Certification of COTS Software In NASA Human Rated Flight Systems

Andre Goforth[1]
NASA Ames Research Center, Moffett Field, California, 94035

Adoption of commercial off-the-shelf (COTS) products in safety critical systems has been seen as a promising acquisition strategy to improve mission affordability and, yet, has come with significant barriers and challenges. Attempts to integrate COTS software components into NASA human rated flight systems have been, for the most part, complicated by verification and validation (V&V) requirements necessary for flight certification per NASA's own standards. For software that is from COTS sources, and, in general from 3rd party sources, either commercial, government, modified or open source, the expectation is that it meets the same certification criteria as those used for in-house and that it does so *as if it were built in-house*. The latter is a critical and hidden issue. This paper examines the long-standing barriers and challenges in the use of 3rd party software in safety critical systems and cover recent efforts to use COTS software in NASA's Multi-Purpose Crew Vehicle (MPCV) project. It identifies some core artifacts that without them, the use of COTS and 3rd party software is, for all practical purposes, a nonstarter for affordable and timely insertion into flight critical systems. The paper covers the first use in a flight critical system by NASA of COTS software that has prior FAA certification heritage, which was shown to meet the RTCA-DO-178B standard, and how this certification may, in some cases, be leveraged to allow the use of analysis in lieu of testing. Finally, the paper proposes the establishment of an open source forum for development of safety critical 3rd party software.

I. Introduction

This paper grew out of recent experiences in software verification and validation (V&V) within the Exploration Flight Test 1(EFT1) project. A recent attempt was made in formulation of a guidance document on suitability criteria for use of COTS or 3rd party software in its onboard flight critical system. This guidance document was to cover the requirements that 3rd party software had to meet for it to be inserted into the baseline, and for each instance of 3rd party software, it was to identify the changes to be made to the project's flight software development processes to support this insertion. This effort was deferred. Although the immediate reason was project re-scoping, it became apparent during the effort that even with the original resources there were deep-rooted barriers and challenges in the formulation and application of such a criteria.

The software practices and technology used to build 3rd party software often do not take into account important stipulations found in NASA's software guidance documents. These stipulations call for the production of certain types of documentation artifacts that collectively enable end-to-end traceability from requirements to running code, which is critical to building a robust evidence gathering certification trail.

One observation reached out of this experience is that there is no single solution for easing the insertion of 3rd party software into safety/mission critical systems. However, pressure to integrate 3rd party software technology into flight critical systems is increasing because of rapidly growing innovations in software technology and because of changes in the economics of software.

A. Intended Audience

Because of the ubiquity of software in NASA flight systems this paper is meant to cross cut a variety of roles and disciplines of stakeholders in NASA and its partners. The roles include managers of programs at HQ to managers of projects at the Centers. The disciplines include System Engineers, Flight Software Engineers, Safety and Mission Assurance (SMA) engineers to Software Quality Assurance (SQA) engineers and others. All of these stakeholders

[1]Flight Software Research Engineer, Intelligent Systems Division, MS 269-4

will have a role in future efforts to insert 3rd party software in NASA's safety/mission critical flight systems. Another key category of stakeholders is those technologists who focus on the insertion of innovative software technology into large-scale NASA projects. They are usually outside of a flight project and consist of software business entrepreneurs in the commercial sector and R&D personnel within NASA as well as those external, at universities and research institutes.

B. Purpose And Scope

The purpose and contribution of this paper is to uncover the barriers and challenges, some of which are systemic, that are currently complicating the insertion of 3rd party software in critical flight systems. These significantly complicate the development of suitability criteria for 3rd party software certification for NASA flight systems. Identifying and understanding these is one step towards development of a proactive strategy to enhance NASA's use of 3rd party software in its mission/safety critical systems.

NASA classifies software into seven categories, with Class A being safety critical, and Class B being mission critical[1]; the remaining classes of software cover scenarios that are unlikely to have acute catastrophic consequences resulting from software malfunction. The distinction between the two is that Class A software systems are required to ensure human safety and mission viability whereas Class B is responsible for mission viability, or equivalently, no loss of mission (LOM). Since the EFT1 flight is un-crewed, the software is being developed as Class B. On a subsequent crewed test flight the software will have to be upgraded to Class A. This paper is focused on these two classes of software for two reasons. For one, software developed as Class B is sometimes a direct precursor to Class A; second, NASA's certification requirements for software in human rated systems is the highest that may be attained, and as such, when compared with the other classes, is unique. Furthermore, just because software has be used as Class B in a mission does not mean it is easily upgraded for use in a Class A mission. In the case of the EFT1, there will be significant effort in engineering the software to be robust enough to be suitable as Class A.

II. Background

The EFT1 project is a test flight and one of several milestones in the development of NASA's Multi-Purpose Crewed Vehicle (MPCV). The MPCV is the first major human rated spacecraft to be developed by NASA since the Shuttle[2]. The EFT1 onboard flight software development project is large: Hundreds of flight software engineers are involved; flight software documentation is in the thousands of pages. The Software Development Plan (SDP) alone is close to a thousand pages. The initial budget estimate for onboard flight software was between one and two billion dollars. Some of the preliminary estimates for onboard software lines of code (SLOC) were on the order of a million.

A. NASA Standards

Overarching programmatic and system level V&V guidelines for certification of human rated flight systems is provided by NASA Procedural Requirements (NPR) 8705.2B, Human-Rating Requirements for Space Systems. For software components, NASA has guidance documents, such as NPR 7150.2, Software Engineering Requirements and others that are used to tailor the processes, procedures and reviews used throughout the software life cycle of a given flight software project. These documents require the collection of evidence that is used to determine whether there is sufficient trustworthiness to ratify a system for human use. Depending on the project there may be additional flight readiness reviews and designations that go further and cover specific operational context or intended use scenarios. In NASA such trustworthiness is officially designated in a Certificate of Flight Readiness (COFR)[2].

B. Non-NASA Standards

The MPCV uses a flight software architecture based on the ARINC 653 standard[3]. The use of this specification for its on-board flight software is a first for NASA. This architectural specification is used extensively in the commercial airframe industry and enjoys support of products and tools that have been used in DO-178B certified systems.

The use of DO-178B[4] is also breaking new ground for NASA and, as such, correlating guidelines in FAA's safety critical standard and NASA's 7150.2 and their integrated use for a flight project is a work in progress. The EFT1 project uses the DO-178B standard in two ways. The first is as overarching guidance in conjunction with NASA's 7150.2, when its use is congruous and is advantageous to do so. The second use is as contract requirements in software acquisition. This COTS software has been previously used in DO178B certified systems and NA expecta the contractor will continue to use their existing in-house DO178B compliant processes and procedures. The contractor's prior certification artifacts may be leveraged in the new system.

The use of software with DO-178B legacy offers a unique advantage that is not currently available with NASA's 7150.2B. This advantage is the ability, in certain circumstances, to leverage prior certification evidence towards that needed in the certification of a new system. The circumstances are those where all the differences between the existing systems and the one to be developed can be sufficiently correlated to make such a case. This process is in common practice in the airframe industry and, in fact, has been turned into a product by some vendors, such as Green Hills Software and Wind River Systems. This product is a package that provides the customer a detailed step-by-step set of guidelines that, if used appropriately, will provide a V&V trail sufficient to be audited for certification. Although the vendor cannot guarantee that following the package's steps will automatically result in certification approval by the FAA, it has the pedigree of successful prior use in attaining FAA certification. In many cases, this is more cost efficient than to create the V&V processes and procedures from scratch and then generate the artifacts necessary for certification.

C. Software Verification And Validation

Although NASA's guidelines for software verification and validation apply across the Agency, the implementation and use of guidelines, in practice, is subject to adaptations local to a project. This paper provides a short background that is based on the author's day-to-day observations of software V&V processes and practices within the EFT1 project.

Software verification is a software engineering activity that demonstrates that the product meets specified requirements. The enumeration of a system's (software included) features as a list of requirements, where each are documented with forward and backward traceability to source code and verification success criteria throughout the life cycle of the project, is a fundamental process for both NASA and FAA certification. The requirements traceability matrix (RTM), in principle, defines the complete end-to-end coverage of the processes and documentation. In practice its implementation is project specific and may or may not be part of a software build plan. Regardless of its form, it is the project's software life cycle documentation linked together within the structure of this matrix that ties every bit in the as-built or *piece part* of software to a *shall* and to its verification success criteria (of test cases).

Software validation is a software engineering activity that demonstrates the as-built software product or *piece-par*t satisfies its *intended use* in the target or *intended environment*. It follows verification because it occurs down stream in the testing regime where there is a test platform of sufficient fidelity and sufficient aggregation of other software *piece-parts* to conduct testing—validation—of the software product's *intended use* or role in an operational or *intended environment*. On account of the need to develop special avionics computing components, which are sufficiently radiation tolerant for space flight, the MPCV has multiple verification activities that address the development of each of these components and their software. These verification activities are linked together and in some cases dovetail one another. The results of one verification activity become the basis upon which a subsequent verification activity starts and continues with another set of testing, but on a higher fidelity or more specialized test platform. Each of these test activities is a validation of *intended use* upon *its* test platform or rig.

The number of verification activities and their sequence depends on unique project needs, and, in the case of a flight project such as EFT1, on the principle of Test Like You Fly (TLYF) and Fly Like You Test (FLYT). This principle means that, out of a series of integration test platforms or environments, there is a final one that is identical to the actual flight system both in hardware and software configuration, and in process configuration, which includes provisioning a system for operation and life cycle maintenance.

For simpler systems, software verification and validation is a two-step process where individual software components are tested—verified—and then integrated on the target platform or environment and tested there—validated—for *intended use* in the actual environment. For the EFT1 the series of integration test verification activities, which may be viewed as a sequence of validation steps, have to culminate in one final one. This sequence is a chain, which means that regression testing requiring the use of a prior verification activity's tests and environment has to be supported to some prior designated configuration. What is important to note is that software *piece-parts* move through successive approximations of *intended use* validation in the series of integration test activities, which are traceable to the final end-to-end system validation. Also worth noting is that all the software life cycle processes and procedures built up over the course of the verification activities have to converge to those used to support TLYF-FLYT at the final validation.

III. Barriers And Challenges

For purposes of this paper the distinction between barriers and challenges, which is not hard and fast, is that the former has solutions resulting from the accomplishment of concrete objectives; for example, modifications in an

organization's standard operating procedures can remove barriers. Challenges, on the other hand, require goal setting that result in paradigm shifts. For example, changes corporate culture and mind-set may be carried out through the use of strategic level policies and directives.

A. Complexity Of Software In Human-Rated Space Systems

A major challenge in the systems engineering of human rated systems is complexity. This paper notes two aspects of this complexity that impacts software, in general, and COTS software in particular. First is that the growth of software within NASA space systems is driven by more and more ambitious mission requirements. *NASA Study on Flight Software Complexity*[5] provides extensive data of the growth of software in missions over time and related software metrics. This growth in more ambitious requirements is a major driver to adopt solutions available from 3rd party sources.

The second aspect of complexity is the role or type of software within a system. This leads to the fundamental dichotomy between software and hardware: software is intangible as it has no physical properties and its final manifestations in a flight system are in the form of binary sequences, whereas hardware has physical properties. This raises the next topic.

B. Managing Abstraction Of Software

From the systems engineering perspective, Kossiakoff[6] notes that software's intangible property of *abstraction* is the most fundamental difference between software and hardware that faces the discipline. From a software reliability engineering perspective, Lyu also notes this fundamental difference[7], which makes for major challenges in reliability engineering.

From a systems engineering vantage the roles or types of systems are *software-embedded systems*, *software-intensive systems* and *computing-intensive systems*[8]. *Software-embedded systems* have software that is integral to the reactive behavior of a system's physical components that interact with the physical world in real-time. In many cases, safety of the equipment, the environment or personnel is of paramount importance. *Software-intensive* systems, of which the Internet is a prime example, have their observable or reactive behavior done completely by software. Control and interaction with the physical world is negligible and responsiveness is not tied to real-time, stable closed loop control of a physical device, but to intermittent interaction with a keyboard.

Computing intensive systems consist of large scale farms of computing resources dedicated to massive computing tasks such as those found in weather analysis/prediction, or data mining in enterprise-wide data warehouses. The role of software here is to coordinate the processing resources while executing data processing algorithms; the observable or reactive behavior is negligible until it is rendered for human use.

It may be said that in *software-intensive* and *computing intensive systems* the software is the system. The system's physical plant is mainly the computing platform there to support the software. Whereas, in *software-embedded systems,* the role is reversed, where software is there to animate the physical plant.

Because of the unique operational constraints of radiation intensive space environments, real-time scheduling needs, requirements for close integration with hardware device characteristics, and limitations in processing and bandwidth, the technology base for flight critical software is relatively limited. This is in comparison with what is available on the whole in ground-based systems where most COTS or 3rd party software reside. There the re-use of heterogeneously implemented software components over a broad range of computing platforms operating in benign environments the usual case. Reuse on different platforms is readily done through the use of wrappers and other forms of functional encapsulation or by means of interface proxies

Unfortunately, the use of these and other abstractions come with associated long functional call stack traces that in current hard real-time platforms result in the lack of timing predictability. A systems engineering solution is to limit the use of software with abstraction features that make predicting timing performance difficult. This is a barrier when the project does not have the expertise to apply such software in a way that permits adequate real-time performance. This may be viewed as a challenge when more fundamental technology readiness level (TRL)[21] research is needed to overcome such an obstacle.

C. COTS Software Obsolescence In Mission Critical Systems

COTS software obsolescence in mission or flight critical systems is a challenge. A recent paper[9] provides data based on COTS acquisitions by DoD programs. It shows that program/project obsolescence is encountered because of the differences between the lengths of life cycles found in technology versus those found in DoD programs. For COTS software acquisitions of today, the difference between program/project life cycle periods versus COTS software life cycle periods is growing wider rapidly. The rapid pace of innovation in software in the commercial

sector and its pressures for short time-to-market life cycles tends to accelerate software obsolescence found in slower paced large flight critical projects. What adds even more to the trend of obsolescence is that obsolete software often requires obsolete hardware, and, as a result, NASA's flight systems, as a whole, become more and more removed, in general, from COTS technology of the day and are, hence, made obsolete.

E. COTS Software Acquisition Barriers Uncovered In The EFT1 Flight Software Project

The EFT1 flight software team considered three COTS software products that had no certification legacy in either NASA or FAA flight critical systems. These software products included an executive scripting control language application, software to operate a global positioning system (GPS) receiver and a testability application that aided in system fault detection, isolation and recovery (FDIR).

As part of NASA's oversight role, it had the responsibility to formulate guidance in the form of suitability criteria by which COTS or 3rd party software would be assessed initially for inclusion and what subsequent verification and validation artifacts were necessary for certifying this software. This guidance document was to be a tailoring of guidelines from NPR 7150.2, NASA STD-8739.8[20], and EFT1 contract and project management documents. The draft of the plan had a number of assessment or suitability criteria artifacts such as, product design requirements, software design documents, source code (including compilers used), test records, service history including faults, bugs, quality assurance records, etc.

Nearly all of these artifacts were found to be proprietary and accessibility was dependent on negotiations. As the effort to complete the plan progressed the overall feasibility and practicality of adopting these products was put in doubt because of barriers. One barrier was there was no way to predict the negotiation completion date. As a result this acquisition was seen as a liability because it came with a major schedule risk. Even if, for sake of argument, negotiations took no time, the time and effort to adequately perform the assessments of the artifacts was difficult to estimate. This, too, was seen as a significant liability because of limited project resources.

For each product the following had to be ascertained:

1) How well the products' features or capabilities matched EFT1 requirements,

2) How well did the source code trace to requirements and

3) What verification success criteria—test cases, if any, matched up with source and requirements.

In a nutshell, for COTS software to be inserted into the project baseline without major reservations, without major waivers that would cast a cloud on the possibility of flight critical certification, the software product would have to undergo significant transformations to make it compatible with EFT1's requirements traceability matrix (RTM). Matching up products' features or capabilities with the project's software requirements baseline and resolving mismatches are major resource sinks and can require more effort than it's worth. EFT1 software engineers were faced with a task equivalent to reverse engineering the requirements—creating a list of *shalls*— from the existing features as could be found in a working version of the software (if it were available), its user documentation and any available product design documents.

Having created such a list it was then necessary to determine whether there is complete coverage of targeted EFT1 requirements. The product's source code would have to be modified to add in missing capabilities. On the other hand, there is the issue of what to do with those features already in the product that had no counterpart in NASA's requirement set. As mentioned earlier, for mission/safety critical software the verification process does not allow any bits to be in the final product that are not traceable to a *shall*; in terms of source code those lines of code without traceable requirements would have to be removed.

Simply put, the match between NASA's requirements and the features in the software had enough mismatches to make a reconciliation effort tantamount to developing the software from scratch.

A possible alternative was to commit to requesting an extensive series of waivers to deal such incompatibilities and others, such as programming language and programming coding standards usage. However, this alternative would have required the overhead of carrying special verification processes and procedures tailored for each one of these candidates through the multitude of verification activities mentioned in section II C. Software Verification and Validation. This could be a significant source of life cycle process and procedure overhead.

1. NASA's Guidance For COTS Software Acquisitions

NASA's acquisition of COTS software, which is similar to FAA's, is covered in section 2.3 in NPR 7150.2. Its guidance on use of off-the-shelf software in critical systems anticipates the likelihood that maintenance and support will not be available from its vendor and makes additional requirements for COTS or 3rd party software use. These are summarized here:

1) A copy of the source code is provided to NASA.

2) Risk mitigation plan to cover the possibility of loss of the supplier or maintenance is made.

3) Agreement with the supplier for access to defect reporting by client community is made.

4) Life cycle maintenance and support plan specific for this product is developed.

5) Documentation of changes to the project's software life cycle caused by use of COTS is made.

For a flight project such as EFT1 to satisfy these requirements the effort in licensing, contracting and project logistics was found to greatly depend on the vendor's business model. There are two business models. One is a DO-178 compliance business model; the other is the propriety business model. The latter is discussed below in this section; the former is covered in a subsequent one.

For COTS software with little or no legacy of use in mission/safety critical systems the barriers to incorporate within a class A/B flight project are nontrivial. Licensing, contracting and project logistics overhead may serve as significant barriers depending on the vendor's business model and the type and role of software product to be used.

2. Type of Software

For EFT1 much of the COTS or 3rd party software, which was from vendors with proprietary business models or no model at all, was not clearly of the *software-embedded system* type, where the product has interface features that aid in close integration with other parts of a real-time system. At best, some of the software had been used in reactive systems before but as stand-alone applications that interacted through remote-like interfaces. This disparity between fundamental types of software where the intent to use software of one type in another type is a barrier because of the extensive tailoring necessary to add features that support integration and to support close coupled real-time performance.

3. Propriety Business Model

When the vendor's business model is proprietary and does not accommodate licensing of sources any negotiation on terms and price to provide source code means that the vendor has to make a major exception in their business practices. This results in uncharted negotiations for both parties and may be long and indeterminate in length as noted earlier. If negotiations result in terms that are mutually acceptable then NASA, technically, owns a copy of the source code and is free to incorporate it into its baseline. However, even with the terms including life cycle support and maintenance, once the software is made part of the project's software base it starts to lose its status as one of the vendor's ongoing products. Furtheremore, the broad experience and usage base that is continually updated within the dynamics of a user-vendor community is eroded. The reason for this is that the sheer cost and complexity to create a software development infrastructure, which is compatible with an external verification and validation cycle, is too great for either party. This would require contracting for services that these COTS software vendors are unlikely to support in the manner required by NASA.

4. Licensing and Contracting Barriers

To compound the overhead further for a vendor, current contracting practices of NASA and its support contractors have SBU and ITAR restrictions that would have to be navigated to include a 3rd party software provider. In these circumstances, if NASA decides to take sole responsibility and continue with the acquisition with little or no support vendor support, any changes to the product for the sake of bug fixes or feature updates are NASA's. As a consequence, the software becomes an orphan with its own unique set of V&V processes and procedures that are not the original vendor's and that require custom tailoring of the project's baseline software development processes and procedures. This is a major maintenance barrier in the adoption for such COTS or 3rd party software.

IV. Motivation To Use COTS or 3rd Party Software

A. Affordability

The first motivation is to lower software life cycle acquisition costs. In the past several decades there are been a strong trend in all commercial sectors to use COTS software either as components of an in-house built system or as a commercial turnkey system. This is particularly true of *software-intensive* and *computing-intensive systems*. In terms of software for NASA and other government agencies *program affordability* deals with the cost to acquire, cost to integrate and maintain a software acquisition. This is a make-versus-buy decision. The rate of growth and innovation of COTS *software-intensive* and *computing-intensive systems* products is making this decision process less of a make-versus-buy choice and more of a selection of the most appropriate from a pool of COTS candidates. Even for those instances where the choice to make is appropriate, the trend is that the in-house built software functions as *glue* that integrates a collection of COTS software modules.

B. Competitiveness

The second motivation is competitiveness. Affordability or cost is a key part of competitiveness but without consideration of other factors it can be irrelevant. A flight software product that is low cost, i.e. affordable, can have a short shelf life and be prone to obsolescence, which is to say not competitive. Competiveness is about sustaining an effective technological edge, which means building software systems whose life cycle costs are commensurate with current economic trends.

In the past the cliché of *faster, cheaper, better* was used to characterize the need for NASA missions to fit in with smaller budgets. This is an excellent goal to strive for, but blindly used, it may be unrealistic due to intrinsic physical costs. However, software does not have intrinsic physical properties of mass and volume. So, in principle, software is not fettered to the life cycle of material systems, where a given amount of stock must be acquired, then processed in construction of the system and then maintained against wear and tear. Because of this intangibility, it may be argued that there is virtually no limit to the pursuit of *faster, cheaper, better* systems through software. However, not to get carried away, there are limits to what can be done by software. The key point is that the relative cost to acquire software in comparison to the cost to acquire non-software components of a system is continuing to diminish with successive generations of innovation found in 3rd party software technology. This is another motive for the use of COTS or 3rd party software.

C. Ultra Reliable Software To Support Permanent Human Presence In Space

The MPCV is one of NASA's next steps to support permanent human presence in space. This project is a harbinger of those to follow that will require significant application of innovation in the use of existing technologies as well as in the creation of new ones. Information technology, which covers all types of software, will be one of the critical technologies. The reason for this is the growing role of software in NASA systems. An example of this trend is found in the EFT1 project where 60% of the avionics subsystem requirements are delegated to software. Two potential sources for software with ultra reliable legacy are vendors that have a business model geared for their products to be inserted into flight critical systems and vendors with large scale, open source business models that have products that make up much of the backbone of the Internet's information infrastructure and of many corporate enterprises.

1. DO-178 Compliance Business Model

Vendors, that provide software with a DO-178B pedigree, have a business model geared for its products to be inserted in flight critical systems, i.e. *software-embedded systems*. Such a business model includes services that address the flight critical requirements as found in DO-178B and NPR 7150.2 in a way that preserves intellectual property and lowers the cost of insertion into a client's flight system's baseline. In the case of the EFT1 these requirements were addressed in contract negotiations at the start of the project and, for those exceptions uncovered later on, by subsequent negotiations in the tailoring of and waivers to the vendor's software life cycle processes and procedures, which are covered by the project's Software Development Plan (SDP). For the EFT1, the only significant deviation in meeting the requirements was in access to source code. The contracted alternative was to provide NASA opportunities to inspect the source code and certification artifacts during briefings under the control of the contractor. The contractor has been responsible for providing previously acquired evidence for flight critical certification and producing updated V&V artifacts for certification that have the necessary and sufficient equivalence to the kind of V&V evidence that NASA creates for in-house developed products.

However, there will be significant differences in the operational theatres that NASA missions will operate in versus the current ones that are addressed by these vendors whose domains cover aviation and automotive. In terms of systems reliability, spacecraft, automobiles and aircraft share in the need of being ultra-reliable where the chances for mishaps with serious injury or loss of life caused by malfunction is on the order of 1 in 10^9 hours. This in turn, depending on the built-in system level redundancy features, may require software reliability to be no less than this, where the system's software may directly contribute to the root cause of failure leading to a mishap. For automobiles and aircraft the relatively short length of missions and proximity to maintenance depots make achievement of such reliability easier than that with spacecraft where proximity and mission length are considerably different.

2. Open Source COTS or 3rd Party Business Model

For a spacecraft the length and remoteness of its operational theatre requires autonomy that is virtually without parallel in current domains, such as aviation. The longest mission (flight) for a passenger airplane is currently around twelve hours, which is measured between take-off and landing nominally, at a designated airport. Human rated space flight missions beyond low earth orbit (LEO) range in days to months, even years. This will have a major impact on the engineering of system level mission/safety attainment and, specifically, on software mission/safety certification. Put another way, in terms of computing resources and their robustness a crewed mission to the moon or Mars or beyond will need to bring with it the software capabilities found on earth, like those found in

ultra reliable software-intensive and *computing intensive systems.* Furthermore, depot maintenance in the form of tools and processes, which is currently provided in close proximity to operational systems, will have to be carried along and made available in flight and in situ at remote outposts. Humans in space will need to have the same capabilities to fix and enhance software as found on the ground. But, as it has been previously covered, there are barriers to adoption of COTS or 3rd party software that is not developed specifically to be in compliance with portions of software safety critical standards found in DO-178B or NPR7150.2. To address this will require a paradigm shift in how software is developed both on the part of open source vendors and NASA. The rest of this paper focuses on aspects of this paradigm shift.

V. Future Directions

Given the barriers and challenges and the motivation to use COTS or 3rd party software in safety critical space flight systems the question becomes how to reap the benefits of these safely and economically. A vision is offered to aid those who are involved in strategic planning of flight software technology use within NASA and its partners. In addition, based on experiences in the EFT1 project, some plausible ways are provided to lower software V&V and certification costs by leveraging prior V&V artifacts found in COTS software certified to DO-178B and by leveraging architectural properties of the ARINC 653.

A. Cross Cutting Vision

Software verification and validation consists of software tools and processes that aid in the production of target or end-item software; they are not a *piece-part* in a loadable image of the target flight software. As such they may be viewed as overhead or a means to an end to those focused on strictly the capabilities of the end-item software products to satisfy operational requirements. An ideal circumstance for this group is to have a software life cycle environment that automatically takes care of V&V. Whether this is attainable in the future or not[10], the role of software V&V is not to be viewed merely as a necessary evil, but to be viewed in terms of a cross cutting vision. For purposes of this section this vision is made in terms of a medical analogue, that of an immune system.

In the big picture, the success of inserting 3rd party software from one V&V life cycle instance into another depends on the compatibility between the two much like that found in transplanting tissues or organs from a foreign immune system into the recipient's. R&D papers and technologies that have direct or significant bearing on COTS software certification may be grouped into several categories. The first one is systems engineering (SE), which covers a number of disciplines and aspects of engineering including software[13]. The second is software reliability engineering (SRE), a major subset of SE, and which deals with software quality in terms of a hardware-software vantage of fault management[13 14]. The third is suitability criteria for COTS software certification, which does not appear to have much attention thus far. The fourth is safety cases research, which falls within the discipline of formal methods[15]. It straddles the boundaries of safety assurance and hazard analysis, which are sub-disciplines of systems engineering and software safety engineering[16].

The immune system guards against and fights off bugs and infections; likewise, the processes and procedures in the V&V life cycle are there to fight off the *incorporation* of software bugs or faults. This parallels two systems engineering fault management lifecycle techniques mentioned by Lyu, that of *fault prevention* and *fault removal*[11]. The V&V life cycle in which COTS software is produced in is in the donor's immune system. On the other hand, the recipient's life cycle has its own end-item software products and does this in a way as to make them integral to its own system-wide fault management capabilities. This parallels another systems engineering fault management technique that of *fault tolerance*[11] or from a software reliability engineering management technique of *fault containment*[12]. Compatibility between a donor and a recipient is a multi-level complex subject; however, advanced software architectures that provide explicit fault management frameworks and component-based software engineering have the potential to address software incompatibilities[15].

Although current R&D in these categories is pushing the envelope for the benefit of the V&V or certification community, for 3rd party software insertion into mission critical systems there is still a need for R&D that tries to find ways to make immune systems (V&V life cycles) more immune friendly (compatible) while maintaining their own immune integrity. Details of what such research should be is outside the scope of this paper; the objective at this point in the paper is to call attention to such a need, which does not fit neatly into pre-existing subject categories. In general terms and to provide a vision, such research may pursue the creation and application of advanced concepts such as architectural structures, algorithms, formal methodology, etc. toward this end. These would be put into practice through the development of architectural artifacts linked together via a tool chain within a unified information infrastructure that guarantees the integrity of the RTM. From these advances, immune systems,

i.e. V&V life cycles would be smarter and more quickly capable of synthesizing foreign end-item software with their own and, also, synthesize adaptations (updates) to their own.

B. Extending Structural Integrity: The ARINC 653 Specification

In terms of architectural artifacts, the ARINC 653 specification is a step in easing the V&V efforts necessary for flight critical certification. The specification is based on an integrated modular architecture (IMA) framework and provides for definitions of services and software interfaces. These collectively support the *golden rule* of space-time partitioning where it is possible to certify that a change made in one partition—a collection of real-time tasks— will not have unintended side effects in another. This property eases the certification effort of the whole system because trustworthiness of a partition is independent of others.

Such a property of structural integrity is made possible by architectural features of the specification. This is an example where, from a systems engineering view, software *abstraction* is not viewed as a liability, as mentioned earlier in section III B, but serves as a systems and software engineering asset. With hardware the dictum that *form follows function* is grounded in physical reality. With software one approximation to *form follows function* is found in the concept of *reification* as applied to software[17]. Application of this research to flight software architectural specifications, such as ARINC 653, may result in the adoption of more software architectural features within the systems engineering lexicon, which currently consists mainly of *partition*, *process,* and terms related to the ARINC 653 APEX. Such applied research needs to be grounded in a systems engineering environment, which is one of practice, where there are practical candidate problems with a community of stakeholders that have an urgency to create specifications that embrace even more structural integrity.

The ARINC 653 architecture would be even more certification friendly if its specification was extended to cover additional properties of structural integrity found at the partition level to within the partition where each process's certification is independent of others. Note that the term certification has been used loosely, and to be more precise, the paper is referring to certification's role as a capstone of verification and validation, where officially designated personnel outside the V&V organization are commissioned to audit *run for record* verification artifacts.

Another area to address is what may be added to the ARINC 653 architecture to ease the certification of a collection of cooperating partitions. The focus is on practical specification extensions that the current ARINC 653 specification leaves to the *system integrator*—a system architect cum system administrator cum system engineer— a complex role that is distinct from that of a partition application developer. Currently, every flight project that wants to use the ARINC 653 architecture has to create its own custom infrastructure or middleware to handle partition interactions.

This infrastructure, which the application partitions have to abide by, includes supporting interfaces to overarching features of system initialization, restart, fault management and others that are all under the purview of the system integrator. In support of the application developer, the vision is to create an *empty box* architecture that, as an extension to the ARINC 653 specification, provides its own sphere of structural integrity consistent with the existing one. This would allow one to drop COTS or 3rd party software code into an implementation instance, for example, a partition or within a partition's process of the architecture and have the main *unknowns* be for the most part limited to the verification of the code locally, verification of its project coding standards and verification of its use of *empty box's* interfaces. Note that once the *empty box* implementation infrastructure has been used in the certification of a flight critical system, this certification could be leveraged in other NASA flight critical systems that decide to use the ARINC integrated modular architecture approach.

C. Software Reliability Analysis

For a NASA project such as the EFT1 requirements verification artifacts and methods fall into one of the following types: analysis, demonstration, inspection or test. Testing of software is by far the greatest consumer of verification and validation resources. Any means to lower the total testing effort profile across the life cycle translates into a major improvement in affordability. Historically, testing has not been seen as fungible with any other verification method. However, with software that has been DO-178B certified on a given platform, it is plausible to consider the use of software reliability analysis in lieu of testing.

The MPCV project had base-lined a COTS software real-time operating system (RTOS) that was a DO-178B compliant. This software is marketed as DO-178B certified, which means it has prior usage in safety critical systems certified by the FAA. The MPCV project decided to leverage this certification to the extent is was consistent with its own standard, NPR 7150.2.

The features of this software are all in terms of application program interface (API) calls. These API calls as specified in the vendor's documentation were identically mapped to requirements contained in a software

requirements specification (SRS) document. Although there is no one way to formulate verification and validation documentation (it is something tailored by projects) there has to be a designated list of capabilities that the software will perform and for each capability a means of verification established, which EFT1 software requirements specification (SRS) type documents do. What was under consideration was the means of verification to be used for the vendor's software, which implemented the API calls; solutions ranged between two extremes: one was to test it no differently from in-house developed software; the other was to do no testing at all. Here are the rationales for these approaches.

Treating this COTS software no differently from in-house developed software is arguably the most straightforward and safe way to go. No waiver or tailoring of the project's software verification and validation life cycle is necessary.

On the other hand, in this context with the no-testing-at-all approach the testing of *as built*, *piece part* code that has identical *intended use* as found on a large number of other operational systems may be seen as redundant and therefore, analysis was a plausible option. With the no-testing-at-all approach, the observation that in-house testing, in these circumstances, would be insignificant when compared to the fact that this software has been in operation—*tested in practice*— 24 by 7, 365 days a year, for the past ten or so years, which follows from aviation industry's experience that such a COTS software product was ultra-reliable, i.e. there is a 1 in 10^9 hours chance of failure. Any testing of this software by the project would be a drop in the bucket compared to even only a day's worth of operation.

The rationale for use of analysis is further strengthened when there is similarity in the computing platforms used in prior certified systems with those to be used. Two salient platform characteristics are the CPU clock rate and the backplane clock rate. If the to-be-used processor's CPU and backplane clock rates are multiples of prior ones, which have certified with the same version of the RTOS, and all other characteristics are the same, then it is plausible to follow through with an *argument of similarity*. To do this requires a re-audit of the certification package associated with the RTOS software along with the prior *as built* flight units. Such a re-audit is within the nominal business services provided by the contractor responsible for this software and is available at a fixed price[2]. The purchase of this service would result in assurance that prior certification credit could be used in the current system and therefore, testing is not necessary to maintain certification credit.

The choice of whether to use analysis in lieu of testing is now a matter of the cost of a re-audit versus the cost of testing. Specific cost figures from the EFT1 project were not available on account of the proprietary nature of the data. However, it is fair to say that the cost of testing was several fold more than by analysis. It would require a significant increase in hiring over several months just for the testing; whereas the analysis could be done with little, if any, increase in staff over several weeks. The reader may wish to use their own estimates in FTE hours in the following: There were around 200 RTOS API calls to test, which means that around 200 verification success criteria have to be written; for each of these criteria there is at least one test case to develop; and each test case needs to be run a sufficient number of times with sufficient variation in test parameters to cover all corner cases for complete verification.

An alternative under consideration was to use a hybrid approach where testing is limited to only those API calls actually used in the test flight. This alternative uses analysis that relies on the legacy of prior certification evidence.

The previous discussion is a case in point in the potential use of software reliability analysis in lieu of testing as a means of verification. The challenge for NASA with such an analysis is that, even though it may appear compelling to some, it is not part of its conventional—tried and true—systems engineering practice and judgment used in Agency's flight critical projects. It is only recently that NASA has started to use 3rd party software in flight critical systems. However, this pedigree is not perceived as *legacy* software, where by definition, it is code that was developed and flown by NASA. With experience in the use of 3rd party software that has a safety critical pedigree, NASA will adapt to its advantages. What would help, in addition to such experience, is more applied research that focuses on extending structural integrity of architectures so that a priori decisions to use analysis or other verification methods in lieu of testing are formally grounded and become part of flight critical systems engineering discipline's practice.

VI. Steps To Take

Innovation in flight software system verification and validation per se is not part of the primary mission objectives of a flight project. Verification and validation within the flight software portion of a project such as the EFT1 are a means to an end. The insertion of new processes and methodology are seen as a project risk to be

[2] Vendor marketing presentations showed this cost to be around 500K.

avoided because of unknowns in resources needed and in delays to schedule. Any innovation to broaden and ease the insertion of 3rd party software, which has no prior flight critical certification legacy, into NASA's flight systems needs to be pioneered and sponsored outside the project level.

A. Software V&V Process Technology Demonstration

An alternative, which would not likely be part of the MPCV, is to create a flight critical technology demonstration project that starts with the premise that it must lower software development and maintenance costs through the extensive use of 3rd party software. In this circumstance the risk is confronted head on in the same way that a technical solution with high risk comes from the necessity— to develop new hardware to satisfy a currently unobtainable processing requirement—and is therefore unavoidable.

This would drive from the very start the development of guidelines for use of 3rd party software into the baseline and at the same time, may find ways for software V&V processes to be more accommodating without compromise in terms of guardians of trustworthiness. There are two concerns with such an approach.

The first is simply that making software process improvement the mission of the project is a difficult sell because of its intangible and abstract nature. Other flight critical projects may laud the demonstration of success in the form of dramatically lowered certification cost, but be skeptical as to whether the mission criticality of the demonstration project was comparable to their own, particularly if theirs is a safety critical one. Another project level issue is finding and allocating resources necessary to incorporate the new technology; the results of a technology demonstration project do not usually include the availability of such resources.

The second problem is that creating one such exemplar project is simply not sufficient to create the critical mass of advocacy for wholesale adoption throughout the Agency. Systemic shifts in the use of technology in an organization as large as NASA rarely occur by fiat from the top but, instead, trickle up from the bottom through incremental adoptions of new techniques by worker bees and through prime contractors and their subcontractors.

B. An Open Source Forum For 3rd Party Software Certification

NASA already sponsors and participates in an open source software forum[18]. The focus of this forum is to improve software quality used by NASA through community review.

What this paper proposes is the creation of an open source forum that includes this objective and, in addition, fosters and promotes the use of 3rd party software in mission/safety critical systems with a focus on the creation and maintenance of verification artifacts that aid in flight certification. The overarching goal is to create a community consisting of NASA, industry, and academic technologists that want to share software and software verification artifacts that have flight critical legacy and to share software that they wish would attain flight critical legacy.

In addition, this forum would be a resource available to NASA flight software engineers and contractor counterparts to participate in, at the discretion of their project management, on their own terms in schedule and budget. It could serve as a conduit to infuse technological innovations into mission critical projects such as EFT1. The forum could provide the means for preliminary vetting of the suitability of 3rd party by a flight project. This would be a starting point that would make it much easier for a project such as the EFT1 to create its own suitability criteria guideline for which it has contractual responsibility to provide to the prime contractor.

1. No Fee Software Model

A great deal of COTS software is open source; some examples are LINUX, Apache, and GNU software[19]. Open source means that an individual or corporation is allowed to use the software without charge as long as they abide by the user license agreement (ULA), which, for the most part, limits what proprietary designation a user may in turn assign to products they create that either incorporates or uses the original software. This promotes communal use and sharing and results in having a large community of developers, testers and users working together, either as individuals or company employees. All participants and users benefit from the built-in maintenance and incremental development of features, where these features are tested in a large variety of *intended use* settings.

Participation is not limited to individuals. Numerous open source software products have corporate and academic sponsorship. In some cases, private companies have opened up their proprietary products in open source venues and thereby leveraged the efforts of a large pool of *free* workers while contributing in kind their own resources. This is true for a large portion of *software-intensive systems* and *computing-intensive systems*, but to a rather limited degree, to *software-embedded systems*.

Unfortunately, it has already been seen virtually all products in this category face a major uphill battle to become a part of the software baseline of a NASA mission or safety critical system on account of the gap between their software V&V life cycle and NASA's. The forum's primary motivation and role is to bridge this gap for the benefit of all parties.

2. For Fee Software Model

On the other hand, there is COTS software that is not open source. An example are vendors providing software products for use in FAA certified systems, where they have had to make large investments in the creation of V&V artifacts essential to DO-178B certification. For the most part these products are developed and maintained by smaller contingents of workers because the product's market size is relatively small. Furthermore, in these cases, there is unique intellectual property that needs to be kept proprietary for the business to retain its market place advantage.

For the vendors of these products an open source COTS software forum may not be a good fit with their business model. But the need to partner with other vendors, such as with non-DO-178B compliant products, to share interfaces and portions of code could be a reason to participate in a special sub-forum. This sub-forum would focus on verification practices that work around proprietary restrictions that, for example, result in the unavailability of source code and other V&V artifacts. For such parties they have the FAA as a 3rd party resource to go to and see if their application of the DO-178B standard in these circumstances is credible to the FAA's Designated Engineering Representative (DER). The incorporation of new V&V processes based on DO-178B as guidance may be tried out and adopted by NASA within the context of this forum.

C. Suitability Criteria And Workbench For COTS Or 3rd Party Software In Flight Critical Systems

For NASA flight critical software projects NPR 7150.2 has 256 guidance requirements to be followed. While all of these apply, in principle, to COTS software no differently than in-house developed software, six are specific to COTS software. As observed in practice in a project such as EFT1 the use of COTS software in onboard flight critical systems may be subject to extensive alterations that make it as if it were built in-house.

In broader terms of the cross cutting vision of the immune analogue covered in section V A, the question may be asked: what is the minimum that a donor, the software product advocate, needs to do to increase the likelihood of product acceptance in the recipient, such as a flight project? The forum will need to develop suitability criteria for COTS software in class A/B systems, which was originally to be done by the EFT1 project, but the forum's guidance will have to be broader to have relevance to a larger community where the constituents are not limited to a specific flight project and may be partners from outside the Agency.

There would be two types of forum donors. One is the software product advocate with an existing product who wants to consider the kind and degree of modifications and retrofitting they have to make to gain some credible measure of compliance. The other is the software product advocate with a blank sheet who wants to build from the start software artifacts that are compatible with guidance criteria. Note that final software certification is project specific and therefore, may only be done under the authority of the project. As a consequence, whatever measure of compliance with forum criteria a software product demonstrates, it is not a replacement of what is created from a project's run-for-record verification activities.

The following software V&V artifacts are suggested as an entry point into the formulation of suitability criteria:

1) A list of requirements
2) The source code that implements these requirements
3) Verification success criteria and associated test cases and specification of the intended environment
4) Two way traceability between these.

There is much more work to be carried out in regards to the specifics of these artifacts, and undoubtedly, others will be uncovered upon review of NPR 7150.2 and will make their way into the criteria. This guidance criteria is one product of two major components of the forum. The other is a complement to the guidance criteria and is the vehicle upon which compliance to the criteria is measured.

This product will consist of an operational common framework, a workbench per se, that will support all of the end-to-end processes for traceability, verification, and configuration management of requirements, source code and other documentation artifacts. The vision is that software developed under the auspices of the forum will come as a package that contains the tools, check-lists and other artifacts, as well as any prior usage evidence, all of which is to be supported by the workbench under an open source license.

For requirements management there is the matter of how are requirements represented. For example, NASA uses English to state requirements, as *shall* statements. When you attempt to bring in a COTS software product that has its capabilities enumerated in a list not necessarily stated as *shalls* (or not in a list at all) there remains the matter to resolve the semantics used in the donor representation with the semantics used in the target or recipient system. This is an opportunity for researchers in requirements based specification, for example, to apply their technology to ease the insertion of COTS software in terms of this artifact.

For source code there are multiple choices for programming languages and for coding standards. The challenge here is to come up with tailor-able specifications and criteria that allow alignment with a target recipient environment.

Verification success criteria and their test cases are usually created for the different activities found in phases and sub-phases of development, and in test and integration. For example, a development phase may have informal unit testing performed that is distinct from formal unit test. Likewise test and integration activities may have multiple instances of verification, some informal and some informal. The challenge here is to provide a substantive set of test cases that aid in complete coverage of requirements testing.

For the other artifacts, they have their own unique matters that, in turn, could benefit from the application of advanced technology. The forum is seen as a place where advanced technologists and flight project personnel are brought together with a common goal to ease the adoption costs of COTS software in flight critical systems.

VII. Summary

Complexity of software in human-rated space systems is increasing because of the demands for autonomy in long term, remote missions. This trend will accelerate because more and more mission requirements are being delegated to software. A recent example is that 60% of EFT1 avionics subsystem requirements are served by software and it is only an un-crewed test flight; the percentage is likely to increase for a crewed flight.

A major challenge for the discipline of systems engineering is managing abstraction of software. Software is often aggregated into a computer system configuration item (CSCI). The CSCI is more of a contracting and project management artifact and less of a representation of a software architecture or framework artifact. In fact, a systems' software architecture may be somewhat dictated by contract specifications and this may impact the complexity of software and may force architectural solutions, which are less than optimal, that are necessary to address system requirements.

COTS software obsolescence in mission critical systems is caused by the difference between the length of time of software technology innovation and the length of a project. Once software is developed for a flight critical mission it is difficult and impractical to upgrade to the current software technology base. This is a challenge.

COTS software acquisition barriers depend on the vendor's business model and type of systems in which the software has been used. If the vendor's business model is proprietary and its product is not of a type, such as *embedded systems*, then the barriers are rather high because of the lack of availability of verification artifacts. On the other hand, if the vendor's business model supports the availability of verification artifacts and its product has been used in a similar system as the target then the barriers are low.

A motivation to use COTS or 3rd party software is affordability. Program affordability is a perennial issue for all of NASA's missions. Software affordability is becoming a bigger part of program affordability because of its growing role in the implementation of mission and system requirements. Stakeholders inside and outside of NASA observe software performing critical functions in large scale earth based systems that have the same order of complexity that NASA flight systems have and perform with a reliability that is on par with what is required in NASA's flight critical systems.

Competitiveness is another motive to use COTS or 3rd party software. Competitiveness, which may be summarized by the phrase of *faster, cheaper, better,* is defined by the observation that the cost ratio of COTS or 3rd party software to the overall system cost is trending downward over time. To accomplish this either substantial re-use of software certification has to occur or the cost to certify new software has to drop or both.

Another motivation to use COTS or 3rd party software is the need for large scale, use of ultra reliable software to support permanent human presence in space. Such software is found in ground based systems.

This paper has covered major issues in the certification of COTS or 3rd party software in NASA flight critical systems in terms of challenges and barriers. Challenges are strategic in nature and are often systemic, which means there is no one place or position from which to make a difference. Barriers are tactical: they can be addressed either by a well-positioned individual or dedicated team. Creation of suitability criteria for COTS software in mission critical systems needs strategic and tactical breakthroughs, some of which are organizational and some are technological.

To counter the challenges and barriers the paper identified a cross cutting vision of how advanced software related technology may aid in taking advantage of COTS software in flight critical systems. The ARINC 653 specification is an immediate candidate for applying such research.

Finally, a recommendation for a call to action in the form of sponsorship and participation in an open source forum dedicated to the adoption of COTS software in NASA safety critical systems was made.

Acknowledgments

The following colleagues helped significantly with observations and suggestions from their reviews: Rick Alena, Joe Coughlan, Misty Davies, Steve Jacklin, Mike Lowry, Tom Pressburger, and Peter Robinson.

References

[1]NASA Procedural Requirements 7150.2B, URL: http://nodis3.gsfc.nasa.gov/

[2]NASA Procedural Requirements 8705.2B, URL: http://nodis3.gsfc.nasa.gov/

[3]Avionics Application Software Standard Interface ARINC Specification 653-1, October 16, 2003, Published by Aeronautical Radio, Inc.

[4]RTCA/DO-178B, "Software Considerations in Airborne Systems and Equipment Certification," December 1, 1992.

[5] D.L. Dvorak, ed., NASA Study on Flight Software Complexity, 3 Mar. 2009, NASA Office of Chief Engineer, URL: http://www.nasa.gov/pdf/418878main_FSWC_ Final_Report.pdf.

[6]Kossiakoff, A., and Sweet, W. N., *Systems Engineering Principles and Practice*, John Wiley & Sons Inc., 2007, pp. 361

[7]Lyu, M.R., "Software Reliability Engineering: A Roadmap," *Future of Software Engineering (FOSE 2007)*, IEEE Computer Society Press, pp. 153,170

[8]Ibid 6, pp. 366, 367

[9]Alford, L. D., "The Problem with Aviation COTS," *IEEE AES Systems Magazine*, Vol. 16, No. 2 Feb. 2001, pp. 33,37

[10]Bertolino, A., "Software Testing Research: Achievements, Challenges, Dreams," *Future of Software Engineering, (FOSE 2007)*, IEEE Computer Society, pp. 85,103

[11] Lyu, M. R., *Software Reliability Engineering*, IEEE Computer Society Press, McGraw-Hill, 1996, pp. 19, 20

[12]Torres-Pomales, W., "Software Fault Tolerance: A Tutorial," NASA / TM-2000-210616. pp. 26

[13] Lyu, M. R., *Handbook of Software Reliability Engineering*, IEEE Computer Society Press, 1996

[14]Teng,X., Pham,H., "Reliability Modeling of Hardware and Software Interactions, and Its Applications," *IEEE Transactions On Reliability*, Vol. 55, No. 4, December 2006.

[15]Kelly, T.P. (1999): " Safety - A Systematic Approach to Managing Safety Cases," DPhil Thesis, Department of Computer Science, University of York, Department of Computer Science, University

[15]T. P. Kelly. "Arguing Safety – A Systematic Approach to Managing Safety Cases," PhD. Thesis, Department of Computer Science, University of York, York, UK, 1999.

[16]Storey, N., *Safety-Critical Computer Systems*, Addison Wesley, 1996, pp. 364, 365

[17]Elrad,T., Filman, R. E., Bader, A., "Aspect-oriented programming: Introduction," *Comm. of the ACM*, Vol. 44 Issue 10, Oct. 2001, pp. 29, 32

[18]NASA Open Source Software, URL: http://ti.arc.nasa.gov/opensource/

[19]Fitzgerald, B., "Open Source Software: Lessons from and for Software Engineering," IEEE Computer, Vol. 44, No. 10, Oct. 2011, pp. 25,30.

[20]Software Assurance Standard, NASA-STD-8739.8B w/Change l, July 28 2004, URL: http://www.hq.nasa.gov/office/codeq/doctree/87398.htm

[21] HRST Technology Assessments Technology Readiness Levels, A Chart, http://www.hq.nasa.gov/office/codeq/trl/