parameters may be defined as a probability distribution. A Monte Carlo methodology can then be used to evaluate the likely outcome. The distributions are sampled, and the resulting parameter values are used to calculate the total life cycle cost difference between two designs. This process

is repeated for many trials, so that the probability of different outcomes may be obtained.

2.1: Cost Drivers

2.1.1: Design Costs

The costs associated with C_{Design} are given by:

$$C_{Design} = C_{Design\ Infrastructure} + C_{Design\ Creation} + C_{Design\ Implementation}$$

Equation 5

Each of these terms is explained below.

Design Infrastructure

$C_{Design\ Infrastructure}$ are the costs associated with supporting, training, and maintaining a design team, and any other costs associated with the infrastructure required to produce designs and prototypes. This cost is highly enterprise dependent, and could depend on how much of the design is produced in-house as opposed to provided by external entities. However, except for extreme cases, the design infrastructure costs for two systems designed by the same enterprise will be roughly similar.[19] This is particularly true in larger enterprises, when there are always multiple

[19] There are significant cost differences between using a COTS component and developing a proprietary component, but these costs are captured in NRE costs.

types of systems under development simultaneously. As such, the difference between the two values would be negligible, and can be treated as a wash.

Design Creation

$C_{Design\ Creation}$ is the cost of actually designing the system, including defining the functional requirements and other parameters. It is dependent on the size and complexity of the project, the number of functional requirements, and the intended usage environment, among other things. All of the systems involved in the analysis would be approximately the same in these regards, and the cost difference between would be negligible, and could be treated as a wash.

Design Implementation

$C_{Design\ Implementation}$ are the later-stage design costs to finalize and qualify the design. This includes defining component interconnections, finalizing the assembly process, debugging, defining minor modification procedures for procured components, and completing any necessary APIs[20] or glue code. These costs are highly dependent on the architecture of the system, the design strategy, and the specific components used. A system designed from the ground up from mostly proprietary, purpose-built components will likely be more expensive than one formed

[20] Application Programming Interfaces

out of previously designed and reusable components and subsystems, be they COTS or proprietary.

However, open-systems design does not necessarily correlate to lower implementation costs. This is because one of the main factors in implementation cost is how interoperable the components are – a system designed of COTS or reused components may work on paper, but parts may have unintended and unforeseen interactions. Additionally, from a debugging standpoint, in-house proprietary software components are more "open" than their COTS counterparts, because access to and experience dealing with the source code allows for faster localization and resolution of bugs.

$$C_{Design\ Implementation} = \sum_{i=1}^{N_{Components}} Integrate(Component_i)$$

Equation 6

where:

$N_{Components}$ is the total number of components used in the system, counting multiple instances or locations[21] of the same component

$Integrate(Component_i)$ is the cost to integrate the component for use in location i. It depends on the accessibility and standard of the component being integrated.

[21] "Location" is used to mean the niche or slot in the system and functionality that the component is used to fill.

The cost to integrate a hardware component for use is given by:

$$Integrate(Component_i) = BC_{Integrate\ HW}\ CF_{Integrate\ HW}(Component_i, N_{Interfaces})$$

Equation 7

where:

$BC_{Integrate\ HW}$ is the base cost for adapting a hardware component

$N_{Interfaces}$ is the number of other components or subsystems that the component interacts with in location i.

$CF_{Integrate\ HW}(Component_i, N_{Interfaces})$ is the overall cost factor associated with adapting the component for use in location i. It is a function of the number of interfaces, enterprise experience, and the popularity and accessibility of the component, among other factors.

For software cost estimation, many models, including all of the models in the COCOMO family, use lines of source code (SLOC or KSLOC) to estimate a base effort or cost. Other models, like the Loral Federal model, are based on a function point analysis (FPA). KSLOC and FPA based estimation techniques have their own advantages and disadvantages.[22] However, the number of function points and size in KSLOC or a software component are correlated, so conversion between the two is

[22] Counting lines of code is faster and simpler than counting function points, but care must be taken to standardize the counting method – are blank or commented lines counted? The number of lines of code could also depend on the developer who wrote the code. FPA analysis is slower, more expensive, and more subjective than counting lines of code. Additionally, FPA analysis may not properly account for complexity "hidden" in the inner workings of a component [71].

theoretically possible. [23] In this model, effort is estimated using the KSLOC method because it is simpler to calculate. Lines of code will be counted using the same rules used by COCOMO. Use of KSLOC estimation is also beneficial because it allows data from the CICC (and other COCOMO models) to be easily integrated into the model.

The CICC [62, 63] provides a methodology for estimating the costs to integrate COTS software component into a larger system. This model will utilize the following simplification of the CICC:[24]

$$Integrate(Component_i) = BC_{Integrate\ SW}\ (Size(Component_i))^A \sum_{j=1}^{5} CF_j(Component_i)$$

Equation 8

where:

$BC_{Integrate\ SW}$ is the base cost to adapt a software component

$Size(Component_i)$ is the size (in KSLOC) of the component

A is a nonlinear architectural scale factor

[23] The conversion from KSLOC to function points, and vice versa, depends on the language used, among other factors. Typically, KSLOC or FPA is used to estimate the effort required, and that effort is then used to estimate cost. As such, switching between the methodologies necessitates a change in base effort calculation, but should not significantly affect the cost adjustment factors (which may need recalibration), or change the general trends of the results.

[24] The CICC measures 14 different cost drivers, each of which is rated on a scale of very low, low, nominal, high, and very high. These ratings are used to calculate a cost factor to predict the effect the cost driver has on integration costs. 5 of these cost drivers, which are relevant to openness, were incorporated. The remainder were omitted both for simplicity, and because the necessary data may not be available. This is valid because in the CICC model a "nominal" rating always gives a cost factor of 1, ensuring that the model can be used accurately even without complete data.

$CF_j(Component_i)$ is the jth cost factor associated with designing $Component_i$. The five factors are: enterprise experience with the standard, enterprise experience with the component, and the longevity, accessibility, and volatility of the component.

2.1.2: Non-Recurring Engineering Costs

The non-recurring engineering costs are given by:

$$C_{NRE} = C_{Supplier} + C_{Standard} + C_{Component} + C_{System}$$

Equation 9

Supplier NRE

$C_{Supplier}$ is the cost to qualify a supplier or vendor as a source for one or more required components. This cost is assumed to be approximately independent of the supplier, so the total system cost for vendor qualification can expressed as:

$$Cost_{Supplier} = BC_{Supplier\ NRE} \sum_{i=1}^{N_{Suppliers}} CF(Supplier_i)$$

Equation 10

where:

$BC_{Supplier\ NRE}$ is the base cost of qualifying one supplier

$N_{Suppliers}$ is the number of suppliers to be qualified

$CF(Supplier_i)$ is the overall cost factor associated with supplier i, which is primarily a function of the number of projects for which the supplier provides components.[25]

The N_{Sup} depends on the number of different types of components used in the system, the number of sources desired per component,[26] and the ability of certain suppliers to provide more than one necessary component. Use of pre-approved suppliers that have already been qualified by the enterprise as sources for other projects may significantly reduce this cost, especially in larger enterprises that maintain a "library" of approved suppliers. For many cases, the number of new suppliers needed would be similar for the systems being compared.[27]

Standard NRE

$C_{Standard}$ is the cost to assess and select an interface standard for use in the system. While no costs are directly incurred by using one standard or another, each standard used must be evaluated to ensure that it meets the performance, reliability, and security requirements for the system. If the enterprise maintains a library of known standards, these investigative costs are reduced, though the enterprise must

[25] If the supplier provides the enterprise with components for more than project, the cost may be amortized across all of those projects. Alternatively, the full cost may be attributed to the first project, with later projects being able to use the approved vendor without having to pay any qualification fees.
[26] While, in theory, having one source for each component is enough, it is often desirable to have multiple sources to help avoid lead-time delays and increase availability.
[27] This is not always a valid assumption, particularly in cases where the enterprise is trying to decide whether or not to alter their design paradigm. Switching from mostly closed, proprietary designs to open, COTS-based ones, for example, would require qualifying many new suppliers. For large enterprises, however, the cost of this shift would be amortized over many projects or considered a one-time investment, and can so be assumed to play a negligible role in any given system.

pay to maintain and add to the library as necessary. The NRE costs for each standard used can be expressed as:

$$C_{Standard} = BC_{Standard\ NRE} \sum_{i=1}^{n_{Standards}} CF(Standard_i)$$

Equation 11

where:

$BC_{Standard\ NRE}$ is the base cost of qualifying one standard

$n_{Standards}$ is the number of standards to be qualified

$CF(Standard_i)$ is the overall cost factor associated with standard i, which is a function of the complexity, accessibility, longevity, and market share of the standard, among other factors. The number of projects under the enterprise that use the standard also plays a roll.

Component NRE

$C_{Component}$ is the cost to assess, select (or, for proprietary components, design), and certify a component for use in the system. This includes developing any necessary acceptance tests, and the purchase of any testing infrastructure required. Purchase of the technical data package (TDP)[28], if applicable, also falls in this

[28] A technical data package (TDP) is a technical description of the component, and may include (among other items): engineering drawings; specifications; performance, reliability, and quality statistics; and product definition data, which "denotes the totality of data elements required to completely define a product… [including] geometry, topology, relationships, tolerances, attributes, and

category. If the enterprise has previous experience with the component, or component is already in the enterprise's component library, these costs are significantly reduced. The total cost per system for component NRE costs can be expressed as:

$$C_{Component} = \sum_{i=1}^{n_{Components}} \left(BC_{Design}\, CF_{Design}(Component_i) + BC_{Component\ NRE} \sum_{j=1}^{N_i} CF(Component_i, Location_j) \right)$$

Equation 12

where:

$n_{Components}$ is the number of unique components to be qualified

BC_{Design} is the base cost to create a component

$CF_{Design}(Component_i)$ is the overall cost factor associated with designing $Component_i$. It is a function of the enterprise, the standard used, the size of the component, and the number and difficulty of the functional requirements, among other factors.

$BC_{Component\ NRE}$ is the base cost of qualifying the component for use in one location

N_i is the number of instances (locations) of $Component_i$

$CF(Component_i, Location_j)$ is the overall cost factor associated with qualifying $Component_i$ for use in $Location_j$, which is a

---

features necessary to completely define a component part or an assembly of parts for the purpose of design, analysis, manufacture, test, and inspection" [73].

function of the size, accessibility, interface standard, market share, longevity, and volatility of the component. The difficulty of the system's functional requirements, enterprise experience, and the type and extent of any necessary modifications also play a roll. C_{NRE} costs are incurred per component instance, not component type, because if a component is used in more than one location, it must be qualified for use in each of those situations.[29]

System NRE

C_{System} is the cost to qualify the final system design, and develop any necessary outgoing tests to verify it is functioning properly before it is shipped. This cost is a function of the system complexity, number and types of components and interface standards used in the system, required system reliability, and other relevant metrics. This cost can be expressed as:

$$C_{System} = BC_{System\ NRE}\ CF(System)$$

Equation 13

where:

$BC_{System\ NRE}$ is the base cost of qualifying any system

[29] In cases where a subsystem is reused (or, analogously, a component is reused for the same purpose in different locations), the use of the component in that subsystem may only need to be qualified once.

$CF(System)$ is the overall cost factor associated with the system, which is a function of system complexity, the number, types, and accessibilities of the components and interface standards used, enterprise experience with those components and standards, and required system reliability

2.1.3: Production Costs

Production costs are incurred in each year that a new instance of the system is produced. All production costs are incurred at the beginning of the year. The production costs are given by:

$$C_{Production} = C_{Procurement} + C_{Part\ Preparation} + C_{Assembly} + C_{Recurring\ Test}$$

Equation 14

Procurement

$C_{Procurement}$ is the cost to purchase all the components required to build one instance of the system. This can be expressed as:

$$C_{Procurement} = \sum_{i=1}^{N_{HW}} Cost(Component_i) + \sum_{j=1}^{n_{SW}} Cost(Component_j, N_j)$$

Equation 15

where:

N_{HW} is the total number of hardware components in the system, including multiple instances (locations) of the same component, and any consumable components

$Cost(Component_i)$ is the cost of purchasing $Component_i$

n_{SW} is the number of unique software component types in the system

N_j is the number of instances of $Component_j$ in the system.

$Cost(Component_j, N_j)$ is the purchase cost[30] for N_j instances of $Component_j$. The function $Cost(Component_j, N_j)$ is used to reflect the fact that software purchasing costs are often nonlinear or non-continuous functions of the number of instances required.

Consumable components are items such as fuel, fuses, filters, and batteries, or any other component that is used, and then replaced, at a constant rate. Hardware components that have a constant failure rate or that are replaced prior to failure (preventative maintenance) may also be modeled as consumable components.

Part Preparation

$C_{Part\ Preparation}$ is the per-instance cost of acceptance testing components to ensure they meet requirements, as well as the cost of any part modification, uprating,

[30] The purchase cost is a one-time fee incurred when the software is initially procured. Recurring licensing or subscription fees are covered in operation and support.

or burn-in, when applicable. This cost only applies to hardware and consumable components.

$$C_{Part\ Preparation} = \sum_{i=1}^{N_{HW}} Modification(Component_i)$$

Equation 16

where:

N_{HW} is the total number of hardware components in the system, including multiple instances of the same component, and any consumable components

$Modification(Component_i)$ is the cost of modifying, uprating, and burning-in $Component_i$. If components are rejected during the screening process, the costs associated with purchasing, screening, and disposing of rejected components should be included here [66].

Assembly

$C_{Assembly}$ is the cost of assembling one instance of the system from its constituent components. For hardware (and consumable) components, the cost to place a component into the system is assumed to be proportional to the purchase cost. For software, it is assumed to proportional to its size. This can be expressed as:

$$C_{Assembly} = CR_{Install\ HW} \sum_{i=1}^{N_{HW}} Cost(Component_i)\ CF_{Install\ HW}(Component_i)$$
$$+ BC_{Install\ SW} \sum_{j=1}^{N_{SW}} Size(Component_j)\ CF_{Install\ SW}(Component_j)$$

Equation 17

where:

$CR_{Install\ HW}$ is the cost ratio to install hardware. It is used to define the proportion of a component's purchase cost that is incurred for installation. It is dependent on the enterprise.

N_{HW} is the total number of hardware components in the system, including multiple instances of the same component, and any consumable components

$Cost(Component_i)$ is the cost of purchasing hardware $Component_i$

$CF_{Install\ HW}(Component_i)$ is the overall cost factor associated with installing one instance of hardware $Component_i$. It is a function of several factors, including the number of components and different standards affected, the level of coupling, and enterprise experience.

$BC_{Install\ SW}$ is the base cost to install a software component 1 KSLOC in size

N_{SW} is the total number of software components in the system, including multiple instances (locations) of the same component

$Size(Component_j)$ is the size of software $Component_j$, in KSLOC

$CF_{Install\ SW}(Component_j)$ is the overall cost factor associated with installing an instance of the software component $Component_j$. It is a function of the number of components and different standards affected, the level of coupling, and enterprise experience, among other factors.

This model uses KSLOC to estimate the size of any proprietary software or glue code, and calculate the effort and cost associated with that software. Function points could also be used, with appropriate changes to the definition of $Size(Component_j)$ and any relevant cost factors.

Recurring Test

$C_{Recurring\ Test}$ is the cost of verifying that an assembled system is fully operational and ready to be shipped. It is primarily a function of the number of interface standards used the complexity of the system. It may be calculated as:

$$C_{Recurring\ Test} = BC_{Recurring\ Test}\ CF_{Recurring\ Test}(System)$$

Equation 18

where:

$BC_{Recurring\ Test}$ is the base cost of testing one instance of one system. It is dependent on the enterprise, and defined by user input.

$CF_{Recurring\ Test}(System)$ is the overall cost factor associated with testing the system, which is a function of the number and types

of interfaces standards, the number and types of components, and the complexity of the system.

2.1.4: Operation and Support Costs

The costs associated with Operation and Support are:

$$C_{O\&S} = C_{Operation} + C_{Maintenance} + C_{Configuration} + C_{Obsolescence}$$

Equation 19

Each of these terms is explained below.

Operation

$C_{Operation}$ is the annual cost of operating the system. This includes relevant labor costs, [31] and the cost of consumable items such as batteries or fuel, where applicable. This cost driver also covers any annually recurring licensing or subscription fees that are incurred for the use of software or other IP. The costs associated with replacing (hardware) components that have relatively high or consistent failure rates, or that are replaced on regular basis (preventative maintenance) can also be included here.[32]

[31] A system that requires more operators or a higher level of skill/more training to operate will be more expensive to operate each year. If the same amount of labor is required for each instance of the system in each year, the labor may be modeled as a consumable item.

[32] For example, a fuse that fails approximately 7 times per year, or an air filter that must be replaced every four months. These costs may also be placed under the "sparing" section (see later in this section). When appropriate, modeling hardware failures as a consumable item in this fashion is desirable because it simplifies calculations and decreases computational time.

$$C_{Operation} = \sum_{i=1}^{N_{Components}} \Big(Licensing(Component_i) + Demand_i \left(Cost(Component_i) + Modification(Component_i) + Install(Component_i) \right) \Big)$$

Equation 20

where:

$N_{Components}$ is the total number of components in the system, including multiple instances (locations) of the same component

$Licensing(Component_i)$ is the annual licensing or usage fees associated with $Component_i$. When the same component is used in multiple locations, this may be the same for each location, incurred once for all locations together, or different in each location, depending on the licensing terms.[33]

$Demand_i$ is the number of instances of the component required annually for location i. For non-consumable items, $Demand_i$ = 0.

$Cost(Component_i)$ is the cost of purchasing $Component_i$

$Modification(Component_i)$ is the cost of modifying, uprating, and burning-in an instance of $Component_i$. If components are rejected during the screening process, this includes the costs

[33] For some components, including most hardware, $Licensing(Component_j) = 0$. This may occur if no licensing fee is incurred, or if the licensing fee is only incurred for the first instance (location) of the component.

associated with purchasing, screening, and disposing of rejected components [66].

$Install(Component_i)$ is the cost of installing one instance of the component, as given by:

$$Install(Component_i) = CR_{Install\ HW}\ Cost(Component_i)\ CF_{Install\ HW}(Component_i)$$

Equation 21

where:

$CR_{Install\ HW}$ is the cost ratio for installing hardware, which is used to define the proportion of the component's purchase cost that is incurred for installation. It is dependent on the enterprise, and taken as an input.

$CF_{Install\ HW}(Component_i)$ is the overall cost factor associated with installing an instance of hardware component $Component_i$. It is a function of the component's complexity, standard, level of coupling.

Maintenance

$C_{Maintenance}$ is the cost to maintain the system and keep it in working order. For hardware, the maintenance costs include: the costs of performing reactive maintenance, including repairing damaged components; purchasing and installing spares, including the costs associated with ordering, receiving, and holding procured

components; collateral damage mitigation; and the cost of downtime. The cost of repairing a component depends on the component's complexity, failure mode, and the repairer's experience. The amount of life remaining in the component after repair is also an important factor. However, this model will assume that 100% throwaway is used, and that no components are repaired or harvested for future use. Likewise, collateral damage and downtime penalties are assumed to be zero. Using these assumptions, the hardware maintenance costs become:

$$C_{Maintenance\ HW} = \sum_{i=1}^{N_{HW}} \Big(Demand_i \left(Cost(Component_i) + Modification(Component_i) + Install(Component_i) \right) \Big)$$

Equation 22

where:

N_{HW} is the total number of hardware components in the system, including multiple instances of the same component (i.e., the total number of locations)

$Demand_i$ is the number of spares required annually for location i, which is dependent on the component's failure rate

$Cost(Component_i)$ is the cost of purchasing one instance of $Component_i$

$Modification(Component_i)$ is the cost of modifying, uprating, and burning-in an instance of $Component_i$. If components are rejected during the screening process, this includes the costs

associated with purchasing, screening, and disposing of rejected components [66]

$Install(Component_i)$ is the cost of installing one instance of the component, as given above (see $C_{Operation}$)

For software, maintenance includes debugging and providing minor updates as necessary. This cost is a function of the size and complexity of the software, the level of coupling, and maintainer experience. For proprietary software components, all of the code must be maintained by the enterprise, while for COTS and similar software components that are maintained by a third party, the enterprise is only responsible for maintaining the glue code or APIs necessary to integrate the component into the system. In either case, software maintenance costs include the cost of implementing the upgrade – sending the updated code to units in the field. The cost to implement is incurred every time a fielded component is updated. The update frequency depends on the enterprise's strategy, and the urgency of the update. For low priority updates, common schedules include monthly, semi-monthly, and annual updates. High priority updates may be implemented as they become available, which depends on the update schedule and security of the component. The cost for software maintenance in each year can be written:

$$C_{Maintenance\ SW} = \sum_{j=1}^{n_{SW}} \left(Debug(Component_j) + \sum_{k=1}^{N_j} Debug(GlueCode_{j,k}) + Implement(Component_{j,k}) \right)$$

Equation 23

where:

n_{SW} is the number of unique software components in the system. In this case, 'unique' means that modified versions of the same software should be counted separately (with the possible exception of some superficial modifications).

$Debug(Component_j)$ is the cost of debugging software associated with $Component_j$. This is only incurred for proprietary components. For COTS components, debugging is provided by the supplier, and any associated maintenance or subscription fees should be included under operational costs. Debugging costs are a function of the size, accessibility, interface standard, and complexity of the component.

N_j is the number of instances of $Component_j$ in the system

$Debug(GlueCode_{j,k})$ is the cost of debugging the APIs or glue code associated with $Component_j$ in location k.

$Implement(Component_{j,k})$ is the cost of installing the updated $Component_j$ code to location k in a fielded system. It is given by:

$$Implement(Component_{j,k}) = BC_{Implement}\ Size(Component_j)\ Frequency_j\ CF_{Install\ SW}(Component_{j,k})$$

Equation 24

$BC_{Implement}$ is the base cost for implementing a software update.

$Size(Component_j)$ is the size of $Component_j$, in KSLOC

$Frequency_j$ is the expected number of updates per year to $Component_j$

$CF_{Install\ SW}(Component_{j,k})$ is the overall cost factor associated with installing an updated version of $Component_j$ in $Location_k$. It is a function of the size, accessibility, and interface standards used by the component and the components with which it interacts.

For both $Component_j$ and $GlueCode_{j,k}$, the debugging function can be expressed as:

$$Debug(Software) = BC_{Debug\ SW}\ Size(Software)\ CF_{Debug\ SW}(Software)$$

Equation 25

where:

$BC_{Debug\ SW}$ is the base cost to maintain 1 KSLOC of software

$Size(Software)$ is the size of $Software$, in KSLOC

$CF_{Debug\ SW}(Software)$ is the overall cost factor associated with maintaining $Software$. It is a function of the size, stability, and complexity of the software, maintainer experience, and other factors.

Configuration

$C_{Configuration}$ is the cost to track all the different fielded versions of the system. A system version in this case is any unique set of hardware and/or software. Initially, all systems should be identical (though sometimes there are multiple production versions). Over time, however, as hardware components become obsolete and failures are replaced with spares of a newer version of the component, and as software is updated, the number of system configurations grows. For proprietary systems, this growth is relatively slow, but for COTS intensive systems, the high volatility and short procurement lives of the components means that the number of configurations can grow quickly over time, until every fielded system is slightly different from all the others. Configuration tracking is important because it allows error tracking to identify problematic component combinations. It also allows the enterprise to track all the different components in the field, and enables better scheduling and prioritization of preventative maintenance and upgrades.

One way to prevent configuration proliferation is to require fielded systems that experience a failure to be repaired using the same version of the component that they were using previously. Newer versions are only introduced when the entire system is

updated to the most recent iteration of the design. This increases support costs because a larger number of components will need to be stockpiled when obsolescence occurs. However, it reduces the number of possible configurations to the number of design refreshes that have occurred. Additionally, it reduces qualification costs associated with requiring refreshed components to be completely backwards-compatible with all current and previous versions of the components with which it interacts. The configuration costs are given by:

$$C_{Configuration} = BC_{Configuration}\ CF_{Configuration}(System, n_{Configurations})$$

Equation 26

where:

$BC_{Configuration}$ is the base cost to manage one configuration. It is dependent on the enterprise, and taken as an input.

$n_{Configurations}$ is the number of system configurations

$CF_{Configuration}$ is the overall cost factor associated with maintaining multiple configurations of system. It is a function of the system, the number of fielded configurations, and the upgrade and support strategies used by the enterprise.

Obsolescence

$C_{Obsolescence}$ is the cost of mitigating the obsolescence of procured components. The form of mitigation used may depend on the enterprise, the type of component, and the type of obsolescence encountered.

For hardware, one option is to make use of 'secondary sources,' including aftermarket parts, though this increases the risk of counterfeit components significantly. A better option is to use a life-of-type or lifetime buy strategy, wherein all components that will be required between the obsolescence and EOS dates are purchased immediately before the component goes obsolete. An annual holding cost that is proportional to the purchase price of the component and the number of instances held is incurred. When a lifetime buy purchase is to be made, the number of required spares is estimated, and that amount, plus an additional pre-defined buffer, is purchased. A penalty is assessed for component shortages (an under-buy penalty), to simulate the additional cost incurred to obtain the component from a second source. If a component surplus occurs, a separate (smaller) penalty for overbuying is charged for disposal of the unneeded component.[34] Both of these penalties are proportional to the component's initial purchase price.

Another mitigation strategy is to find a not obsolete component that can be used in place of the obsolete one. This can be an expensive, since the new component will have to be qualified and tested (see $C_{Refresh}$). However, it is often a good option for long-term support, because it avoids the costs associated with making an up-front bulk purchase of components, stocking those components for long periods, and overbuying or under-buying the component.

[34] In many cases, this penalty can be omitted, as it will be negligible compared to the purchase and holding costs already assessed. Alternatively, unnecessarily paying to purchase and hold a component in inventory is its own penalty.

In many cases, a combination of these two strategies, called a bridge buy, may also be used. This entails purchasing components to last until a pre-defined refresh date, when the design is refreshed and several components are replaced at once. This involves less logistics and support than performing lifetime buys, while reducing the total refresh cost and number of configurations in the field compared to a refresh-only approach. For both an immediate refresh and bridge buy approach, an upgrade period must also be defined. The upgrade period is the maximum amount of time allowable before all fielded systems are retrofitted, so that they no longer use the obsolete component. Frequently, the upgrade period is equal to or double the refresh period. In the former case, all systems are updated before the next refresh occurs, so at most two fielded configurations are possible. In the latter, systems may "miss" an upgrade (for example, skip from design revision 1 to design revision 3), and at most three configurations are fielded at a time. However, any integer value may be used.[35] The obsolescence mitigation costs, assessed at the end of each year, can be written:

$$C_{Obsolescence\ HW} = \sum_{i=1}^{N_{HW}} Cost(Component_i)\left(N_{i_{Purchase}} + N_{i_{Stock}}\ Holding + N_{i_{Over}}\ Overbuy + N_{i_{Under}}\ Underbuy\right)$$

Equation 27

where:

[35] Very short upgrade periods may be impractical, but long upgrade periods increase the number of fielded configurations and increase support costs.

N_{HW} is the total number of hardware components in the system, including multiple instances (locations) of the same component

$Cost(Component_i)$ is the cost of purchasing $Component_i$

$N_{i_{Purchase}}$ is the number of instances of $Component_i$ purchased to create a stockpile for after the component becomes obsolete[36]

$N_{i_{Stock}}$ is the number of instances of $Component_i$ being held in inventory at the end of the year

$Holding$ is the proportion of the initial purchase cost paid to support one instance of the component in inventory for one year. It is dependent on the enterprise, and taken as an input

$N_{i_{Over}}$ is the number of extra instances of $Component_i$ left over in inventory at the end of the year that must be disposed of

$Overbuy$ is the proportion of the initial purchase cost paid to dispose of one unneeded instance of a component. It is primarily dependent on the enterprise, and taken as an input

$N_{i_{Under}}$ is the number of instances of $Component_i$ that were needed in the current year but were unavailable because the component was obsolete, and no instances remained in the stockpile

$Underbuy$ is the proportion of the initial purchase cost paid to acquire one instance of an obsolete component. It is a function of the enterprise, and taken as an input

---

[36] Incurred in the year immediately prior to obsolescence

In any given year, at most one of $N_{i_{Purchase}}$, $N_{i_{Over}}$, and $N_{i_{Under}}$ will be larger than zero. $N_{i_{Purchase}}$ is only larger than zero in $Component_i$'s last year before obsolescence, when a stockpile ($N_{i_{Stock}}$) is purchased. $N_{i_{Over}}$ and $N_{i_{Under}}$ can only be larger than zero after the component has been obsolete for some amount of time. $N_{i_{Under}}$ larger than zero occurs if the stockpile has been depleted, but additional instances of the component are required. $N_{i_{Over}}$ larger than zero only occurs after a refresh or after the systems' EOS date, when any instances of $Component_i$ remaining in inventory are no longer needed.

This equation implicitly assumes that all component purchases, including stockpiling, occurs at the beginning of the year, and that holding costs are assessed at the end of the year. No holding costs are paid for components used during the current year, regardless of when during the year they are used. Additionally, all costs associated with procuring the component from a second source are assumed to be included in the underbuy penalty. In particular, it is assumed that all counterfeits[37] are detected during acceptance and screening, and so no counterfeit components are actually fielded. The added cost of purchasing, detecting, and rejecting counterfeit components is included in the underbuy penalty [66].

Software obsolescence must be handled differently than hardware obsolescence. Depending on the licensing agreement (no annual licensing fee), it may be possible for the enterprise to continue using the software, unsupported, indefinitely. In other cases (when there is an annual licensing fee), "down licensing"

[37] Counterfeit components include used (salvaged) instances of the correct component that are passed off as new components.

may be required, where the enterprise pays the licensing fees for the newer software version, and is allowed to continue using the old version of the software, unsupported. Using unsupported software is generally not a viable long-term solution. As changes in the surrounding system are made and new bugs in the software are encountered, the software may become unusable.

For longer support periods, the rights to the software's source code can sometimes, but not always, be purchased and maintained by the enterprise, or placed in escrow for support by a third party. Maintaining source code in this manner, either internally (within the enterprise) or by a third party, increases the cost of bug fixes and decreases the speed with which errors can be fixed. If this option is not available, replacement software must be found, which must be qualified and tested (see $C_{Refresh}$).

$$C_{Obsolescence\ SW} = C_{Maintenance\ SW}\ Obsolescence(N_{years})$$

Equation 28

where:

$C_{Maintenance\ SW}$ is the cost to maintain software, as given above, applied to non-proprietary, obsolete software

N_{years} is the number of years that the component has been obsolete

$Obsolescence(N_{years})$ is the proportion (≥1) of the maintenance cost paid as a penalty to maintain software N_{years} after its obsolescence

2.1.5: Design Refresh Costs

A design refresh changes the components in the system in order to keep the system manufacturable and/or supportable. While many of the activities and costs are the same, this is different than a redesign or technology insertion, where individual components or subsystems are changed with the primary goal of improving performance or adding functionality.

The total cost to update a system to use new technology depends on the rate of technological advancement, the TRL of the technology to be implemented, limitations on the new technology imposed by other components,[38] the value of the added performance or functionality, and many other factors. The number and variability of these factors make modeling the cost of redesign difficult and inaccurate. However, this model assumes that using a MOSA approach will affect the ability to perform a refresh or a redesign in the same way – a system that is easier to refresh is easier to redesign, and the benefits of having an up-to-date system are similar to the costs avoided by having a more supportable system. Additionally, the benefit of increased functionality in a redesigned system should be similar to the costs avoided in a refreshed system, where new, less expensive components replace older ones with the same performance. Therefore, though the advantages of redesign cannot be directly quantified, the same information can be achieved by modeling refresh alone.

The cost of performing a design refresh is given by:

[38] For example, improved processor speed is not as valuable if there is not enough RAM or high enough bus speed. Similarly, the advantages of moving to a dual or quad core processor will not be realized if the software does not allow for multithreading.

$$C_{Refresh} = C_{Initiate} + C_{Design\ Implementation} + C_{NRE} + C_{Upgrade}$$

Equation 29

Initiating a Design Refresh

$C_{Initiate}$ are overhead costs associated with performing a design refresh that are not directly related to any specific component. These costs are similar to those in $C_{Design\ Infrastructure}$ and $C_{Design\ Creation}$, and are related to supporting, training, and maintaining a refresh team, determining which components are candidates for refresh, and scheduling the timing of refreshes. $C_{Initiate}$ can be modeled as:

$$C_{Initiate} = BC_{Implement\ Refresh} CF_{Implement\ Refresh}(System)$$

where:

$BC_{Implement\ Refresh}$ is the base cost to initiate a refresh

$CF_{Implement\ Refresh}(System)$ is the overall cost factor associated with implementing a refresh of $System$. It is dependent on the number of component types and standards used, and the modularity of the system, among other factors.

Design Implementation

The costs associated with $C_{Design\ Implementation}$ are the same as those explained above. However, these costs are only incurred for the components and subsystems that are affected by the design refresh.

NRE

C_{NRE} is defined as previously, though the costs are only incurred for the components and subsystems that are affected by the design refresh.

Upgrade

C_{Upgrade} are the costs associated with deploying a design refresh to fielded systems. This cost is highly dependent on the configuration management and upgrade strategy used by the enterprise. Due to the time and cost of reequipping fielded systems, system upgrades are generally spread over several years. Depending on the refresh schedule, some systems may skip an upgrade cycle.[39] C_{Upgrade} costs are the same as those in $C_{Production}$, though only applied to the components and subsystems that are affected by the design refresh.

2.1.6: Enterprise Costs

The enterprise costs are any costs that are incurred once at the enterprise level, but the benefits of which apply across several products or projects. These costs include the cost of maintaining a library of pre-qualified components, licensing or

[39] For example, if refreshes are conducted every two years, and each fielded system is updated every fourth year. A system will then receive version updates 1, 3, and 5, (in years 4, 8, and 12) but skip updates 2 and 4.

purchase costs for tools and databases that are used across the design and support of multiple systems, any common maintenance support infrastructure or staff, training of staff, incentive programs to encourage the use of library components, and the development of architectures based on industry reference models. Other enterprise level costs include the cost of researching components and standards that aren't used in any system.

2.2: Model Outputs

The primary output of the model is the total life cycle cost difference between two system designs. The costs associated with each cost category are tabulated separately, allowing for a better understanding of where the extra costs are incurred. However, total cost for each system is *not* calculated. This model cannot be used to predict the total life cycle cost of either system, nor can it be used to determine which cost categories will have the largest absolute expenditures over the life of the system.[40]

As noted, many of the less well defined parameters are input as probability distributions. A Monte Carlo approach is used, so the total costs are also expressed as distributions. The mean of this output distribution can be interpreted as the "likely"

[40] For example, say the design cost for two competing systems are $1.8M (for System A) and $1.7M (for System B), while operation and support costs are $1.5M and $1.1M respectively. This model would only be able to report a cost difference of $0.1M for design and $0.4M for operation and support. More money is spent on design, but the biggest effect the design changes have on cost fall under operation and support.

outcome, while the spread of the distribution is a measure of both the uncertainty in input parameters, and the risk associated with a design.[41]

In addition to the total life cycle cost difference between two designs, each architecture design is given an accessibility score and a modularity score. These are primarily used to give a quick estimate of the overall accessibility and modularity of the architecture so that some context can be provided for cost differences between architectures. The accessibility score measures the average accessibility of all the components used in the architecture. The modularity score takes into account the number of interfaces each component has, the number of standards used, the interdependency of specific components, and the ability of subsystems to be re-qualified independently. Both scores are measured on a scale between 0 and 1, with higher values corresponding to greater accessibility/modularity.

2.3: Model Implementation

The following sections detail the equations and simplifying assumptions that are specific to the current implementation of the model, used to produce the case study in Chapter 3.

As explained previously, this model is intended to compare two or more possible designs of an electronic system late in the design stage. Comparing multiple systems helps to reduce epistemic uncertainty by reducing the impact of unknown or

[41] Consider two system architectures compared to a baseline, where the mean savings associated with System A are slightly greater but more widely distributed than those associated with System B. Using System A could provide for greater cost avoidances, but could also result in far more costs incurred.

unknowable factors. Costs, both known and unknown, that are the same for both designs cancel out, and are not considered in the analysis. Additionally, since this analysis will be conducted after the initial design phase has been completed with the goal of determining which of two systems is the better long-term investment, some of the early design and development costs may be treated as "sunk costs," and omitted from the calculations.[42] This assumption helps to reduce both epistemic and aleatory uncertainty.

Many of the equations presented previously use a cost factor that is a function of the openness metrics to help predict various life cycle costs. This methodology is a common approach in cost modeling. It is used by many models, including the Loral Federal Systems model and all models in the COCOMO family. In all such models, a base effort or cost is estimated and then adjusted using one or more cost factors. In this model, all of the individual cost factors are assumed to be independent, and have a multiplicative relationship to form the overall cost factor. In other words, the overall cost factor may be given by:

$$CF_{Overall} = \prod_{i=1}^{N_{OM}} CF_i(OM_i, weights)$$

Equation 30

where:

$CF_{Overall}$ is the overall cost factor

[42] An alternate goal is to use previous experience and data to predict what type of system or what level-of-openness is the most cost effective to design, as well as build and operate, in order to establish an enterprise-level design strategy for future projects. In such a case, design and other early costs should be accounted for because they play an important role in the overall cost of future systems.

N_{OM} is the number of openness metrics (and individual cost factors) associated with the current $CF_{Overall}$

OM_i is the *i*th openness metric

$weights$ are a set of input parameters that define the scale used to measure OM_i

$CF_i(OM_i, weights)$ is the *i*th individual cost factor, which is calculated by measuring the value of OM_i on a scale defined by $weights$. Depending on the metric and location, CF_i may be calculated using one of the following:

$$CF_i = Linear(OM_i, weights)$$

$$CF_i = Logistic(OM_i, weights)$$

$$CF_i = Asymptotic(OM_i, weights)$$

Equation 31

$Linear(CM_i, weights)$ is a function that weights the value of metric *i* on a linear scale between a minimum and maximum value

$Logistic(CM_i, weights)$ is a function that weights the value of metric *i* on a logistic or "S" shaped curve, similar to those used in modeling population growth. These curves are characterized by slow initial growth starting from a minimum value, followed by a period of exponential growth, and then slowing growth as a maximum value is approached.

$Asymptotic(CM_i, weights)$ is a function that weights the value of metric *i* on an asymptotically increasing curve. This curve is

characterized by fast initial growth from an initial minimum value. The speed of the growth diminishes as the maximum value is approached.

$weights$ are the lower and upper values of the curve, and the metric values to which those bounds correspond. These values depend on the specific metric, and are used to help tune and calibrate the model.

2.3.1: Design Costs

Design Infrastructure

Design infrastructure costs are dependent on the enterprise. Since the same enterprise is designing both systems, these costs will be approximately equal and can therefore be treated as a wash. Additionally, these costs would be incurred early in the design process, so they may be considered sunk costs for this analysis. As such, they are excluded from this implementation of the model.

Design Creation

At the time this analysis is conducted, the two (or more) prospective designs will already have been created. However, for the purposes of this model, the design of both options considered can be considered sunk costs.

Design Implementation

The implicit assumption is made that all components selected for integration are suitable for their intended location, and that once they are integrated and qualified, interacting components behave properly. All issues associated with architectural mismatch or other incompatibilities are assumed to be included in the selection and implementation costs. Additionally, when calculating integration costs, all standards and components are assumed to have interfaces of similar difficulty.

For the purposes of this model, it is assumed that the cost analysis is taking place after both designs have been created, but not fully implemented. The nonlinear scale factor *A* in the CICC-based model for COTS software integration is therefore taken as the "nominal" value of 1.04 [63].

2.3.2: Non-Recurring Engineering Costs

Supplier NRE

In many cases, the number of suppliers needed would be similar for the two systems being compared, so supplier qualification costs may be treated as a wash. [43] Additionally, in large enterprises, the same suppliers will be used to supply components for multiple types of systems, so many of the suppliers will already have

[43] This is not always a valid assumption, particularly in cases where the enterprise is trying to decide whether or not to alter their design paradigm. Switching from mostly closed, proprietary designs to open, COTS-based ones, for example, would require qualifying many new suppliers. For large enterprises, however, the cost of this shift would be amortized over many projects or considered a one-time investment, and can so be assumed to play a negligible role in any given system.

been qualified. In such cases, supplier qualification costs may also be treated as a sunk cost. For the purposes of this implementation, one or both of these are assumed to apply, so supplier qualification costs are omitted from the calculations.

Standard NRE

In practice, when an enterprise is selecting an interface standard for use in a system, it will research several possibilities, and select the one most appropriate for the current situation. The cost of researching unused standards should also be included. This model will assume that these costs would be similar in all cases, and that these costs are included in the base cost of selecting a standard. If an in-house, proprietary standard specific to the enterprise is to be used, it is assumed that the standard already exists – costs to develop new standards are not included in the model.

Component NRE

Similar to the process used when selecting a standard, an enterprise will generally investigate several potential components, including COTS or proprietary options, before deciding which component will be used. The costs associated with considering other components are assumed to be roughly independent of the final selection, and are therefore included in the base cost of selecting a component.

2.3.3: Production Costs

Procurement

In this model, procurement costs are assumed to independent of the number and size of purchase orders made. This is true for initial production, maintenance, and sparing during the life of the system, and for updates made to existing systems. Additionally, all ordered components are assumed to arrive just before they are needed, so no holding costs are incurred. All components needed for production in the current year are ordered and delivered at the beginning of the year.

The cost to procure each instance of a component is assumed to be constant over the course of its procurement life. When a component is refreshed, it is replaced with a newer version of similar functionality. Since the cost of a constant functionality level decreases over time, the procurement cost of the replacement component is expected to be less than that of the original. Abel, Berndt, and White, for example, found that between 1993 and 2001, the cost for software declined 4.26% annually, on average, while simultaneously receiving improvements and additional functionality [67]. During the same period, general prices increased 2% annually.

Part Preparation

Acceptance tests and modification processes generally have a non-zero failure rate, and some components are rejected. Depending on the supplier and the purchasing terms, some COTS components may be covered under a warranty. Otherwise, in addition to the components actually used, the enterprise must pay for

the purchasing and testing of inadequate components that are rejected during the testing process, even though they are not used in the final system. For this model, it is assumed that all components that fail acceptance tests are covered under warranty, and any costs associated with components damaged by uprating or modification are amortized over approved components, and included in the part preparation cost.

Assembly

All systems are produced at the beginning of the year. This means that no holding costs are incurred for the components used to produce the system. Additionally, each system must be supported for the full year in which it is created.

Recurring Test

Recurring test fees are those associated with ensuring that all components and subsystems of outgoing systems are functioning properly. In any real-world system, some percentage of units will fail these tests, and require rework. This model implicitly assumes that there is a 100% pass rate. Alternatively, the costs of components that fail quality control may be included in the base cost.[44]

[44] A methodology for calculating the added costs due to these failures is given in [65]. Though not implemented in the current model, the pass/fail rate of a system could be used in the cost factor function to account for these costs explicitly.

2.3.4: Operation and Support Costs

Operation

Software components are assumed to have no direct operational expenses, with the exception of costs associated with staffing or labor costs. If these costs are to be incurred, they should be modeled separately as a consumable item.

Maintenance

While components are not obsolete, all spares are assumed to be purchased and paid for at the beginning of the year. Sparing and delivery schedules are assumed to be perfect – no unneeded spares are purchased, and spares are delivered at exact time and place at which they are needed. This means that holding costs and downtime costs are minimal, and can be neglected.

Production and upgrade events occur at the beginning of each year, and retirement events at the end of each year. Systems must be supported for each year of their use, including the years they are produced and retired.

Obsolescence

In the year before a component becomes obsolete, the number that will be required post-obsolescence is estimated. This quantity, with an additional fraction as a buffer, is purchased at the beginning of the year. Components used in that year are drawn from this inventory without incurring any holding fees. At the end of the year,

and at the end of each subsequent year, holding costs are paid for any components remaining in inventory. Required spares are assumed to be delivered to the system at the exact time they are needed, so no downtime penalties are assessed.

For this implementation of the model, it is assumed that the enterprise uses the same strategy for all components. The possible strategies are lifetime buy, bridge buy, or replacement. For bridge buy, a predefined refresh period may be input, or the optimum refresh period may be calculated using a design refresh planning (DRP) model [53].[45]

In some situations, a component's interface standard may become obsolete before the component itself. This is interpreted as a phasing-out of the interface standard in favor of newer, more capable standards. After this point, the standard itself may no longer be supported, and no new components using the standard will be released. However, existing components utilizing the standard will remain available until their own obsolescence date.

2.3.5: Design Refresh Costs

Design Implementation

When a design refresh occurs, all obsolete components (with the exclusion of components that are explicitly excluded) are refreshed. If a component's interface

[45] Tools such as the Mitigation of Obsolescence Cost Analysis (MOCA) and the Multi-part Multi-event (MpMe) models may be used to perform this optimization.

standard is obsolete, the component is refreshed, even if the component itself is still procurable.

The implicit assumption is made that all components selected for integration are suitable for their intended location, and that once they are integrated and qualified, interacting components behave properly. All issues associated with architectural mismatch or other incompatibilities are assumed to be included in the selection and implementation costs. Additionally, it is assumed that the nonlinear scale factor *A* in the CICC-based model for COTS software integration is still the "nominal" value of 1.04 [63].

Non-Recurring Engineering Costs

It is assumed that the assumptions made previously still apply.

Upgrade

Upgrade costs are assumed to be independent of the number of systems upgraded per year. Once an update is performed, all new production is based on the newer design. Like production, all system updates occur at the beginning of the year. This means that in the year that a system is updated it must be supported with the new version of the component.

An "upgrade period" is used to define the number of years after a refresh occurs, by which point all existing systems must be updated to the most recent design. If the refresh period is shorter than the upgrade period, the upgrade period is the

maximum amount of time a fielded system can operate without receiving an update. Updates (and retirements) are done on a "first in, first out" basis, so that all systems are updated once before the first system receives a second update. However, if a system is scheduled to be retired (reach EOS) before the end of the upgrade period, it is not upgraded.

After a refresh has been performed, fielded systems may be supported by replacing failed components with stockpiled instances of the obsolete component or with an instance of the newer version. Using a newer version requires planning during the refresh process to provide complete backwards compatibility and to limit architectural changes to prevent mismatch.

2.3.6: Enterprise Costs

Some of these benefits may be captured in the previous sections, particularly when the metric of 'Enterprise Experience' is used in the calculation of the cost factor. Many of these benefits, however, cannot easily be quantified. For the purposes of this model, the rest of these costs and benefits will be considered "sunk", and not enumerated.

Chapter 3: Case Study

The model discussed in Chapter 2 was implemented in Microsoft Excel in the form of a Cost of Openness Tool (COT) – see Appendix D for a description of the tool. The tool contains separate worksheets that allow for the definition of the standards, components, parameters, parameter weights, simulation settings, and architectures being examined. Some inputs may be defined as probability distributions. A simulation consists of a predefined number of trials (defined in the simulation settings), during which distributions are sampled. The resulting range of cost differences represents the likely outcomes.

In the sections that follow, two sample architectures based on the US Navy ARCI sonar system are presented. These are used to demonstrate the impacts that the use of MOSA has on a system's life cycle cost, and how investing in an open strategy may affect an enterprise's design and support methodologies.

3.1: Architecture Definitions

When comparing architectures, it is important that they are of similar functionality and described at the same level of detail. This ensures that the cost difference results obtained from the model are primarily due to the difference in openness and the related metrics. Two architectures, based on the first two implementation phases of the Acoustic-Rapid COTS Insertion (ARCI) sonar system as described in [1], with additional information from the AFRL/RYM Interconnect

Trade Study [69], were used to perform the case study. These systems will be referred to as ARCI-1 and ARCI-2 respectively. The baseline metric inputs for the standards and components used, as well as the architectures themselves, are described in Sections 3.1.1 through 3.1.4.

3.1.1: Standards

Every standard used in the model is required to have:

A procurement life, which represents the time between the standard's introduction and obsolescence. This may be described using a probability distribution.

A life code, which indicates where the standard is in its life cycle. A value between 1 and 6 is used, where 1 denotes introduction phase, and 6 denotes obsolescence.

Other information about the standard may also be used, including:

Market share, the percentage of the relevant market that uses the standard

The volatility of the standard, measured in expected number of minor changes or updates per year

The years of use, or how long the standard has been available

Enterprise experience, the length of time, in years, that the enterprise has used the standard

"Standard Defined By", selected from:

"In-house proprietary standard": a proprietary standard that is maintained by the enterprise itself

"Defined by a single entity or group, with restrictions": a proprietary standard belonging to an external entity

"Defined by single entity, no licensing restrictions": typical classification of a de facto standard

"Defined by group, no licensing restrictions": an open standard

The definitions of the standards used in the case study are given in Table 1.

Table 1: Definitions of standards used in the case study

Name	Procurement Life (years)	Life Code	Market Share	Volatility	Years of Use	Enterprise Experience (years)	Standard Defined By
Ethernet	Normal(30, 5)	3	60%	0.5	25	0	Defined by group, no licensing restrictions
Infiniband	Normal(25, 5)	3	20%	0.5	15	2	Defined by group, no licensing restrictions
DDS	Normal(30, 5)	3	30%	0.5	10	0	Defined by group, no licensing restrictions
MTM	Normal(25, 5)	3	10%	1	15	2	Defined by entity or group, with licensing restrictions
SPARC	Normal(50, 5)	3	30%	0	10	2	Defined by entity or group, with licensing restrictions
SHARC	Normal(50, 5)	3	30%	0	10	2	Defined by entity or group, with licensing restrictions
RISC-1	Normal(50, 5)	3	30%	0	10	2	Defined by entity or group, with licensing restrictions
RISC-2	Normal(50, 5)	3	30%	0	10	2	Defined by single entity, no licensing restrictions

"Normal(X, Y)" indicates a normal probability distribution with mean X and standard deviation Y. The procurement life of each standard is assumed to be normally distributed. Other distributions could also be selected.

An "entity" is a single enterprise or other body that defines a standard. A "group" is a consortium of two or more enterprises or other entities that cooperate to formulate and maintain a standard.

3.1.2: Components

Each component is required to have:

A type – Hardware, Software, or Consumable. Consumables are treated as hardware, except that a constant failure/replacement rate is assumed, instead of sampling from a failure distribution.

An Interface Standard, selected from those defined for the Standards

A Procurement cost (which can be a distribution)

A time-to-failure distribution

A procurement life, which indicates the time between the component's introduction and obsolescence (modeled as a distribution)

A life code, which indicates where the component is in its life cycle. A value between 1 and 6 is used, where 1 denotes introduction phase, and 6 denotes obsolescence.

Accessibility of the component's technical data package (TDP) and product definition, selected from:

"Component designed in-house": A component that is designed and maintained by the enterprise itself. This is an option for components that use an in-house or open (unrestricted) standard.

"Limited access (Proprietary/COTS)": A component designed and maintained by an external entity. This may be a component that is created by a contractor for this

specific purpose (proprietary), or a COTS component. In either case, the component is treated as a "black box" – the enterprise cannot access internal specifications or workings of the component for debugging or modification purposes. This option cannot be used if the component uses an "in-house" standard.

"Full access (COTS/Open Source)": A component that is designed and maintained by an external entity, but to which the enterprise has access to the TDP and internal workings of the component, facilitating debugging, and allowing for some modifications. This may be used for components that use "unrestricted" standards.

A Design Refresh Plan (DRP) option, selected from:

"Included in DRP": Components that are included in design refresh planning.

"Excluded from DRP": Components that are excluded from design refresh planning, but which are refreshed at the first opportunity after obsolescence.

"Not refreshable": Components that are excluded entirely from refresh planning and implementation – if the component becomes obsolete, a lifetime buy strategy is used, regardless of the strategy used for the rest of the system.

Other information about the component may also be relevant, including:

Cost of annual licensing, where applicable (typically only applicable to some software)

Software size, given in thousands of lines of source code (KSLOC)

The number of competing components

The volatility of the component, measured in expected number of minor changes or updates per year

The length of use, or how long the component has been available

Enterprise experience, the length of time, in years, that the enterprise has used the component

Definitions of the components used in the case study are shown in Table 2. For all of the components shown, annual licensing costs are assumed to be $0. All hardware components are assumed to have a time-to-failure distribution that can be modeled using a Weibull distribution with shape parameter of 1.75 and scale parameter of 2 years. Weibull distributions are often used to model component failure because selecting an appropriate shape parameter allows for modeling of all three portions of the reliability bathtub curve - infant mortality, random failure, and wear-out. A shape parameter of 1.75 and scale parameter of 2 years gives a positively-skewed distribution with mean of 1.75 years, and variance of 1.10 years, corresponding to wear-out failures. Each of the software components measures 100 KSLOC. Additionally, all components are included in DRP calculations. Since all standards and components used are COTS and therefore at least nominally open, the

data used reflects the assumption that components and standards with higher openness will have greater market share, longer procurement lives, and lower costs.

The procurement life and cost of each component are assumed to be normally distributed. Other distributions could also be used.

Table 2: Definition of components used in the case study

Name	Type	Interface Standard	Procurement Cost ($)	Procurement Life (years)	Life Code	# Competing	Volatility (changes/year)	Years of Use	Enterprise Experience (years)	Accessibility of TDP
Ethernet NIC	Hardware	Ethernet	Normal(75, 10)	Normal(5, 1)	3	10	1	10	0	Full access (COTS/Open Source)
Infiniband HCA	Hardware	Infiniband	Normal(125, 10)	Normal(3, 1)	3	5	0.5	5	2	Limited access (Proprietary/COTS)
DDS Stack	Software	DDS	Normal(50, 5)	Normal(5, 1)	3	10	1	3	0	Full access (COTS/Open Source)
MTM Stack	Software	MTM	Normal(50, 5)	Normal(3, 1)	3	2	0.5	5	2	Limited access (Proprietary/COTS)
SPARC Card	Hardware	SPARC	Normal(90, 5)	Normal(5, 1)	3	5	1	2	2	Limited access (Proprietary/COTS)
AD21062 Card	Hardware	SHARC	Normal(90, 5)	Normal(5, 1)	3	3	1	2	2	Limited access (Proprietary/COTS)
i860 Card	Hardware	RISC-1	Normal(55, 5)	Normal(3, 1)	3	5	1	2	2	Limited access (Proprietary/COTS)
Quad i860	Hardware	RISC-1	Normal(200, 10)	Normal(3, 1)	3	5	1	2	2	Limited access (Proprietary/COTS)
PowerPC Card	Hardware	RISC-2	Normal(55, 5)	Normal(5, 1)	3	5	1	2	2	Full access (COTS/Open Source)
Quad PowerPC	Hardware	RISC-2	Normal(200, 10)	Normal(5, 1)	3	5	1	2	2	Full access (COTS/Open Source)

3.1.3: Architectures

An architecture is defined by a hierarchy of subsystems and components. Each layer in the hierarchy is given a level number and a number of instances. Low-level subsystems are made up of higher level (more detailed) subsystems and components. The instance number tells how many times the subsystem appears in the architecture. The lowest (outermost) subsystem (the zero-level) represents one system instance

Additional data that is specific to the application of the component that may be used includes:

The up-rating and modification costs per unit (hardware components)

The amount of glue code required to integrate the component (software components)

The percentage of the total functionality used

The number of functional interfaces the component has, both with components in the current subsystem, and with components in other subsystems.

If two (or more) components are interdependent, such that the refresh of one necessitates refresh of the others.

The ability of one or more specific subsystems to be requalified independently, without requalifying the design of the entire system. This depends on the modularity of the design and any regulations affecting the system)

Each architecture is given an accessibility score and a modularity score, measured on a scale between 0 and 1, to allow for a quick approximation of the relative openness and modularity of different architecture designs. The accessibility score is the arithmetic mean of the accessibility scores of each individual component used in an architecture. The accessibility score for a component depends on its accessibility, as well as the governing body which defines the standard it uses. The possible combinations and ratings are given in Table 3.

Table 3: Accessibility scores for individual components

Standard defined by:	Accessibility of TDP:	Score:
In-house proprietary standard	Component designed in-house	0
Defined by entity or group, with licensing restrictions	Limited Access (Proprietary/COTS)	0.14
Defined by single entity, no licensing restrictions	Component designed in-house	0.29
Defined by group, no licensing restrictions	Component designed in-house	0.43
Defined by single entity, no licensing restrictions	Limited Access (Proprietary/COTS)	0.57
Defined by group, no licensing restrictions	Limited Access (Proprietary/COTS)	0.71
Defined by single entity, no licensing restrictions	Full access (COTS/Open Source)	0.86
Defined by group, no licensing restrictions	Full access (COTS/Open Source)	1

The modularity of a component is defined as the multiplicative inverse of the component's number of interfaces (1/interfaces). The modularity score for each subsystem is the average modularity of the encapsulated subsystems and components, with small penalties for components that are interdependent and subsystems that cannot be requalified independently. At the architectural level, a weighting is used to reduce architectures that utilize a larger number of standards.

The architectures for ARCI-1 and ARCI-2, which were used in the case study, are shown in Table 4 and Table 5. ARCI-1 received an accessibility score of 0.2245 and a modularity score of 0.7079. ARCI-2 scored higher on both measures, receiving an accessibility score of 0.8929 and a modularity score of 0.8091.

The architectures reflect the assumption that the more open design (ARCI2) incorporates greater reuse (and uses only three standards and four unique component types, compared to ARCI1's five standards and five component types), while the less open system uses more specialized components, allowing for greater design efficiency (each system using the ARCI1 architecture is made up of 7 components, while each ARCI2 is made up of 11 components).

Table 4: Definition of ARCI-1 Architecture

Level	Name	Is Component	Requalification Possible	# Instances	Standards Type	Uprating /Modification cost/unit (HW)	KSLOC of glue code required (SW)	% Functionality Used	Number of Interfaces	Subcomponents Interdependent
0	ARCI-1		TRUE	1						FALSE
1	MPP Module		TRUE	1						FALSE
2	Signal Conditioner		FALSE	1						FALSE
3	SPARC Card	TRUE	FALSE	1	SPARC	0	0	100%	1	
2	Allocatable Drawer		FALSE	1						FALSE
3	AD21062 Card	TRUE	FALSE	1	SHARC	0	0	100%	1	
3	Quad i860	TRUE	FALSE	2	RISC-1	0	0	100%	1	
3	SPARC Card	TRUE	FALSE	1	SPARC	0	0	100%	1	
2	Infiniband HCA	TRUE	FALSE	1	Infiniband	0	0	100%	1	
2	MTM Stack	TRUE	FALSE	1	MTM	0	0	100%	1	

Table 5: Definition of ARCI-2 Architecture

0	ARCI-2		TRUE	1						FALSE
1	MPP Module		TRUE	1						FALSE
2	Signal Conditioner		TRUE	1						FALSE
3	PowerPC Card	TRUE	FALSE	2	RISC-2	0	0	100%	1	
2	Allocatable Drawer		TRUE	1						FALSE
3	Quad PowerPC	TRUE	FALSE	5	RISC-2	0	0	100%	1	
3	PowerPC Card	TRUE	FALSE	2	RISC-2	0	0	100%	1	
2	Ethernet NIC	TRUE	FALSE	1	Ethernet	0	0	100%	1	
2	DDS Stack	TRUE	FALSE	1	DDS	0	0	100%	1	

3.1.4: Parameters

The parameters used to conduct the case study are shown in Table 6.

Table 6: Parameter values used in the case study

Base Cost to qualify a standard:	$500
Base Cost to qualify a HW component:	$500
Base Cost to qualify a SW Component (/KSLOC):	$50
Base Cost to qualify system design:	$500
Base Cost of outgoing test:	$500
Hardware replacement/installation cost ratio:	0.25
Base Cost to install SW update (/KSLOC):	$50
Base Cost to maintain proprietary SW (/KSLOC):	$200
Software update implementation frequency (/year)	1
Base Cost to manage one system configuration	$100
Annual Discount Rate	7.00%
Inventory Support Cost Ratio	0.3
Spares Overbuy Fraction	0.25
Spares Overbuy Penalty Ratio	0.25
Spares Underbuy Penalty Ratio	1.5
Software Obsolescence Penalty Ratio	1.5
Base cost to initialize a design refresh:	$500
Base cost to design proprietary HW component:	$500
Base cost to design proprietary SW component (/KSLOC):	$50
Base cost to integrate a HW component:	$500
Base cost to integrate a SW component (/KSLOC):	$50
Life code for replacement components	3
Life code for replacement standards	3

3.1.5: Simulation Control Parameters

The simulations described in this chapter were run using between 10 and 100 trials. Each trial models twenty system instances produced at the rate of four per year for the first five years of the simulation. No system retirement dates are specified, so all systems are retired at the final end of service (EOS) date. The discount rate is used

to translate all costs and avoidances into year 0 dollars. The effects of varying the EOS date and other parameters are examined in the sections that follow.

3.1.6: Example Model Result

The model is used to find the cost difference in each cost category between two architectures in each year that the systems are deployed. The results from each category may be summed to give the total cost in each year. The cumulative cost in each year is found by summing the total costs from year 1 up to the year of interest. This enables an enterprise to see when the primary cost differences are incurred. An example of this is shown in Figure 7.

Figure 7: Cumulative total cost difference between ARCI2 and ARCI1 in each operational year, for a support lifetime of 22 years. Each of the five trials is shown in purple, blue shows average result.

The total cumulative cost difference, the final cumulative cost difference observed in the EOS year, is an important figure in making a business case for

choosing one architecture over another, so it is the primary indicator examined in the analysis in the following sections.

Since some of the input data are described by probability distributions, a single iteration or "trial" of the modeling procedure results in one possible outcome, though it may not be the most likely outcome. Repeating the procedure several times gives a range of results representing the possible outcomes that can be statistically analyzed. The final end of service date is also an important factor because, as will be demonstrated, the decision to invest in a more open architecture depends on the number of years of system use. The results of each simulation will therefore be calculated for a range of EOS dates.

An example output from the model is shown in Figure 8. In this case five individual trials were conducted, and variables sampled in each trial were used to calculate the total life cycle cost difference for EOS dates between 20 and 25 years after system introduction. Each solid pink line shows the results of one trial, and the dashed yellow line shows the average result in each year.

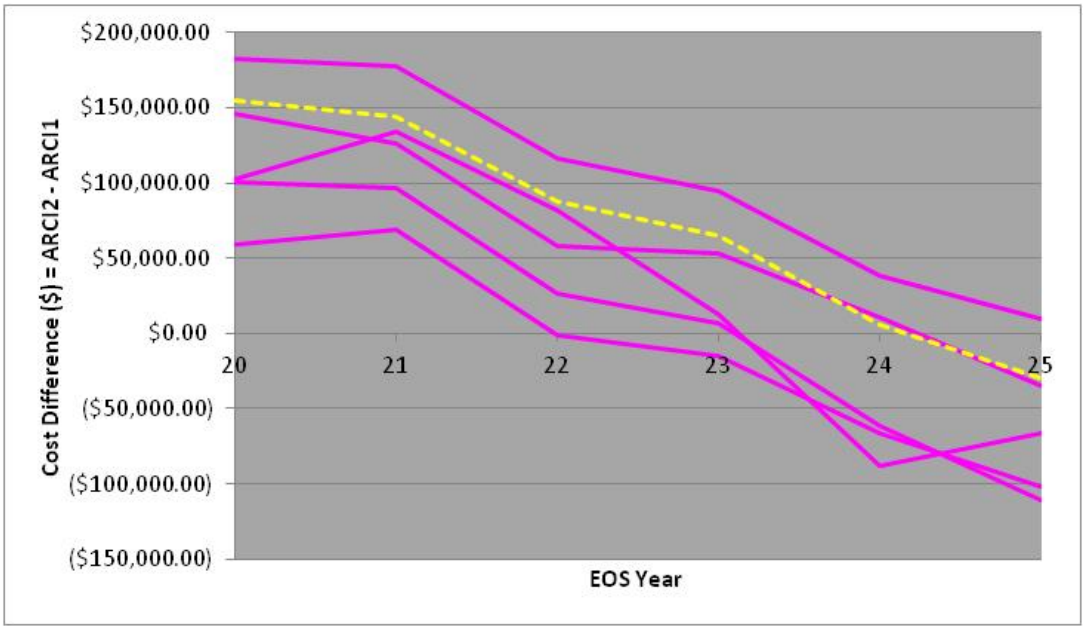


Figure 8: Example model results. Each solid pink line represents a single trial. The values sampled in each trial are used to estimate the total life cycle cost difference for support until each possible EOS year. The dashed blue line is the average outcome.

The difference, ARCI-2 minus ARCI-1, is positive in years 20 and 21, indicating that ARCI-2 is more expensive than ARCI-1 for support lifetimes of 20 or 21 years. The overall behavior of the different trials depends on the nature of the architectures, components, and management techniques being examined. Higher variability in the input distributions will lead to higher variability in the results. Some results, especially those involving lifetime buy management (which is highly dependent on the initial components' price and procurement life), may be observed to diverge for longer EOS dates. Other results, such as those involving bridge buy management (which implements a periodic design refresh), may give results that demonstrate periodic behavior over a range of EOS dates.

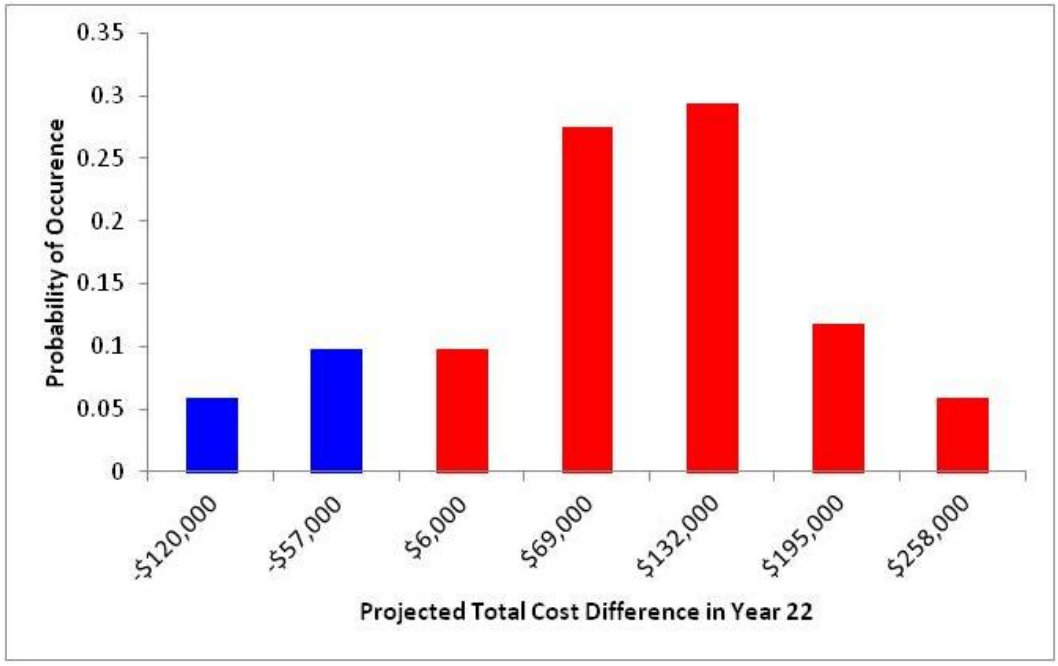


Figure 9: Histogram of the total cumulative cost difference (ARCI2 minus ARCI1) results from 50 trials for support until an EOS date of 22. ARCI1 is less expensive 80% of the time (shown in red)

The range and distribution of the results can be used to express the expected outcome and its confidence. For example, Figure 9 shows a histogram of 50 results of total cumulative cost difference for support until an EOS date of year 22 for the ARCI architectures using different support strategies. This histogram can be used to estimate the probability of a result falling in a particular range. The average result is a cost difference of approximately $87,000, with a standard deviation of $100,000. The ARCI2 architecture is more expensive than ARCI1 in 80% of the results.

3.2: Life Cycle Cost Analysis

When the ARCI-1 and ARCI-2 architectures described in Section 3.1.1 through 3.1.4 are sustained using a lifetime buy methodology, the less open ARCI-1 architecture usually costs less[46]. When a lifetime buy methodology is used, a bulk purchase of a component is made just before that component becomes obsolete. This stockpile is used to produce new systems, and support existing ones. This stockpile is ideally as close as possible to the number actually needed to support all systems through the EOS date. Figure 15 shows the results for EOS dates between year 10 and year 30. Though generally the more open system is expected to be less expensive to maintain, a lifetime buy methodology does not allow for many of the benefits of openness to be realized.

[46] The difference Cost(ARCI-2) –Cost(ARCI-1) is positive.

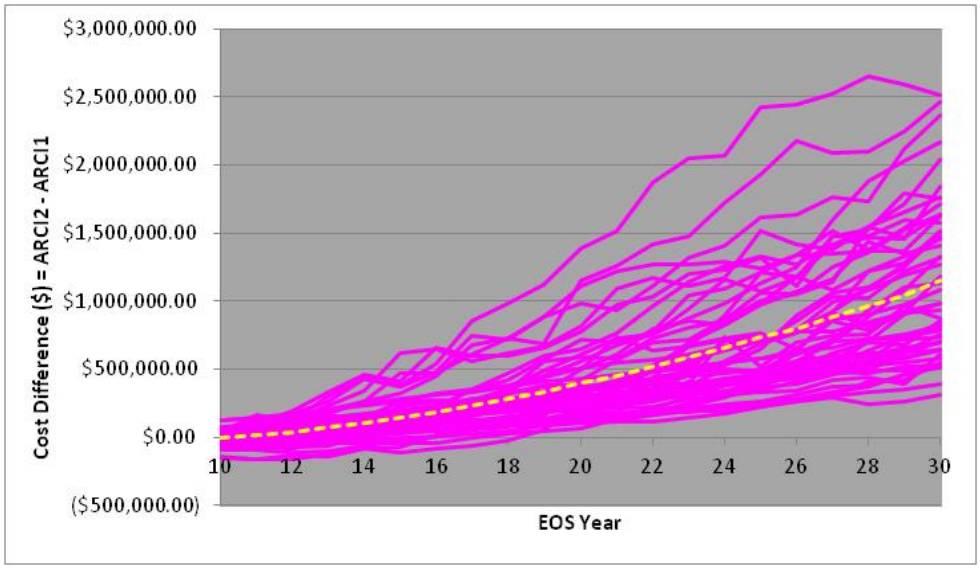


Figure 10: Total life cycle cost difference between ARCI-2 and ARCI-1 for the support of 20 system instances, using a lifetime buy methodology, for support lives between 10 and 30 years. Solid pink lines show individual trials, dashed yellow line shows average.

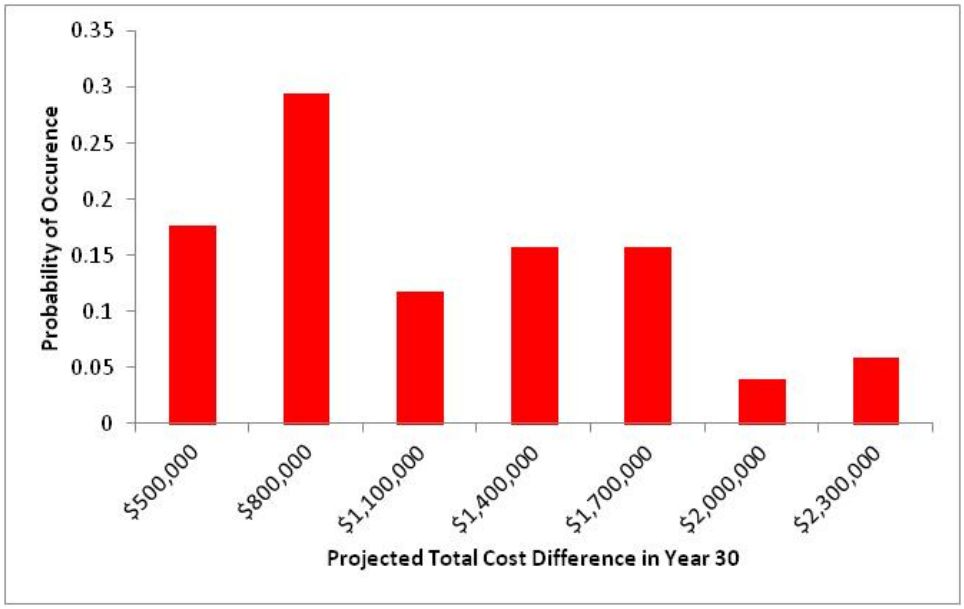


Figure 11: Probability of different outcomes for total cost difference between ARCI-2 and ARCI-1 architectures. Costs calculated for 50 trials of 20 system instances supported for 30 years using a lifetime buy methodology.

As Figure 10 shows, there is a large variability in the possible results. This variability is one of the weaknesses of relying on a lifetime buy approach to obsolescence mitigation – the result is highly dependent on the actual procurement lives of the components and standards. A small change in procurement life can result in a large overall cost difference. Despite this variability, the ARCI-1 architecture is expected to be less expensive than ARCI-2. Figure 11 shows a histogram of outcomes observed in year 30. The average cost difference was $1.1M, with a standard deviation of $575,000.

When a bridge buy methodology is used in conjunction with design refreshes, however, the value of ARCI-2's openness is observed. Figure 12 shows that for twenty system instances supported using a bridge buy methodology, where the design

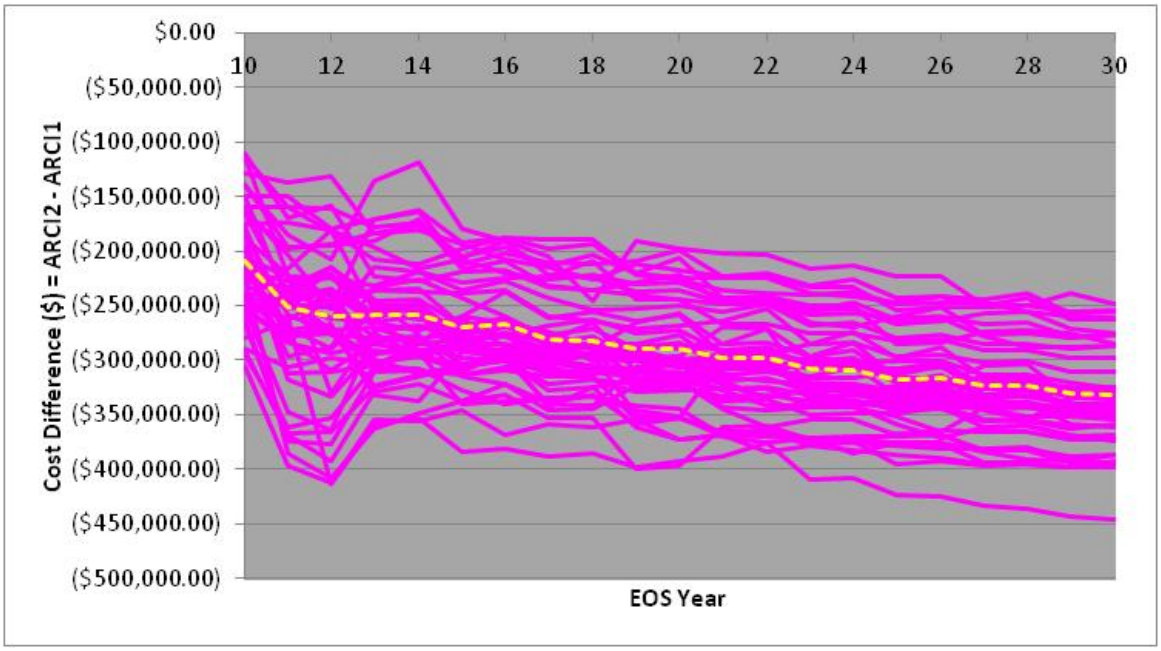


Figure 12: Total life cycle cost difference between ARCI-1 and ARCI-2 for the support of 20 system instances, using a bridge buy methodology, for support lives between 10 and 30 years. Design refresh occurs every two years, with systems upgraded every four years. Solid pink lines show individual trials, dashed yellow line shows average.

is updated every two years, and each system instance is updated every four years, the

ARCI-2 design is less expensive (the difference Cost(ARCI-2) – Cost(ARCI-1) is negative).

Using a bridge buy methodology helps limit the variability in cost due to obsolescence. This is especially true when a short refresh period is used, as in case shown in Figure 12 and Figure 13, because obsolete components are replaced relatively soon after obsolescence. In year 30, the average cost difference is negative $335,000 and the standard deviation is $44,000. For longer periods between refreshes, the shapes of the results curves are similar to the ones in Figure 12, though the variability is increased and the cost difference grows more slowly as the EOS date increases. As long as the support life is several times longer than the refresh and

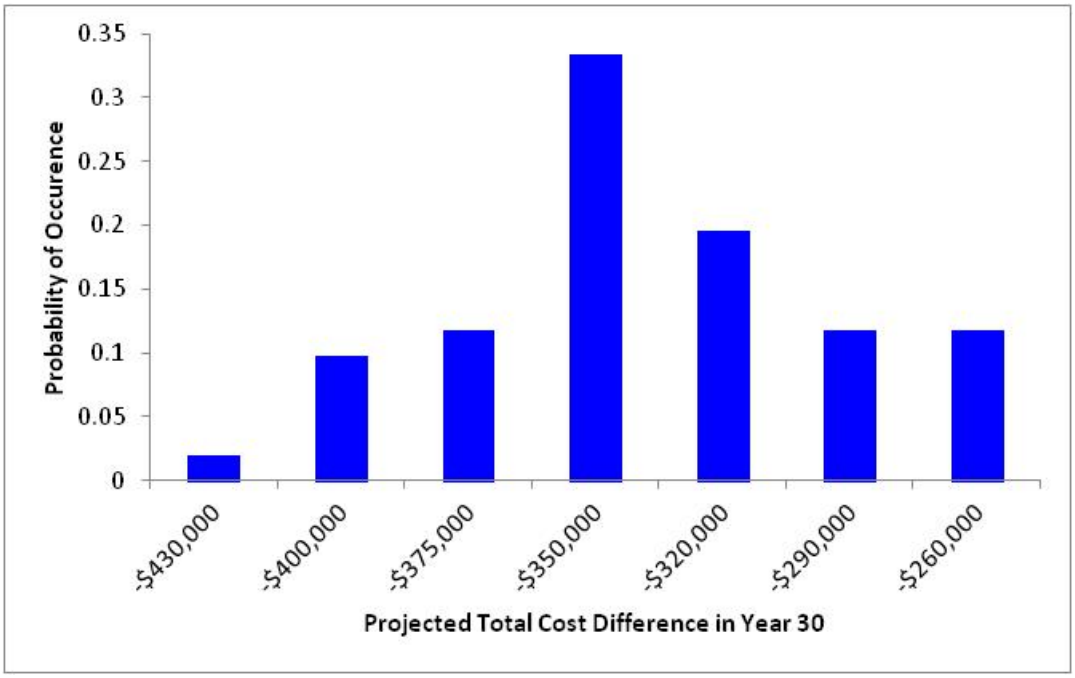


Figure 13: Probability of different outcomes for total cost difference between ARCI-2 and ARCI-1 architectures. Costs calculated for the case shown in Figure 12 - 50 trials of 20 system instances supported for 30 years using a bridge buy methodology.

upgrade periods, the more open ARCI-2 architecture is less expensive than ARCI-1.

When a lifetime buy approach is used, the largest contributions to the overall cost difference come from operation and support. When bridge buys are used, the refresh category is the largest contributor, because performing design refreshes is an expensive process (see Figure 14). In many scenarios, refreshing a system every two years may not be economically feasible nor the preferable approach to obsolescence mitigation. This is particularly true for systems with relatively short support lives, because obsolescence plays a smaller role in such systems.[47] This is shown in Figure 15, where the same ARCI-1 and ARCI-2 architectures are compared using different support methodologies. ARCI-1 uses lifetime buy, and ARCI-2 uses a bridge buy.[48]

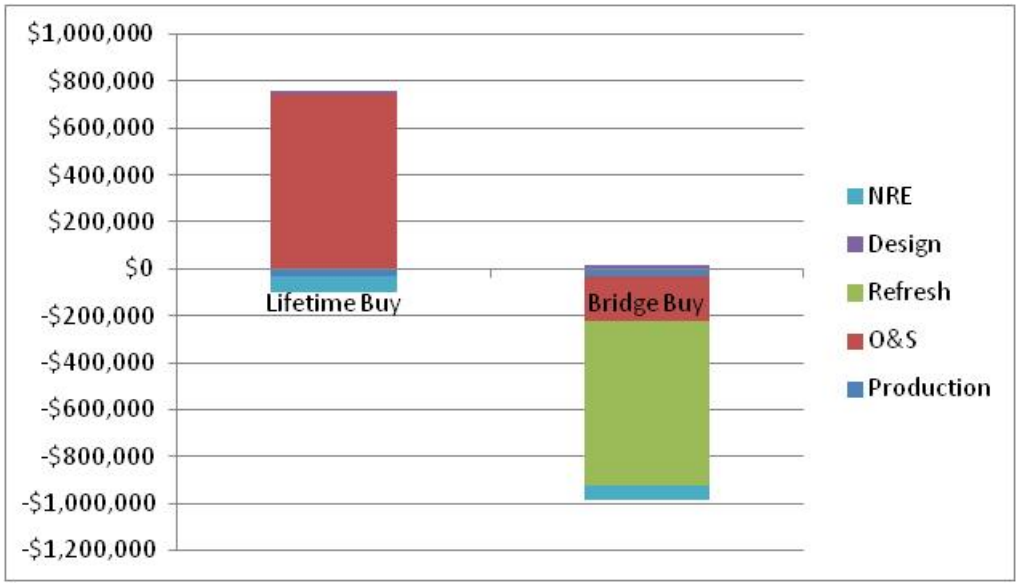


Figure 14: Average contributions, by category, to total life cycle costs for the cases shown in Figure 10 (lifetime buy) and Figure 12 (bridge buy). Contributions are positive when ARCI-2 is more expensive, negative when ARCI-1 is more expensive.

[47] For earlier EOS dates, obsolescence is less likely to be encountered and requires smaller component inventories when lifetime buys are made.
[48] As before, the refresh period is 2 years and the upgrade period is 4 years.

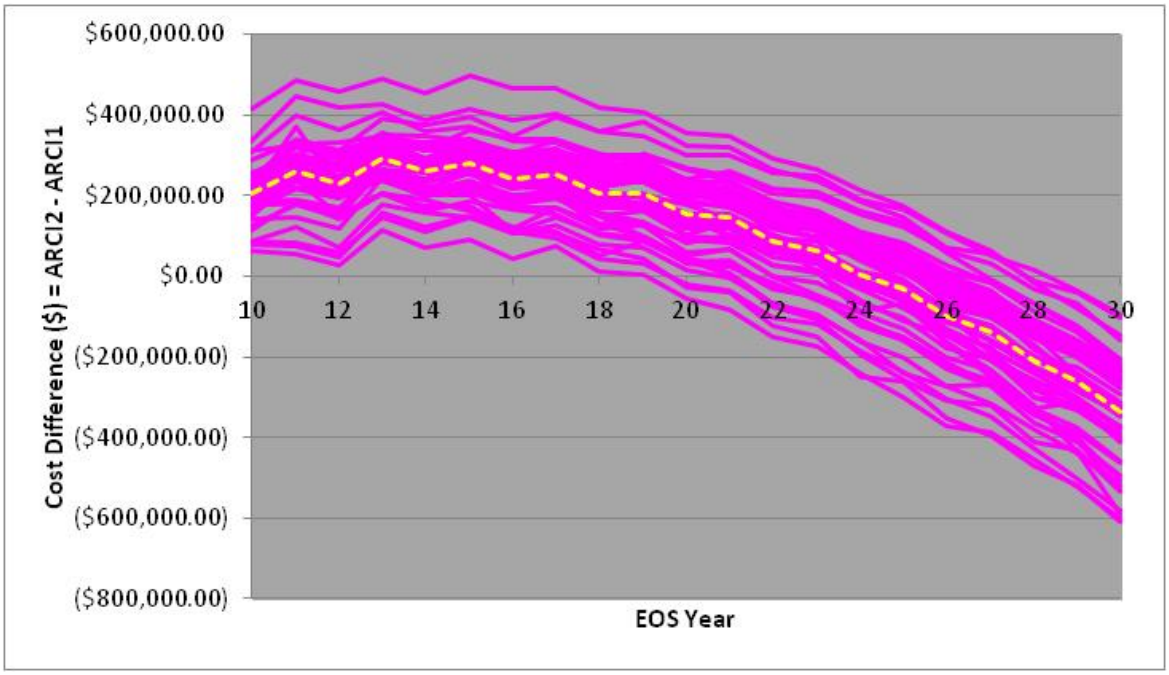


Figure 15: Total life cycle cost difference between ARCI-1 and ARCI-2 for the support of 20 system instances for support lives between 10 and 30 years. ARCI-1 uses lifetime buy, ARCI-2 uses a bridge buy with design refresh period of 2 years, and upgrade period of 4 years. Solid pink lines show individual trials, dashed yellow line shows average.

The result in Figure 15 is important because it demonstrates that using the more open architecture is not always beneficial – there is a 'break-even' point that occurs, in this case at EOS dates of approximately 25 years. If the systems are to be supported for fewer than twenty-five years, the more closed ARCI-1 architecture is less expensive regardless of whether a lifetime buy or a 2-year bridge buy is used.

The break-even EOS date depends on many factors. In addition to the architectures themselves, it is affected by the refresh period used for the bridge buy. As noted before, increasing the refresh period can, up to a point, decrease the total support costs.[49] The affect of using a bridge buy strategy in conjunction with different refresh periods is shown in Figure 16.

[49] Finding the optimal refresh period is a fundamental problem in system support. See [53].

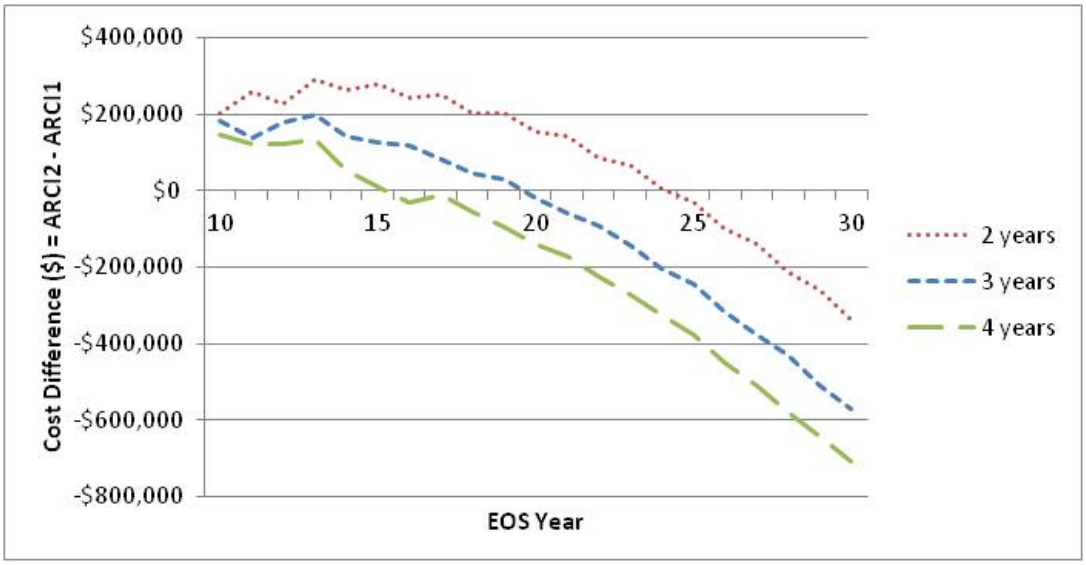


Figure 16: Effects of supporting ARCI2 with different refresh periods on the average total life cycle cost. In all cases, ARCI-1 uses a lifetime buy strategy.

Additionally, the number of system instances being supported can play a role –the cost of stockpiling components grows linearly with the number of systems, while the cost of conducting a refresh stays roughly constant. If only a few system instances are to be fielded, the more closed architecture may be the less expensive option. This is shown in Figure 17.

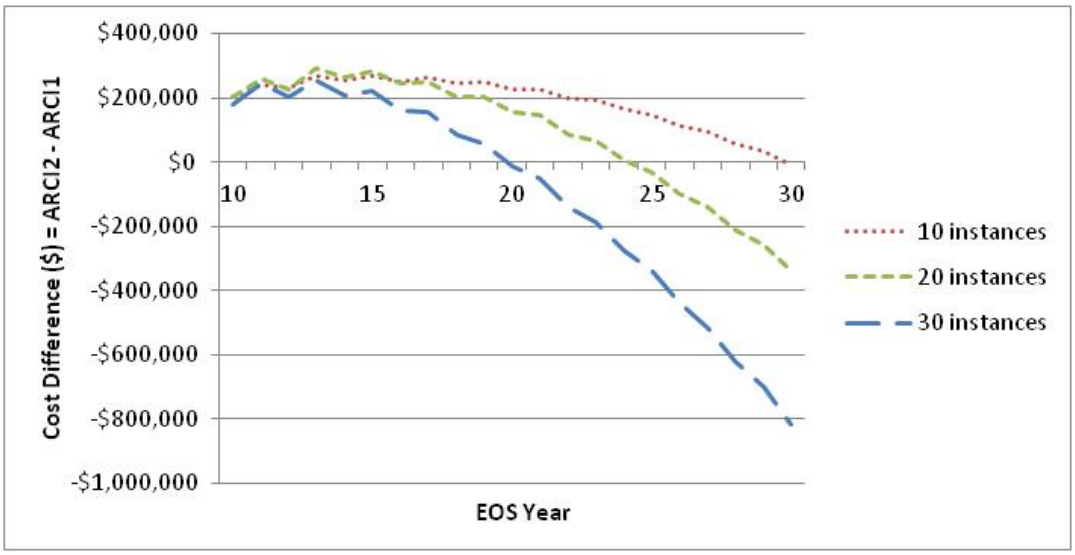


Figure 17: Effects of different number of fielded instances of the system on the average total life cycle cost difference for the ARCI architectures. ARCI-1 uses a lifetime buy strategy, ARCI-2 uses a two-year bridge buy.

3.3: COTS-LIMO Hypothesis

This model can also be used to examine Abts' COTS-LIMO hypothesis (see Section 1.5.5). Abts suggested the key factor in the life cycle cost of a COTS-based system (CBS) was not the amount of functionality provided by COTS components, but rather the COTS functional density (CFD). A CBS with low CFD, which uses more kinds of COTS components, is more expensive to maintain because of the higher likelihood of obsolescence and changes in its components.

To test the COTS-LIMO hypothesis, modified versions of the ARCI2 architecture were compared to the ARCI1 architecture. The only change made to the ARCI2 architecture was to split the functionality provided by the 5 Quad PowerPC boards among different versions of that component. The total number of components

in each design are the same, and each version of the Quad PowerPC board had the same properties and distributions – the architectures are effectively the same, but with different CFDs. The effects of this modification can be seen in Figure 18.

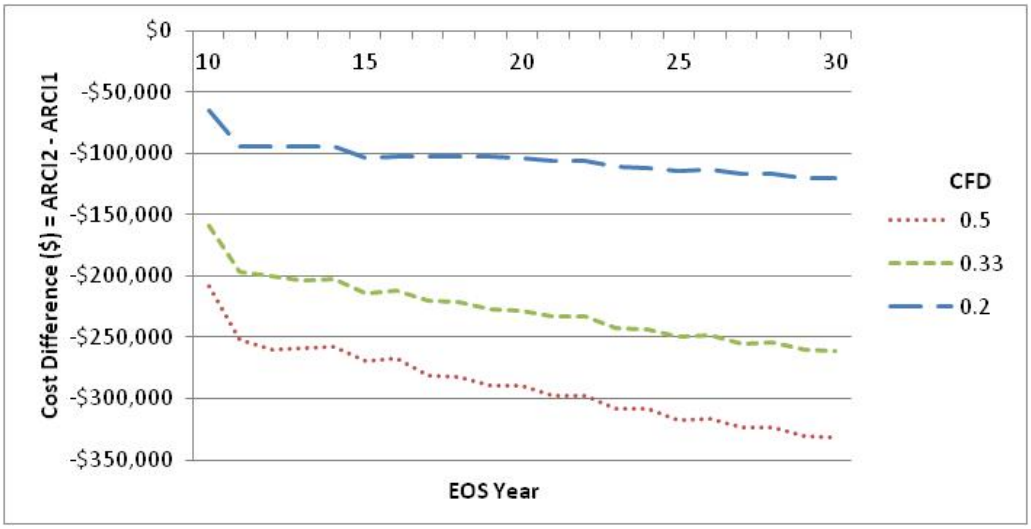


Figure 18: Effect of COTS Functional Density (CFD) on average total life cycle cost. Both architectures are supported using a bridge buy, with a design refresh occurring every two years. CFD is calculated for the allocatable drawer in the ARCI2 architecture, with the assumption that the drawer fills three functions.

The result in Figure 18 provides support for Abts' COTS-LIMO hypothesis because it demonstrates that decreasing ARCI2's CFD increases its life cycle cost (ARCI2 – ARCI1 becomes less negative).This result also shows the value of component reuse. Extreme care must be taken when designing a system, because even seemingly superficial design changes can have a significant impact on life cycle cost.

3.4: Life Extensions

As shown in Section 3.2, a bridge buy strategy, in conjunction with periodic design refreshes, is beneficial when a design is to be supported until a known EOS date because it decreases both the total life cycle cost and the variability of that cost.

This allows cost forecasts and budgeting to be planned more accurately. Use of periodic design refresh can also be advantageous when the end of support date is unknown or uncertain. This is because the pattern of refreshing the design and updating fielded systems may simply be repeated until the extended EOS date is reached. In a similar scenario, use of a lifetime buy management technique creates significant logistical and financial difficulties. Purchasing and holding a stockpile of components is expensive, so when a component becomes obsolete and a lifetime buy is made, the number stockpiled is ideally as close as possible to the actual number of components required to support the fielded systems until EOS.[50] If the system is life-extended, the stockpiles of components on hand may be insufficient, and potentially difficult or impossible to replenish.

To examine the effects of life-extension on total system cost, that the model starts by assuming that the system will be supported until a given default EOS date. Some number of years before the EOS date arrives, notice is given that the system is to be life-extended, and must be supported until a new (later) EOS date. Before the life extension warning, all support calculations, including stockpiling of components, are made until the original EOS date. After the notice of life extension has been received all calculations are made using the new EOS.

The effects of life-extension were modeled using the ARCI-1 and ARCI-2 architectures as described in Section 3.1, and the same production schedule used previously. The default EOS date was year 20, and notice of life extension was given

[50] In practice, the lifetime buy problem is an asymmetric newsvendor problem in which the penalty associated with running short of components is significantly greater than the penalty of having too many. As a result, a "buffer" is usually purchased on top of the actual demand estimate.

in year 15. Life extensions between 1 and 10 years were examined, corresponding to final EOS dates between year 21 and year 30. The overbuy ratio was 0.25.

For the case shown in Figure 19, ARCI-1 is managed using a lifetime buy, and ARCI-2 is managed using a bridge buy with a refresh occurring every two years, as before. For the case in Figure 20, both architectures are managed using a two-year bridge buy.

These figures demonstrate the large effect that life extension can have on life cycle cost, particularly when a lifetime buy support methodology is employed. The dashed green line overlaid in Figure 19 shows the average results attained in Section 3.2, when the final EOS date is known from the beginning. In Figure 19, the averages initially coincide (a life extension of 0 years), and then quickly diverge, with an order of magnitude difference between them after only 10 years. The analogous dotted yellow line in Figure 20 shows that using a bridge buy methodology with prescheduled periodic refreshes can effectively neutralize the issues associated with life extension.

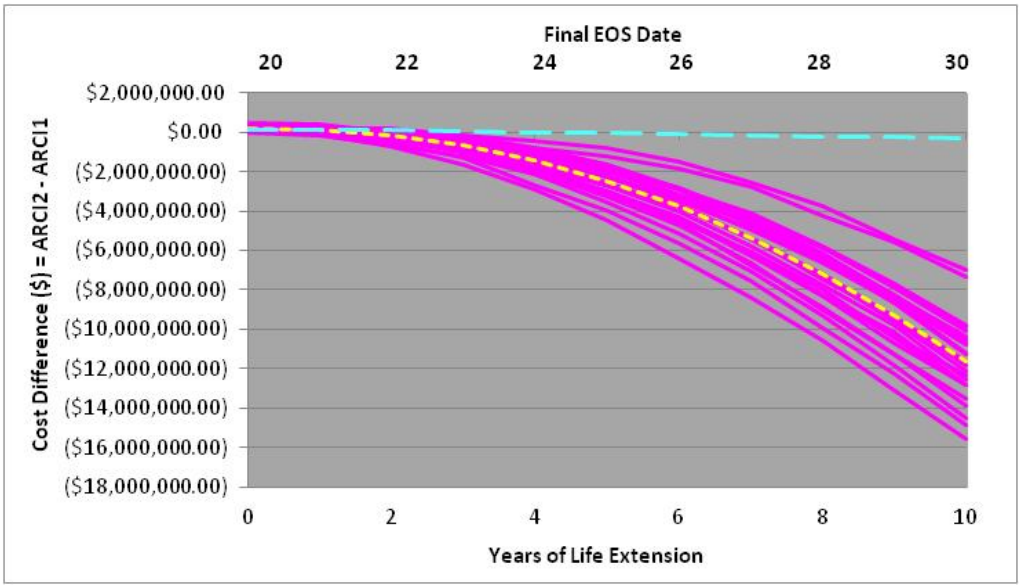


Figure 19: Total life cycle cost difference between ARCI-1 and ARCI-2 for life-extensions of up to 10 years from an initial (default) EOS date of year 20. ARCI-1 uses a lifetime buy, ARCI-2 uses a bridge buy. The solid pink lines show individual trials, dotted yellow line shows the average. The dashed blue line, overlaid from Figure 15, is the average total cost difference when the final EOS is known from year 0.

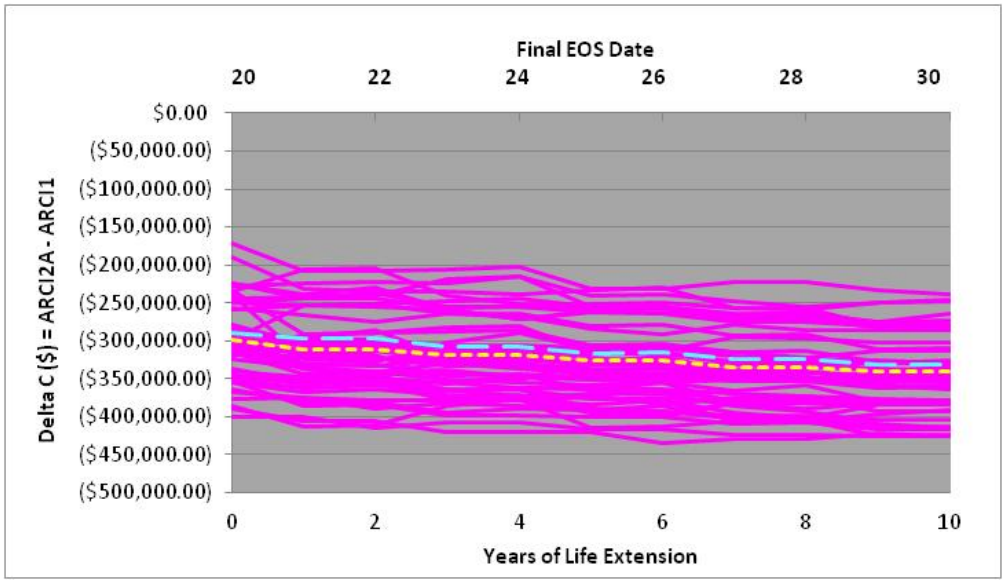


Figure 20: Total life cycle cost difference between ARCI-1 and ARCI-2 for life-extensions of up to 10 years from an initial (default) EOS date of year 20. Both architectures use a bridge buy. The solid pink lines show individual trials, and dotted yellow line shows the average. The dashed blue line, overlaid from Figure 12, is the average total cost difference when the final EOS is known from year 0.

Chapter 4: Conclusions

4.1: Summary

Openness is the characteristic of being accessible and available. For objects such as an architecture or a component, openness is a measure of how well the object conforms to well-known standards, and the openness or accessibility of those standards. The goals of openness are commonly expressed as interoperability, maintainability, extensibility, composability, and reusability. In general, these openness goals are attained using some combination of modular architecture, standardization of components and interfaces, use of open standards, use of COTS hardware and software, and design and component reuse. Openness, particularly when used in conjunction with modularity and reuse, is frequently associated with cost avoidances from several sources, including: more efficient design and design refresh; increased opportunity to multi-source parts; reductions in obsolescence, counterfeit, and other part supply chain disruption management overhead; more efficient innovation and technology insertion; and modularization of qualification. However, use of an open systems approach requires investment and increases risk exposure because the enterprise has little or no control over the definition and supply chain of COTS or open components, and these components by nature are more volatile and have short procurement lives.

While the concept of openness is intuitively understood, quantifying a system's degree of openness and placing a value on that openness is difficult.

Although several previous efforts have been initiated to assess and compare openness, they have generally relied on qualitative measures, and none address the issue of cost.

The model developed in this thesis quantitatively determines the effects of openness on life cycle cost of electronic systems by identifying metrics to measure openness and cost drivers that can be used to predict life cycle cost. Connections between these were identified and a model for the life cycle cost difference between different designs/implementations was created. By comparing similar systems of disparate openness, this model can be used to determine the life cycle cost impact of openness.

The model was applied to a case study of the U.S. Navy's ARCI Sonar system. Simplified versions of two architectures with different openness levels, corresponding to different implementation phases of the ARCI system, were analyzed. The case study was conducted for support lives between 10 and 30 years. The study demonstrated that for support times longer than approximately 20 years, or in cases where the EOS date is uncertain, significant cost avoidance could be realized by implementing a more open design supported using a bridge buy and periodic design refreshes. However, for support lives shorter than 20 years, or in cases when a small number of system instances are to be deployed, the more closed architecture, managed using a lifetime but methodology, leads to greater cost avoidance.

4.2: Contributions

This section summarizes the contributions made by this thesis.

4.2.1: Formulation of the first quantitative model for assessing the cost impacts of openness

The first quantitative model for assessing the cost impacts of openness was developed. Previous models are either qualitative or do not address cost impacts. The model is time-dependent, and able to handle uncertainty in input parameters. The model utilizes a novel cost-difference approach that minimizes the amount of data required and isolates the portion of cost avoidance that is directly attributable to the use of open systems.

4.2.2: Demonstrated that more openness is not always better

The model and ARCI case studies demonstrated that there are scenarios in which increasing the openness of an architecture will increase life cycle costs. The case study also provides the first known quantitative support for Abts' COTS-LIMO hypothesis that increasing CFD increases cost avoidance. However, these results challenge Henderson's implicit assumption that marginal openness is always positive (increasing openness is always beneficial).

4.2.3: Established that the value of openness depends on several external factors

The ARCI case study establishes that, in addition to the architecture itself, the value gleaned from using an open system depends on several external factors. Specifically, the number of system instances to be fielded, the number of years those systems are to be supported, and the likelihood of those systems to be life-extended.

4.3: Future Work

The model developed and tested is the first of its kind. Many improvements to the model could be made:

4.3.1: Capture more benefits of reuse

The current model accounts for reuse of components within a design, but cannot account for cost savings due to reuse between different architectures, or for reuse at the subsystem level.

4.3.2: Collect data for calibration and validation

The current model depends on many cost parameters that are assumed to be enterprise dependent. Access to historical data would allow for more accurate calibration and validation of the model.

Additionally, though the current model takes procurement life as an input distribution, this could be predicted from other input metrics, including market share, number of competing components, volatility, and years of use. This would require historical on each type of hardware and software component used.

4.3.3: Capture the risks associated with early adoption

When new technologies are first implemented, they tend to be more unstable, with more frequent and significant updates. While operating on the "bleeding edge" can help improve performance and add new functionalities, it also increases the risk of breakage, architectural mismatch, and expensive retrofits.

4.3.4: Further test Abts' and Henderson's Hypotheses

Further work is needed to test if COTS equilibrium value proposed by Abts' COTS-LIMO model exists, and where. Subsequent analysis could focus on the effects of different CFD values for systems above and below the equilibrium point. A generalization of this model should be used to generate a continuous cost versus openness curve, which can be used to measure the utility of openness.

Appendix A: OAAT (and MOSA PART) Questionnaire

Sections A.1 through A.9 below are taken from [17].

MOSA PART consists of the questions listed in section A.1 (MOSA PART Programmatic) and section A.3 (MOSA PART Technical). For MOSA PART, all questions are given equal weighting.

The OAAT consists of all the questions below. Equal waiting is not given to all questions. "Key" questions are weighted between 3 and 4 times stronger than other questions, and an answer is required (the "not applicable" response is disabled). "Litmus" questions limit the maximum score, even if the responses to all other questions would suggest a higher score.

A.1: MOSA PART: Programmatic

A.1.1 To what extent is MOSA incorporated into the program's acquisition planning?

A.1.2 To what extent did the program plan for its implementation of MOSA?

A.1.3 To what extent is the program's MOSA implementation based on systems engineering principles and processes?

A.1.4 To what extent are responsibilities assigned for implementing MOSA?

A.1.5 To what extent is the program staff trained on, or have relevant experience in MOSA concepts and implementation?

A.1.6 To what extent does the program's configuration management process encompass changes to key interfaces and corresponding standards?

A.1.7 To what extent have program requirements been analyzed, and refined as needed, to ensure that design-specific solutions are not imposed?

A.1.8 To what extent do the system level functional and performance specifications permit an open systems design?

A.1.9 To what extent are modular, open system considerations included as part of alternative design analyses?

A.1.10 To what extent are mechanisms established to migrate key interfaces that are proprietary or closed to key interfaces that are open?

A.1.11 To what extent are MOSA principles reflected in the program's performance measures?

A.2: OAAT: Programmatic

A.2.1 **(KEY)** To what extent does the program have policies and processes that control adding specifications, options, or extensions that limit the use of widely-supported or openly available interface standards?

A.2.2 **(KEY)** To what extent are Non-mission unique capabilities supplied using either components reused from other programs or available from the commercial market?

A.2.3 **(KEY)** To what extent have the proprietary or unique non commercial elements been limited or well defined such that they do not hinder other developers from interfacing or developing any part of the system?

A.2.4 NOTE: As of OAAT Version 1.1, the MOSA PART questions have been combined with the OAAT questions and this question has been obviated. This question is no longer part of the OAAT question set. To what degree has the Modular Open Systems Approach (MOSA) been implemented in the programmatic and business aspects of the Program (organization)? Enter the results from the Business Section of the OSJTF MOSA PART.

A.2.5 To what extent is the Program complying with the Joint Capability Integration and Development System (JCIDS)?

A.2.6 To what degree is the Program complying with the Interoperability and Supportability requirements for National Security Systems in references like CJCS 6212.01C and DoDD 4630.5 and DoDI 4630.8?

A.2.7 To what extent does the Program plan, directive documentation and funding enable orderly migration of proprietary or program unique system modules to open system alternatives when capabilities are upgraded?

A.2.8 To what extent is the program free of system components that have proprietary characteristics, restrictive licensing or prohibitive cost that could limit or preclude the reuse of the components in other Navy Systems or the competitive selection or re-assignment of those components to other vendors?

A.2.9 To what extent has the Prime System Integrator established processes that facilitate flexibility of task assignment, competition of individual tasks, or re-competition of tasks?

A.2.10 **(KEY)** To what extent has the program established and maintained a repeatable, non-restrictive process that discloses in-process design documentation and software tools information directly to third party developers?

A.2.11 To what extent is design documentation disclosed to interested parties from the beginning of the development effort?

A.2.12 **(KEY)** To what extent does the Program documentation stress the use of widely-accepted and supported standards, such as those maintained by recognized organizations (e.g. IEEE), to define both internal and external interfaces?

A.2.13 **(KEY)** Does the Program Plan and directive documentation include a data management strategy ensuring that when the Government exercises its intellectual property rights to obtain any developmental artifacts for anything it paid to develop with either complete or partial funding the Contractor can at most charge a nominal fee covering the marginal cost of the effort to provide that documentation?

A.2.14 **NOTE: As of OAAT Version 3.0, this question is no longer part of the OAAT question set.** To what degree does the Program (Organization) planning and directive documentation (e.g. implementation road map, Acquisition strategy) implement and comply with FORCEnet policies, standards and definitions as indicated by FORCEnet Consolidated Compliance Checklist (FCCC) or other FORCEnet compliance metrics?

A.2.15 To what extent does the program plan include Life cycle Support and Funding for OA elements?

A.2.16 To what extent has the program worked with the applicable Tech Warrant holder or equivalent authority to develop OA-specific metrics as part of its program management processes and reviews?

A.2.17 **(KEY)** To what extent do the program's software selection criteria require that, other things being equal, priority be given to software components/modules/systems that have the least restrictive rights associated with them?

A.2.18 **(KEY)** To what extent does the Program use MOSA- and OA-specific language or contractual provisions in its acquisition and development documentation? Examples of these types of document include RFI and RFP materials, Contracts, Acquisition Plan, System Engineering Plan, and Information Support Plan.

A.2.19 To what extent does the Program use MOSA- and OA-specific language or contractual provisions in its acquisition and development documentation? Examples of these types of document include RFI and RFP materials, Contracts, Acquisition Plan, System Engineering Plan, and Information Support Plan.

A.2.20 To what extent does the Program's documentation provide for cost-effective incremental upgrades without dependencies on a single source or the need to redesign large portions of the system?

A.2.21 To what extent has the Program organization implemented a training program to educate their work force on OA related policy and concepts (e.g., OA/MOSA, FORCEnet, Interoperability, JCIDS and open systems concepts)? Ways to do this could include leveraging available courses, web-based materials, required reading, internal meetings/seminars, etc.

A.2.22 To what extent has the Program used incentives to promote modular designs, commonality and component reuse?

A.2.23 To what extent does the Program's configuration management process use integrated teams such as “communities of interest” to identify how individual changes impact the system’s interfaces and information exchange standards?

A.2.24 **(KEY)** To what extent are multiple third parties directly contracted to develop components of the System, giving the government the flexibility to compete or reassign component development?

A.2.25 **(KEY)** To what extent do the Program's market research and selection processes use criteria that favor commercial, common enterprise wide, or generally accepted interface and information exchange standards?

A.2.26 Does the Program's Acquisition Strategy, contract language and funding profile facilitate subsequent assignment of major tasks and program roles to alternate providers at predetermined intervals?

A.2.27 **(KEY, LITMUS)** To what extent are market research, community of interest teams, peer review groups, or alternative forums used to assess and select among available capability improvement options?

A.2.28 To what extent does the program develop POM Issue Papers or other business planning documents (such as business case analyses) to address OA business and technical issues?

A.2.29 **(KEY)** To what extent does the Program reuse components from other government programs?

A.2.30 To what extent can the program accommodate software tools or other components from sources other than the prime system integrator or existing vendors without requiring significant modifications?

A.3: MOSA PART: Technical

The first section of the technical questionnaire consists of 13 questions adopted directly from the MOSA PART.

A.3.1 To what extent is the system's architecture based on related industry or other standard reference models and architectural frameworks?

A.3.2 To what extent is an architectural description language used to define system modules and interfaces?

A.3.3 **(KEY, LITMUS)** To what extent does the system's architecture exhibit modular design characteristics?

A.3.4 **(KEY)** To what extent is the system's architecture capable of adapting to evolving requirements and leveraging new technologies?

A.3.5 To what extent has the criteria for designating key interfaces been established?

A.3.6 To what extent has the program designated key interfaces?

A.3.7 To what extent has the program assessed the feasibility of using open standards for key interfaces?

A.3.8 To what extent have standards selection criteria been established that give preference to open interface standards?

A.3.9 **(KEY, LITMUS)** To what extent are open standards selected for key interfaces?

A.3.10 To what extent are validation and verification mechanisms established to assure that system components and selected commercial products conform to the selected interface standards?

A.3.11 **(KEY)** To what extent do system components and selected commercial products conform to standards selected for system interfaces?

A.3.12 To what extent do system components and selected commercial products avoid utilization of vendor-unique extensions to interface standards?

A.3.13 **(KEY, LITMUS)** To what extent can system components be substituted with similar components from competitive sources?

A.4: Design Tenet: Interoperability

A.4.1 The Unit of Assessment predominantly complies with what type of interoperability standards?

A.4.2 NOTE: As of OAAT Version 3.0, this question is no longer part of the OAAT question set. (KEY) How standards-based is the Unit of Assessment's data model?

A.4.3 What is the scope of the data model that the Unit of Assessment uses to support interoperability with other systems?

A.4.4 **(KEY)** What is the scope of interoperability of the Unit of Assessment?

A.4.5 To what extent does the Unit of Assessment use mechanisms to discover and invoke services?

A.4.6 To what extent does the Unit of Assessment support mechanisms for service discovery and invocation?

A.5: Design Tenet: Maintainability

A.5.1 **(KEY)** What architectural characteristics address obsolescence and provide for timely technology refresh, fixes, and upgrades?

A.5.2 Do the unit of Assessment's technical artifacts provide sufficient detail and scope for maintenance?

A.6: Design Tenet: Extensibility

A.6.1 Does the program follow a well defined System Engineering process for implementing capability extension?

A.6.2 **(KEY)** Will the technical infrastructure accommodate extensibility of the Unit of Assessment's functionality?

A.6.3 What is the scope of testing needed after new components are added to the Unit of Assessment?

A.7: Design Tenet: Composability

A.7.1 To what extent are the components of the Unit of Assessment implemented and independently deployable as packages?

A.7.2 To what extent can the functional capabilities of the Unit of Assessment be re-combined or re-arranged to support a modified process/workflow/mission?

A.8: Design Tenet: Reusability

A.8.1 **(KEY, LITMUS)** What reuse strategy is used within the Unit of Assessment?

A.8.2 What is the scope of the set of processes used to identify and evaluate reuse candidates for incorporation into the Unit of Assessment?

A.8.3 **(KEY)** Which approach best describes the operational run-time infrastructure supporting the Unit of Assessment?

A.8.4 Have the commonalties and variations of the Unit of Assessment been specified to facilitate reuse congruent with a broader Software Product Line?

A.9: Design Tenet: OSJTF – MOSA

A.9.1 NOTE: As of OAAT Version 1.1, the MOSA PART questions have been combined with the OAAT questions and this question has been obviated. This question is no longer part of the OAAT question set. What is the level of MOSA compliance for the Unit of Assessment?

Appendix B: OAAM Version 1.0

Taken from [70]

Business and Acquisition Characteristics

0 – Isolated
- Exclusive use of closed sole source contracts
- Proprietary interface, no access to systems

1 – Connected
- Initial OA language in contracting and acq docs
- Program (gov't/industry) educated on FORCEnet/OA
- Initial use of commercial standards and best practices
- Program has achieve "Marginal" level for MOSA business indicators

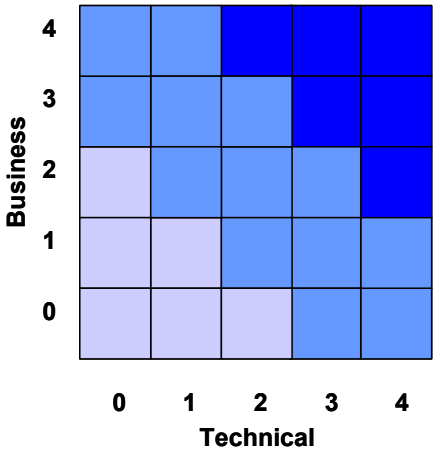
2 – Migrating to Openness
- Program has validated NR-KPP
- Transitioning to JCIDS capability needs documents
- Contracting approach maximizes cost competitiveness and innovation
- Use of commercial standards based COTS products
- Some market research employed to leverage commercial investment
- Completed FIBL Survey and verified information
- Program has achieve "Satisfactory" level for MOSA business indicators

3 – Common
- Spiral development/evolutionary acquisition employed to facilitate rapid technology insertion
- Applicable program acquisition and engineering documentation (AS, SEP, ISP, etc) includes OA language
- Integrated team approach to development involving requirements, resource, testing, user community members
- "Community of Interest" teams employed to develop system
- Program has robust FORCENet/OA implementation roadmap

4 – Open and Net-Centric
- OA compliance metrics part of PM processes and program reviews
- Extensive use of commercial standards and best practices across Enterprise
- Program conducts continuous market research
- Continuous process for FORCENet/OA improvement
- Program has achieve "Exemplary" level for MOSA business indicators

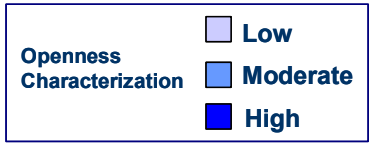
OA Assessment Model

Version 1.0 (8 March 2005)



Business and Acquisition Strategy Characteristics refer to the processes & documentation programs employ to acquire and manage systems;

Architecture and Technical characteristics are the technical features of computing environments and application software



Architecture and Technical Characteristics

0 – Closed
- Proprietary Hardware or API (O/S or middleware)
- Predominant number of point to point legacy interfaces
- Highly integrated applications with integral middleware

1 – Layered
- Standards-based COTs Hardware & O/S
- Specialized middleware
- Highly integrated monolithic applications isolated from Computing environment
- Standard communications between layers
- Program has achieve "Marginal" level for MOSA technical indicators

2 – Layered & Open
- Computing Environment / App. S/W independence
- Open published APIs
- Modular application components
- Facilitates technology insertion/replacement
- Standard communications between layers
- Exposes data to network via I/Fs to legacy system/subsystems
- Separates operator, application, and data
- Program has achieve "Satisfactory" level for MOSA technical indicators

3 – Common
- Discovers/publishes capability using standards (where applicable)
- Adheres to a common architecture across multiple programs
- Uses common services (such as security)
- Common semantics and data model
- Ability to Interact with GIG/FORCENet

4 – Enterprise
- Adheres to a common architecture across multiple domains
- Data exchange between domains via std interface
- Commercially accepted services or data model
- Uses core services (e.g., NCES, DIB)
- Exposes services and data to GIG/FORCENet
- Program has achieve "Exemplary" level for MOSA technical indicators

Appendix C: MOSA Metrics Calculator

The following is adapted from the AFRL/RYM Metrics Working Groups' MOSA Metrics Calculator [58].

C.1.1 Component Name:

C.1.2 What is the component type? (Hardware or Software)

C.1.3 Is the component scalable? (Yes/No)

C.1.4 Is the component upgradeable? (Yes/No)

C.1.5 Has the component's extensibility been considered? (Fully/Somewhat/None)

C.1.6 This component is part of which function?

C.1.7 Is the component available from multiple vendors? (Yes/No)

C.1.8 Is the component domestically produced? (Yes/No)

C.1.9 Does the component have a plan for continued support through the life cycle of the system? (Yes/No)

C.1.10 Does this component have a key interface? (Yes/No)

C.1.11 Is a proprietary standard used for the key interface? (Yes/No)

C.1.12 Is the standard used for the key interface well defined? (Yes/No)

C.1.13 Does the component use a proprietary standard? (Fully proprietary/Partially open/Fully open)

C.1.14 Does this component include a test plan? (Yes/No)

C.1.15 What impact does the component have on information assurance? (Significant/Some/None)

Appendix D: Cost of Openness Tool: Quick-Start Guide

The following manual is a brief introduction to the Cost of Openness Tool and its user interface. It covers all of the inputs to be entered by the user, and briefly discusses the functionality of the tool.

The Cost of Openness Tool has been implemented in Microsoft Excel, using VBA and various form inputs. For best results, it should be used on a PC running Excel 2007 or later. The Cost of Openness Tool contains the following sheets:

1. Standards
2. Components
3. One or more Architecture sheets
4. Parameters
5. Weights
6. Simulation
7. One or more Results sheets

When setting up and running a simulation, these sheets should be completed in this order to ensure that the data is properly input into the model. Each sheet is further explained below.

Important note about distributions: On the Standards and Components worksheets, some of the data is expected to be input as a distribution. In those cases, the affected cells are locked to prevent user input. To input the desired information, click on the "D" (for "Distribution") button located on the right side of the appropriate cell. This will bring up a dialogue that can be used to define the distribution. Select the desired distribution from the "Distribution Type" drop-down, and input the required parameters. Select "OK" to input the data, or "Cancel" to abort. Selecting the "None" distribution type allows a constant value to be used.

D.1: Standards

Each row on the Standards sheet represents a standard that is to be used in one of the Architectures to be modeled. Extra, unused Standards may also be included on this list. New standards may be added by pressing the "Add Standard" button; old standards may be removed by pressing the corresponding "Remove" button. Each standard is required to have:

1. A unique name
2. A procurement life, which tells the time between the standard's introduction and obsolescence (entered as a distribution)
3. A life code, which tells where the standard is in its life cycle (a value between 1 and 6, 1 denoting introduction, 6 denoting obsolescence)

Other information about the standard may also be input, including:

4. Market share, the percentage of the relevant market that uses the standard
5. The volatility of the standard, measured in expected number of minor changes or updates per year
6. The years of use, or how long the standard has been available
7. Enterprise Experience, the length of time, in years, that the enterprise has used the standard
8. "Standard Defined By", selected from:
    a) "In-house proprietary standard": a proprietary standard that is maintained by the enterprise itself
    b) "Defined by a single entity or group, with restrictions": a proprietary standard belonging to an external entity
    c) "Defined by single entity, no licensing restrictions": typical classification of a de facto standard

d) "Defined by group, no licensing restrictions": an open standard

D.2: Components

Each row on the Components sheet represents a component that is to be used in one of the Architectures to be modeled. Extra, unused components may also be included on this list. New components may be added by pressing the "Add Component" button; old components may be removed by pressing the corresponding "Remove" button. Each component is required to have:

1. A unique name
2. A type – Hardware, Software, or Consumable. Consumables are treated as hardware, except that a constant failure/replacement rate is assumed, instead of sampling from the failure distribution.
3. An Interface Standard, selected from those defined on the Standards worksheet
4. A Procurement cost (entered as a distribution)
5. A time-to-failure distribution
6. A procurement life, which tells the time between the component's introduction and obsolescence (entered as a distribution)
7. A life code, which tells where the component is in its life cycle (a value between 1 and 6, 1 denoting introduction, 6 denoting obsolescence)
8. Accessibility of the component's technical data package (TDP), selected from:

a) "Component designed in-house": A component that is designed and maintained by the enterprise itself. This is an option for components that use an in-house or open (unrestricted) standard.
b) "Limited access (Proprietary/COTS)": A component designed and maintained by an external entity. This may be a component that is created by a contractor for this specific purpose (proprietary), or a COTS component. In either case, the component is treated as a "black box" – the enterprise cannot access internal specifications or workings of the component for debugging or modification purposes. This option cannot be selected if the component uses an "in-house" standard.
c) "Full access (COTS/Open Source)": A component that is designed and maintained by an external entity, but to which the enterprise has access to the TDP and internal workings of the component, facilitating debugging, and allowing for some modifications. This may be selected for components that use "unrestricted" standards.

9. A Design Refresh Plan (DRP) option, selected from:

a) "Included in DRP": Components that are included in design refresh planning.
b) "Excluded from DRP": Components that are excluded from design refresh planning, but which are refreshed at the first opportunity after obsolescence.
c) "Not refreshable": Components that are excluded entirely from refresh planning and implementation – if the component becomes obsolete, a lifetime buy strategy is used, regardless of the strategy used for the rest of the system.

Other information about the component may also be input, including:

10. Cost of annual licensing, where applicable (typically only applicable to some software)
11. Software size, given in thousands of lines of source code (KSLOC)
12. The number of competing components
13. The volatility of the component, measured in expected number of minor changes or updates per year
14. The years of use, or how long the component has been available
15. Enterprise Experience, the length of time, in years, that the enterprise has used the component

D.3: Architecture Worksheets

The architecture worksheets contain the information about the actual architectures to be compared. While the model can be run with only one architecture sheet, typically two or more architecture sheets are necessary. Additional architecture sheets can be added by pressing the "Add New Architecture" button on the Simulation worksheet. This brings up a dialogue that allows the user to choose between adding a blank architecture and duplicating any existing one. Extra, unused architectures may be left in place while the tool is being used, or may be deleted by right-clicking the appropriate tab, and selecting "delete". To ensure more accurate results, all architectures should be of similar functionality and described with the same level of detail.

An architecture is defined by a hierarchy of subsystems and components. Each layer in the hierarchy is given a level number and a number of instances. Low level subsystems are made up of higher level (more detailed) subsystems and components. The instance number tells how many times the subsystem appears in the architecture. The lowest (outermost) subsystem (typically the zero-level) must represent one instance of the system that the architecture is defining, and should thus have an instance number of 1. Extra empty lines may be inserted between subsystems for visual clarity.

In addition to a level and instance number, each level is given a name. For subsystems, any useful identifying name may be used. For the zero level, this is typically the name of the architecture, while for other subsystems it may be the name of the functional group it represents. At the component level, this name *must* be the same as the name input on the components worksheet. When a name that matches a component defined on the "Components" worksheet is entered, the "Is Component" column will display "TRUE" and the appropriate interface standard will appear in the "Standards Type" column. Otherwise, these cells will remain blank.

Once a component has been input, additional data specific to that application of the component may be input. These include:

The up-rating and modification costs per unit (hardware components)

The amount of glue code required to integrate the component (software components)

The percentage of total functionality used

The number of functional interfaces the component has, both with components in the current subsystem, and with components in other subsystems.

Interdependent Components: Collections of components may be interdependent, such that when one becomes obsolete, they all need to be refreshed as a unit. This may be true in the case

of a specialized interface, or of a hardware-software pair that are interdependent. This is may be modeled by placing any interdependent components into the same subsystem,[51] and setting the "Subcomponents Interdependent" option on the subsystem to "True".

Partial Requalification: On some architectures, refreshed components and subsystems may be re-qualified without needing to re-qualify the whole system. When this is the case, subsystems and components that may be re-qualified independently should be marked as such by setting the "Requalification Possible" option to "True". The whole system (zero-level) should always be marked as "True", though changing it to blank or "False" will not change the results obtained. (Architectures that absolutely may not be re-qualified should be run using the "Lifetime Buy" setting found on the "Simulation" worksheet.

An example architecture is shown below. Note that the zero-level system is the architecture itself, and it has an instance number of 1. The architecture is made up of two cabinets, and each cabinet contains two drawers. Extra spaces are used in Cabinet 2 for visual clarity; these are optional, and may be omitted as they were in Cabinet 1.

Weibull(1.75,2, 0) represents a Weibull distribution with shape parameter 1.75 and scale parameter of 2 units (years, in this case). Normal(5,1) represents a normal distribution with mean 5 and standard deviation 1.

[51] And no other components.

Standards Sheet:

Name	Procurement Life (years)	Life Code	Market Share	Volatility	Years of Use	Enterprise Experience (years)	Standard Defined By
Standard1	Normal(25,5)	3	10.00%	1	5	5	Defined by entity or group, with licensing restrictions
Standard2	Normal(25,5)	3	60.00%	1	10	1	Defined by group, no licensing restrictions

Components Sheet:

Name	Type	Interface Standard	Procurement Cost ($)	Annual Licensing	Size in KSLOC (SW only)	Time to Failure (years)	Procurement Life (years)	Life Code	# Competing	Volatility	Years of Use	Enterprise Experience (years)	Accessibility of TDP	DRP Calculations
Component1	Hardware	Standard1	100			Weibull(1.75,2,0)	Normal(5,1)	3	2	1	5	5	Limited access (Proprietary/COTS)	Included in DRP
Component2	Software	Standard1	150			Weibull(1.75,2,0)	Normal(5,1)	3	2	1	5	5	Limited access (Proprietary/COTS)	Included in DRP
Component3	Hardware	Standard2	125			Weibull(1.75,2,0)	Normal(5,1)	3	3	1	10	1	Full access (COTS/Open Source)	Included in DRP
Component4	Software	Standard2	200			Weibull(1.75,2,0)	Normal(5,1)	3	6	1	10	1	Full access (COTS/Open Source)	Included in DRP

Architecture Sheet:

Level	Name	Is Component	Requalification Possible	# Instances	Standards Type	Uprating /Modification cost/unit (HW)	KSLOC of glue code required (SW)	% Functionality Used	Number of Interfaces	Subcomponents Interdependent
0	Arch 1	False	True	1						False
1	Cabinet 1	False	True	1						False
2	Drawer 1A	False	True	3						False
3	Component1	True	False	2	Standard1	0		75	1	
3	Component2	True	False	1	Standard1	0		75	1	
2	Drawer 1B	False	True	2						False
3	Component1	True	True	1	Standard1	0		50	1	False
1	Cabinet 2	False	True	2						False
2	Drawer 2A	False	False	1						False
3	Component1	True	False	1	Standard1	10		75	1	
2	Drawer 2B	False	False	2						True
3	Component3	True	False	2	Standard2	0		100	2	
3	Component4	True	False	2	Standard2	0		100	2	

D.4: Parameters

The parameters page is used to set many of the parameters and baseline costs used in the model. These are typically defined by the enterprise, so when running many simulations, they may often be set to the desired value, and then left alone. The input parameters are as follows:

1. Base cost to qualify a standard for use by the enterprise
2. Base cost to qualify a hardware component for use in a system
3. Base cost to qualify a software component (per KSLOC) for used in a system
4. Base cost to qualify an overall system design, to verify the architecture will perform as required
5. Base cost of an outgoing test, to verify that the system is functioning properly
6. Hardware diagnosis and replacement cost ratio – the cost to identify and replace a failed component, expressed as a proportion of the component cost.
7. Base cost to install a software update (per KSLOC)
8. Base cost to maintain software (proprietary component or glue code, per KSLOC)
9. Software update implementation frequency (per year) – how often bug fixes are sent to the field
10. Annual discount rate, used to move all cost calculations into year 0 dollars

11. Inventory support cost ratio – the cost to support hardware spares in inventory for one year, expressed as a proportion of the component cost
12. Spares overbuy percentage – the number of extra spares to stockpile in the event of obsolescence
13. Spares overbuy penalty ratio – the cost to dispose of unneeded spares that remain in inventory
14. Spares underbuy penalty ratio – the added cost to procure a component that has gone obsolete in the event that too few spares were purchased
15. Base cost to initialize a design refresh
16. Base cost to design a hardware component
17. Base cost to design a software component
18. Base cost to integrate a hardware component
19. Base cost to integrate a software component (per KSLOC)
20. Life Code for replacement Standards
21. Life Code for replacement Components

D.5: Weights

The weights worksheet allows for easy re-weighting of the metrics, and can be used to fine-tune or calibrate the model. Once appropriate values are found, they should be left alone. If historical data is available, and used to calibrate the model, these

parameters should be locked down to avoid further changes. This can be done by hiding the sheet.

Each of the openness metrics input on the Standards, Components, and architecture worksheets is represented on the weights worksheet. Each metric has five inputs, the minimum and maximum cost factor associated with the metric, the metric's lower and upper bounds, and the nominal metric value. The minimum and maximum cost factors represent the lowest and highest possible values of cost factor (CF). The metric lower and upper bounds are used to shape the CF curve, and generally correspond to the smallest and largest expected value of the input metric itself. The metric may, at times, fall outside this range. The nominal value is the value used for the metric in the event that the input is left blank,[52] and represents a typical value of the metric.

Depending on the metric, the cost factor associated with an openness metric's input value is obtained using either a linear, logistic, or asymptotic curve. Linear curves begin at the minimum value and increase at a constant rate until reaching the maximum value. Logistic curves (also called "S" curves) grow slowly at first, fastest in the middle, and then taper off again. Asymptotic curves begin at a minimum value, but maximum growth rate, and taper off until it reaches the maximum value (and minimum growth rate).

Some important notes:

[52] A blank and a value of zero are treated separately.

1. All openness metric values *must* be non-negative. Most are positive, though some may be zero. The nominal metric value, therefore, must also be non-negative.
2. The metric lower bound, however, may be negative.
3. In all cases, the cost factor (CF) is greater than zero. This means that the minimum CF and maximum CF must be strictly positive (>0).
4. Metric values do not need to be between the lower and upper bounds (though they *must* be non-negative).
5. Even if a metric value falls outside the range [lower bound, upper bound], the resulting CF value will fall in the range [minimum CF, maximum CF].
6. When the metric and the CF are positively correlated, metric values near the lower bound will give a CF close to the minimum value, and metric values near the upper bound will give a CF close to the maximum value. When the metric and the CF are inversely correlated metric values near the lower bound will give a CF close to the maximum value, and metric values near the upper bound will give a CF close to the minimum value.

D.6: Simulation

The simulation sheet is used to define the scenario to be used to run the simulation. These settings include the number of instances of the systems being

modeled, the number of trials to run, the number of years of service, and the production and retirement schedule. These inputs are explained below.

1. Number of system instances: The number of systems to be modeled is a calculated value, and should not be entered directly. Instead, the production schedule should be altered to reflect the number of systems produced in each year of the simulation. The value in cell B2 will update automatically.
2. Default end of service year: The number of years that the simulation is run. For the "Range" run type, this is the base year from which offsets are added/subtracted as appropriate. For the "Life Extended" run type, this is the base year from which the number of years of life extension is measured. Notice of life extension is given some number of years before the default end of service.
3. Number of trials: The number of times the simulation is to be run. More trials take longer, but give a better picture of how the system is expected to behave.
4. Retirement Offset (Min and Max): Used in the "Range" run type, to set the first and last year EOS year to be simulated. The values may be positive or negative, but the minimum value must be less than the maximum value.
5. Maximum life extension: Used in the "Life Extended" run type to determine the maximum number of years that the system may be life extended. This must be a positive value.

6. Life extension notice: The number of years before the default EOS date at which the system is life extended. Before this point, all calculations assume the system is to be retired at the EOS date. Afterward, the life-extended EOS date is used. This must be a positive value.
7. Production and retirement schedules: Used to tell how many systems are produced and retired in each year. Any unretired systems are supported until the EOS date.
8. Number of system types: Used to run multiple simulations at once. Select a number between 2 and 5.
9. System Name: A unique name given to each type of system (architecture and support methodology) being run. When different architectures are being compared, this may be the architecture name.
10. System architecture: The architecture sheet that defines the system being modeled. Architectures to be compared should be of similar functionality, and described at the same level of detail.
11. Obsolescence mitigation strategy: How obsolete components are supported. Options are to perform a Lifetime buy, a bridge buy (as defined by the user or by the DRP model), or to immediately find a replacement.
12. Component refresh period: If bridge buy is selected, this tells how often design refreshes are conducted.

13. Upgrade period: After a design refresh, the deadline by which all active systems must be upgraded to the newest version.
14. For repairs between refresh and upgrade: Once an obsolete component has been refreshed, failed instances of the obsolete component may be replaced by the old (obsolete, but stockpiled) component, or by the current version of the component.

There are three run types. The first, most basic run type is the "Single EOS" run, where the simulation is run once for each trial, using the production and retirement schedules as given. This type of run gives an understanding of which architecture is more expensive to support, and how the difference in support costs between the two architectures changes over the life of the system.

The "Range EOS" runs the simulation several times for each trial, using the production schedule as given, but using a range of end of service dates. For this type of run, the retirement schedule is shifted forward or backward by the number of years as defined by the retirement offsets. This gives a picture of how sensitive the relative support costs of the two architectures are to their support life. This type of run can be used to demonstrate that while one architecture is less expensive for relatively short support lives, the other architecture may be preferable if the support life exceeds some value.

The final run type is the "Life Extended" run. In a life extended run, the system is initially supported for retirement at the EOS dat. At some point ("Life Extension Notice" years before the EOS date), the system is life-extended, and must

be supported until a later date. This is used to examine the impact of life-extension on the total life cycle cost. While one architecture may be less expensive to support until the originally scheduled EOS date, it may be desirable to use a different architecture if there is a high likelihood that the EOS date will be pushed later during the support life.

D.7: Results Sheets

The Results sheets are where the results of the simulation are displayed. Currently, this is a graph that depends on the run type used. Single EOS gives a graph showing cumulative difference in cost over time, as well as a pie-graph categorizing the primary cost differences between the two systems. Range EOS and Life Extended runs give a graph that shows total cost difference for support until a given year.

Histograms are available to give a cross-sectional view of the different trials. This can be used to determine the confidence that the cost difference will be greater or less than a specific value.

Glossary

API – Application Programming Interface. Defines how software components interact with each other.

ARCI – Acoustic Rapid COTS Insertion. An effort to modernize an improve a Navy sonar system through the use of COTS components

CBS – COTS Based System. A system that employs COTS components to provide most, if not all, of its required functionality

CFD – COTS Functional Density. The average percentage of functionality provided by each COTS component.

COCOMO – Constructive Cost Model. An parametric model for software cost estimation developed by Barry Boehm (University of Southern California).

Component – Any part, be it hardware, software, or a combination of the two, which is procured or designed as a single unit

COTS – Commercial Off-The-Shelf

De-facto Standard – A standard that has become popular or dominant after being selected by the market, as opposed to being officially approved.

DMSMS – Diminishing Manufacturing Sources and Material Shortages

DoD – Department of Defense

DRP – Design Refresh Plan. A design refresh plan for a system details the number of refreshes to be conducted, the timing of those refreshes, and components effected by each refresh.

Enterprise - An entity that defines and maintains a system design, and deploys one or more instances of that system to fulfill operational requirements. An enterprise may maintain designs and provide operational support for several types of systems at the same time with similar or diverging purposes.

Interface Standard – a Standard that focuses on how a component interacts with its surrounding components, and not on the internal operation of the component.

Interoperability – The ability of systems and subsystems to work together

IP – 1) Intellectual Property. 2)Internet Protocol

Key Interface – Interfaces that pass the most important information between functionally adjacent components or subsystems, or an interface that connects a technologically volatile subsystem to a more stable one

Mission Critical – components or capabilities that are necessary for the successful completion of a mission [51]

MOCA – Mitigation of Obsolescence Cost Analysis. A methodology and accompanying software tool developed by Dr. Peter Sandborn and the Center for Advanced Life Cycle Engineering (CALCE) that can be used to model the effects of component obsolescence on a system's sustainment costs. Can be used to determine the optimum design refresh plan (DRP) for the system.

MOSA – Modular Open Systems Approach. Also used to mean Modular Open Systems Architecture, an architecture that is designed in accordance with MOSA principles.

MOSA PART – MOSA Program Assessment and Rating Tool

MpMe – Multi-Part Multi-Event. A simplified, less computationally expensive DRP model that can be used in place of MOCA. MpMe only considers DRPs that have a constant period.

OA – Open Architecture. An Open Architecture is any architecture that predominantly or exclusively employs open standards to define the interfaces between its components, particularly at Key Interfaces. The openness of an architecture varies with the type and number of standards used, and with the number of interfaces which use each of those standards. The openness of an architecture may be increased by switching to standards that are more open, and by utilizing fewer standards overall.

OAAM – Open Architecture Assessment Model. "The OAAM describes the business and technical characteristics of a program or system's open architecture

maturity. [It] is the precursor to the OA Assessment Tool (OAAT), which grew out of the need for a more rigorous assessment of program's openness than the OAAM could provide." [68]

OAAT – Open Architecture Assessment Tool. A software tool developed by the Naval Open Architecture Enterprise Team (OAET) to allow program managers to assess the level of openness of their program using. This assessment is done both in terms of technical openness, and the openness of the business plan.

OAET – Open Architecture Enterprise Team

Open Source – A philosophy and movement, particularly for software, based on the idea of universal free and unhindered access to a component's definition, use, and redistribution.

Open Standard – Well-documented, publicly accessible Standards that are either free to use, or licensed for a small fee or royalty.

Open System – Systems that employ Open Standards. See Open Architecture (OA)

OSA – Open Systems Architecture. See Open Architecture (OA). Also used to mean Open Systems Approach, a set of principles, and the deployment thereof, used to design systems with an Open Systems Architecture.

OSJTF – Open Systems Joint Task Force

Proprietary Standard – A standard defined and controlled by a private enteprise which places restrictions on its access and use.

RAND – Reasonable and Non-Discriminatory: a system in which IP is made available for use to everyone equally, and for reasonable cost. Also called Fair Reasonable and Non-Discriminatory (FRAND). Considered a type of open IP license, though less open than RF licensing.

RF – Royalty Free: a system in which IP is made available for use without any licensing fees or royalties. This is the most open type of IP license.

Safety Critical – Components or capabilities that are necessary to prevent human injury or loss-of-life

SDO – Standards Defining Organization. Also SSO.

SSO – Standards Setting Organization. Also SDO.

Standard – Also called a technical standard, a specification (usually hardware) or a protocol (software). A Standard is a formally defined methodology or structure for use by any number of components, particularly for an interface or other external feature that interacts with other components.

Subsystem – a combination of one or more components or lower-level subsystems that interact to perform a higher level of functionality.

System Architecture – any high-level electronic "system of systems" made up of some number of subsystems or components. Architecture – a hierarchical breakdown that details the functional location and interactions of each component in a system

TCO – Total Cost of Ownership

TRL – Technology Readiness Level

References

[1] N. H. Guertin and R. W. Miller, "A-RCI -- The Right Way to Submarine Superiority," *Naval Engineers Journal,* vol. 110, no. 2, pp. 21-33, 1988.

[2] Defense Standardization Program Case Studies, "Acoustic-Rapic Commerical-Off-The-Shelf Insertion," United States Department of Defense, Case Study 13 (DSP-CS-13), from http://www.dsp.dla.mil/, no date.

[3] Defense Standardization Program Case Studies, "Navy Self-Contained Breathing Apparatus: Market Resarch Yields Significant Results," United States Department of Defense, Case Study 1 (DSP-CS-1), from http://www.dsp.dla.mil/, no date.

[4] Defense Standardization Program Case Studies, "Hull Mechanical and Electrical Equipment Standardization Program," United States Department of Defense, Case Study 8 (DSP-CS-8), from http://www.dsp.dla.mil/, no date.

[5] J. W. Abbott, A. Levine and J. Vasilakos, "Modular/Open Systems to Support Ship Acquisition Strategies," in *American Society of Naval Engineers Day 2008 Proceedings*, Arlington, VA, 2008.

[6] Open Systems Joint Task Force, "Program Manager's Guide: A Modular Open Systems Approach (MOSA) to Acquisition," United States Department of Defense, Available online: http://www.acq.osd.mil/osjtf/pdf/PMG_04.pdf, September 2004.

[7] G. T. Logan, "The Modular Open Systems Appraoch (MOSA)," OSJTF presentation to the Executive Program Managers Course, from: acc.dau.mil/CommunityBrowser.aspx?id=37585, May 2004.

[8] M. Boudreau, "Acoustic Rapid COTS Insertion: A Case Study in Modular Open Systems Approach for Spiral Development," in *International Conference on Systems of Systems Engineering*, pages 1-6, San Antonio, TX, 16-18 April, 2007.

[9] J. M. Hanratty, R. H. Lightsey and A. G. Larson, "Open Systems and the Systems Engineering Process," Office of the Undersecretary of Defense, Acquisition and Technology; Open Systems Joint Task Force, 2002.

[10] L. Bass, D. de Niz, J. Hansson, J. Hudak, P. Feiler, D. Firesmith, M. Klein, K. Kontogianis, G. Lewis, M. Litoiu, D. Plakosh, Sc, S. Schuster, L. Sha, D. Smith and K. Wallnau, "Results of SEI Independent Research and Development Project," Software Engineering Institute, Carnegie Mellon University, (Technical Report CMU/SEI-2008-TR-017) July 2008. Available from www.sei.cmu.edu.

[11] P. Lewis, P. Hyle, M. Parrington, E. Clark, B. Boehm, C. Abts and R. Manners, "Lessons Learned in Developing Commercial Off-The-Shelf (COTS) Intensive Software Systems," Federal Aviation Administration (FAA) Software Engineering Resource Center (SERC) Report, 2000.

[12] B. Clark and B. Clark, "Added Source of Costs in Maintaining COTS-Intensive Systems," *Cross Talk, The Journal of Defense Software Engineering,* vol. 20, no. 6, pp. 4-8, June 2007.

[13] C. Abts, "COTS Based Systems (CBS) Functional Density - A Heuristic for Better CBS Design," in *COTS-Based Software Systems*, Orlando, FL, Springer Berlin Heidelberg, 2002, pp. 1-9.

[14] J. W. Abbott, "The Evolution of Modular Open Systems in Naval Ship Design: 1975 - 2010," in *ANSE Breakfast Seminar on Ship Design and Acquistion - Lessons Learned*, June 3, 2010.

[15] US Department of Defense, "The Defense Acquisition System," in *US DoD Directive Number 5000.1 (DODD 5000.1)*, United States Department of Defense, 2003.

[16] J. H. Lienhard, "No. 1252: Interchangable Parts," [Online]. Available: http://www.uh.edu/engines/epi1252.htm.

[17] Naval Open Architecture Enterprise Team, Open Architecture Assessment Tool,

Version 3.0, User's Guide, 2009.

[18] IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries, New York, NY: Institute of Electrical and Electronics Engineers, 1990.

[19] C. Roark and B. Kiczuk, "Open Systems: A Process for Achieving Affordability," *IEEE Aerospace and Electronic Systems Magazine,* vol. 11, no. 9, pp. 15-20, September, 1996.

[20] C. Jones, "Geriatric Issues of Aging Software," *Cross Talk, The Journal of Defense Software Engineering,* vol. 20, no. 12, pp. 4-8, December 2007.

[21] E. Maxwell, "Open Standards, Open Source, and Open Innovation - Harnessing the Benefits of Openness," *Innovations: Technology, Governance, Globalization,* vol. 1, no. 3, pp. 119-176, 2006.

[22] M. D. Petty and E. W. Weisel, "A Composability Lexicon," *Proceedings of the Spring 2003 Simulation Interoperability Workshop,* pp. 181-187, 2003.

[23] Defense Standardization Program Case Studies, "Joint Tactical Radio System," United States Department of Defense, Case Study 12 (DSP-CS-12), from http://www.dsp.dla.mil/, no date.

[24] Defense Standardization Program Case Studies, "The Virginia Class Submarine Program," United States Department of Defense, Case Study 15 (DSP-CS-15), from http://www.dsp.dla.mil/, no date.

[25] M. Bourdreau, "Acoustic Rapid COTS Insertion: A Case Study in Spiral Development," Naval Postgradate School Acquisition Research Program Case Study, Monterey, CA, 2006. Available at: www.acquisitionresearch.org.

[26] H. Livingston, "GEB1: Diminishing Manufacturing Sources and Material Shortages (DMSMS) Management Practices," in *Proceedings of the DMSMS Conference*, Jacksonville, FL, August 21-24, 2000.

[27] P. Y. Chau and K. Y. Tam, "Factors Affecting the Adoption of Open Systems: An Exploratory Study," *Management Information Systems Quarterly,* vol. 21, no. 1, pp. 1-24, March 1997.

[28] K. Krechmer, "The Meaning of Open Standards," in *38th Annual Hawaii International Conference on System Sciences*, 2005.

[29] ITU-T, "Definition of "Open Standards"," 2010. [Online]. Available: http://www.itu.int/en/ITU-T/ipr/Pages/open.aspx. [Accessed 10 2012].

[30] M. L. Marcus, C. W. Steinfield, R. T. Wigand and G. Minton, "Industry-Wide Information Systems Standardization as Collective Action: the Case of the US Residential Mortgage Industry," *MIS Quarterly,* vol. 30, no. Special Issue on Standardization, pp. 439-465, 2006.

[31] N. Aggarwal, Q. Dai and E. Walden, "Are Open Standards Good Business?," *Electron Markets,* vol. 22, pp. 63-68, 1 2012.

[32] A. Layne-Farrar, A. J. Padilla and R. Schmalensee, "Pricing Patents for Licensing in Standard-Setting-Organizations: Making Sense of FRAND Commitments," in *CEPR Discussion Paper No. 6025*, January, 2007.

[33] D. Geradin, "Standardization and technological innovation: Some reflections on ex-ante licensing, FRAND, and the proper means to reward innovators," in *Conference of Intellectual Property and Competition Law*, Brussels, June 2006.

[34] M. A. Lemley, "Intellectual Propery Rights and Standard-Setting Organizations," *California Law Review (online),* vol. 90, 2002.

[35] P. Chappatte, "FRAND Commitments - The Case for Antitrust Intervention," *European Competition Journal,* vol. 5, no. 2, pp. 319-346, August, 2009.

[36] Open Source Initiative, "Open Standards Requirement for Software," [Online]. Available: http://opensource.org/osr.

[37] B. Perens, "Open Standards: Principles and Practice," [Online]. Available:

http://perens.com/OpenStandards/Definition.html.

[38] G. Hagan, Glossary of Defense Acquisition Acronyms and Terms, 13th Edition ed., Fort Belvoir, VA: Defense Acquisition University, 2009.

[39] J. L. Contreras, "Rethinking RAND: SDO-Based Approaches to Patent Licensing Commitments," in *ITU Patent Roundtable*, Geneva, October 10, 2012.

[40] M. Glader, "FRAND licensing obligations and industry standards," in *Jevons Institute for Competition Law and Economics Conference on Antitrust and Intellectual Property*, London, May 2007.

[41] Open Source Initiative, "History of the OSI," 9 2012. [Online]. Available: http://opensource.org/history.

[42] B. Perens, "The Open Source Definition," [Online]. Available: http://opensource.org/osd.

[43] B. X. Chen, "How Lightning Tightens Apple's Control Over Accessories," The New York Times, 14 February 2013. [Online]. Available: http://bits.blogs.nytimes.com/2013/02/14/lightning-apple/.

[44] Defense Standardization Program Case Studies, "NAVSTAR Global Positioning System: A Military Standard Transforms Global Navigation," United States Department of Defense, Case Study 6 (DSP-CS-6), from http://www.dsp.dla.mil/, no date.

[45] Defense Standardization Program Case Studies, "Joint Precision Approach and Landing System," United States Department of Defense, Case Study 11 (DSP-CS-11), from http://www.dsp.dla.mil/, no date.

[46] Defense Standardization Program Case Studies, "Common Air Defense Interrogator," United States Department of Defense, Case Study 16 (DSP-CS-16), from http://www.dsp.dla.mil/, no date.

[47] Open Systems Joint Task Force, "Case Study of the U.S. Army's Intelligence and

Electronic Warfare Common Sensor (IEWCS)," November, 1996.

[48] Defense Standardization Program Case Studies, "Army Battery Standardization: Rechargeable Batteries Power the Future Force," United States Department of Defense, Case Study 7 (DSP-CS-7), from http://www.dsp.dla.mil/, no date.

[49] Defense Standardization Program Case Studies, "Aircraft Batteries and Components: Design Improvements and Standardization Yield Savings and Reliability," United States Department of Defense, Case Study 2 (DSP-CS-2), from http://www.dsp.dla.mil/, no date.

[50] Defense Standardization Program Case Studies, "Mechanically Attached Pipe Fittings: More Cost-Effective Pipe Fabrication Through Standardization," United States Department of Defense, Case Study 5 (DSP-CS-5), from http://www.dsp.dla.mil/, no date.

[51] P. Singh and P. Sandborn, "Obsolescence Driven Design Refresh Planning for Sustainment-Dominated Systems," *The Engineering Economist,* vol. 51, no. 2, pp. 115-139, April-June 2006.

[52] J. E. Devereaux, "Obsolescence: A Systems Engineering and Management Approach for Complex Systems," February 2010.

[53] R. I. Nelson and P. Sandborn, "A New Cass of Multi-Event Models for Determining the Optimum Refresh Frequency of Systems," in *Proceedings DMSMS Conference*, Orlando, FL, November 2012.

[54] K. Emery, "Surface Navy Combat Systems Engineering Strategy," Program Executive Office, Integrated Warfare Systems, 2010.

[55] Defense Standardization Program Case Studies, "Consolidated Acquisition of Standards-Related Information," United States Department of Defense, Case Study 17 (DSP-CS-17), from http://www.dsp.dla.mil/, no date.

[56] M. Wright, D. Humphrey and P. McCluskey, "Uprating Electronic Components

For Use Outside Their Temperature Specification Limits," *IEEE Transactions on Components, Packaging, and Manufacuturing Technology, Part A,* vol. 20, no. 2, pp. 252-256, June 1997.

[57] F. Jensen and N. E. Petersen, Burn-in: An Engineering Approach to the Design and Analysis of Burn-In Procedures, NewYork: Wiley, 1982.

[58] AFRL/RYM Metrics Working Group, *MOSA Metrics Calculator,* Unpublished manuscript, 2012.

[59] P. Henderson, "The Case for Open Systems Architecture," 14 December 2009. [Online]. Available: http://pmh-systems.co.uk/Papers/MOSACaseFor/.

[60] P. Henderson, "A Fuzzy Measure of Openness," 8 February 2012. [Online]. Available: http://pmh-systems.co.uk/Papers/MOSAopennessMetric/.

[61] C. Abts, Extending the COCOMO II Software Cost Model to Estimate Effort and Schedule for Software Systems using Commercial-Off-the-Shelf (COTS) Sofware Components: The COCOTS Model, Ann Arbor, Michigan: UMI Microform, May, 2004.

[62] T. Ellis, "COTS Integration in Software Solutions: A Cost Model," in *Systems Engineering in the Global Marketplace, NCOSE International Symposium*, St. Louis, MO, July 1995.

[63] C. Abts and B. Boehm, "COTS Software Integration Cost Modeling Study," Univesrity of Southern California Center for Software Engineering, 1997.

[64] C. Abts and B. Boehm, "COTS/NDI Software Integration Cost Estimation & USC-CSE COTS Integration Cost Calculator V2.0 User Guide," University of Southern California Center for Software Engineering, September, 1997.

[65] R. Stutzke, "Quantifying the Costs of COTS Volatility," in *Twelfth International Forum on COCOMO and Software Cost Modeling*, Los Angeles, October 1997.

[66] S. B. A. B. Lamine, L. L. Jilani and H. H. B. Ghezala, "SoCoEMo-COTS: A

Software Economic Model for Commerical Off-The-Shelf (COTS) Based Software Development," in *Proceedings of the International Conference on software Engineering Research and Practice, & Conference on Programming Languages and Compilers*, SERP 2006, 2006.

[67] P. Sandborn, "Chapter 14: Burn-In Cost Modeling," in *Cost Analysis of Electronic Systems*, World Scientific, 2013.

[68] J. Abel, E. Berndt and A. White, "Price Indexes for Microsoft's Personal Computer Software Products," National Bureau of Economic Research, Cambridge, Massachusetts, September 2003.

[69] AFRL/RYM MBE Project Team, *Interconnect Trade Study,* Unpublished manuscript, 2011.

[70] Acquisition Community Connection (ACC) Defense Acquisition University (DAU), "Open Architecture Assessment Model (OAAM)," 3 2005. [Online]. Available: https://acc.dau.mil/CommunityBrowser.aspx?id=31395.

[71] P. Sandborn and G. Plunkett, "The Other Half of the DMSMS Problem - Software Obsolescence," *DMSMS Knowledge Sharing Portal Newsletter,* vol. 4, no. 4, pp. 3,11, June 2006.

[72] A. Minkiewicz, "The Real Costs of COTS," Price Systems, L.L.C., March 2001.

[73] D. J. Repici, "Managing IT: The Return of the LOC Monster," Creativyst, 2009. [Online]. Available: http://creativyst.com/Doc/Articles/Mgt/LOCMonster/LOCMonster.htm. [Accessed January 2013].

[74] B. Boehm, C. Abts and B. Clark, COCOMO II Model Definition Manual, University of Southern California, 2000.

[75] Naval Systems Air Command, "Technical Data Packages (TDP): NAVAIR 6.0 Logistics and Industrial Operations," [Online]. Available:

http://www.navair.navy.mil/logistics/TechDataPack/index.html. [Accessed 5 2013].