INVITED PAPER

Challenges and Solutions for Embedded and Networked Aerospace Software Systems

Development of real-time component-based systems, model-based system development, and the use of automated code generation, simulation techniques and desktop test methods are described.

By David C. Sharp, *Member IEEE*, Alex E. Bell, Jeffrey J. Gold, Ken W. Gibbar, Dennis W. Gvillo, Vann M. Knight, Kevin P. Murphy, Wendy C. Roll, Radhakrishna G. Sampigethaya, *Member IEEE*, Viswa Santhanam, *Member IEEE*, and Steven P. Weismuller

ABSTRACT | Aerospace systems are increasingly dependent upon software for their functionality, with associated software spanning a wide range of application domains. These include aircraft and spacecraft flight controls, mission computing, weapons management, command and control, surveillance, sensor management and processing, telemetry, and more. Understanding of their unique challenges has driven technology development on many fronts associated both with the products—such as real-time component-based application frameworks, supporting middleware, and algorithms—and the processes and tools by which they are created—such as model-based development and integration, automated code generation, simulations, and desktop test environments. This paper describes a number of these domains and challenges, future directions associated with networking and systems of systems, and technologies facilitating their development within The Boeing Company.

KEYWORDS | Aerospace systems; aircraft electronics; distributed computing; real-time systems

I. INTRODUCTION

Aerospace platforms rely on software-intensive systems for an increasing portion of their functionality across a wide range of application domains. The most fundamental functionality is maintaining controlled flight. For increased performance, modern aircraft are often aerodynamically unstable by design. Without high rate digital "inner loop" flight controls, such vehicles would quickly depart from controlled flight. Higher level flight management systems optimize routes according to various criteria such as reducing fuel, noise, or time in commercial flights or avoiding threats, intervisibility, or restricted "no-fly zones" in military operations. Even in manned aircraft, such flight management systems are increasingly coupled to flight control systems to provide auto-pilot capabilities.

In many cases, flight is only an enabler of achieving the primary mission objective. Aircraft perform intelligence, surveillance, and reconnaissance (ISR) missions employing sophisticated sensor systems as well as carry and deliver humanitarian aid, firefighting chemicals, and ordnance. Satellites provide communication links, broadcast navigation signals, track weather, map terrain, and image astronomical objects. Spacecraft explore distant planets and moons.

As evidenced above, aerospace systems include many functional domains. Many technology elements are common across multiple aerospace domains, as well as being shared with non-aerospace systems. Section II characterizes aerospace software systems via quality attributes. Section III describes challenges, approaches, and future

Manuscript received February 10, 2009. First published March 24, 2010; current version published March 31, 2010. Release approved via Boeing case public release 09-016.

The authors are with The Boeing Company, St. Louis, MO 63166 USA (e-mail: david.sharp@boeing.com; alex.e.bell@boeing.com; jeffrey.j.gold@boeing.com; kenneth.w.gibbar@boeing.com; dennis.w.gvillo@boeing.com; vann.m.knight@boeing.com; kevin.p.murphy@boeing.com; wendy.c.roll@boeing.com; radhakrishna.g.sampigethaya@boeing.com; vdot.santhanam@boeing.com; steve.weismuller@boeing.com).

Digital Object Identifier: 10.1109/JPROC.2009.2039631

0018-9219/$26.00 ©2010 IEEE

directions for both functional and quality attribute requirements within individual domains, including commercial aircraft systems; military flight controls and mission computing; intelligence, surveillance, and recognizance (ISR) systems; space systems (including the International Space Station and satellite systems); unmanned aircraft; and weapon systems. Section IV states conclusions.

II. AEROSPACE SOFTWARE SYSTEM DOMAIN CHARACTERISTICS AND COMMONALITIES

While different aerospace application domains have unique characteristics, they also share significant commonalities derived from their nature as embedded systems. A standard method of characterizing such traits is via quality attributes [1]. Quality attributes strongly influence software architecture decisions, especially those associated with cross-cutting aspects such as scheduling, data flow, and fault management.

Among quality attributes associated with system operation, performance is perhaps the most pervasive consideration. Hard or soft real-time performance of varying time durations drives fundamental decisions associated with execution determinism as well as speed. Safety is of course an overriding factor where lives or property are at risk. Either safety or mission success probabilities may drive availability considerations, often including the need for hardware redundancy and fault tolerance. As networking of systems expand and systems of systems are created, interoperability with external systems and security considerations grow in importance.

Other quality attributes characterize system development concerns. Aerospace systems are frequently in operation for decades before retirement, so long-term life cycle concerns such as maintainability and extensibility become central. System longevity drives portability so that hardware can be upgraded without redesigning software. The size of aerospace software systems feasibly developed often pushes the boundaries of engineering practice. Architectures supporting composability and reusability of software components are employed to meet these scalability challenges. Testability and integrability qualities must also be met to reduce the risk of successful system integration.

Although many of these quality attributes are shared across domains, the techniques used to provide these qualities are described in the subsequent domain sections to properly place them in an aerospace domain context.

A. Middleware

To simplify development for the large teams typically necessary to create aerospace systems, distributed, real-time embedded (DRE) middleware-based approaches are often used to move low level complexities of the architecture out of the applications. Industry solutions have enabled integrated heterogeneous systems, code portability, and location independence, but these must also address performance and other quality attributes. Middleware-related standards continue to be developed and expanded to enable cross-vendor portability and expand associated markets. Open source development is particularly effective for middleware capabilities where academia frequently takes a leading role and the lack of application domain intellectual property fosters greater sharing across companies. An example of successful standards-based open-source collaboration across academia and industry is the Adaptive Communication Environment (ACE). The ACE Orb (TAO) real-time object request broker was developed partially in conjunction with the Boeing Bold Stroke software product line [2] to gain the benefits of the Common Object Request Broker Architecture (CORBA) in real-time environments. The application of middleware within particular domains is described further within Section III.

Middleware is a highly active research field. One objective is to continue raising the abstraction of application programming models by moving domain independent commonality from applications into middleware component frameworks and expanding tool support for development and integration. For example, the CORBA Component Model has containers for applications that hide the "plumbing" and non-functional concerns from the application developer (e.g., threading, distribution). Again, this must be adapted to meet aerospace quality attributes. U.S. Department of Defense funded research has explored this for real-time systems to simplify application development [3].

As aerospace system networks grow in scale, the diversity of integrated functionality also increases. To integrate systems across diverse processing resources (e.g., full aircraft and handheld devices used for maintenance), criticalities (e.g., safety-critical and non-safety-critical), and determinism needs (e.g., real-time and non-real-time), middleware developers sometimes pursue multiple editions of products. These editions can separately deal with the concerns of real-time and non-real-time, highly embedded to mission computing, safety critical, air worthiness, and degrees of information assurance [4].

III. AEROSPACE SOFTWARE SYSTEM DOMAIN SPECIFICS

This section describes a wide range of aerospace software domains focusing on their functional and quality attribute challenges, approaches to meeting those challenges, and future directions.

A. Commercial Aircraft Systems

The commercial aircraft industry consists of manufacturers of large (over 100 seats) passenger airliners used in revenue flights. The field consists primarily of two manufacturers, Boeing and Airbus, who compete fiercely for market share [5].

Since the deregulation of the U.S. airline industry in 1978 [6], competition among operators has intensified both within the U.S. and globally. With jet fuel prices rising steadily since 1976, hitting record highs in 2008 [7], airlines became increasingly interested in fuel-saving models. Boeing responded, for example, by announcing its plan to offer the 787 *Dreamliner* aimed at achieving significant fuel economy.

Weight reduction tops the list of fuel economy initiatives. Although the use of composites in place of metal aircraft structures has the most direct impact on weight, other initiatives contribute as well. Integrated modular architectures (IMA) [8] replace traditional federated architectures leading to fewer line replaceable units (LRUs), lower power consumption and less wiring—all contributing to overall weight reduction. The trend toward IMAs, first included in commercial airliners in the 1990's [9], is enabled by the availability of higher throughput processors capable of hosting several unrelated or loosely related applications previously distributed over multiple LRUs.

Maintaining the integrity of safety critical IMA applications co-hosted with less critical applications has been achieved with partitioning operating systems based on the popular ARINC 653 standard [10]. The rigid separation of applications in memory space and time, however, comes at a cost. For example, use of processor acceleration features such as cache memories is limited. A (U.S.) Federal Aviation Administration (FAA) Certification Authorities Software Team Position Paper [11] outlines concerns such as cache coherency and timing jitters and calls for restraints on the use of caches in safety-critical applications. Such restrictions lower the obtained performance of modern processors below the levels achieved in less critical applications. This is a common tradeoff in real-time systems where increased determinism is achieved at the cost of reduced overall throughput.

More recently, IMA platform designs consider multi-core and multi-processor configurations to meet increased processing demands. Key features of multi-core processors, like cached memories and pipelined processors, however lead to highly non-deterministic execution profiles and pose special challenges in building demonstrably safe software systems.

More and more software systems are created with model-based development [12] (MBD) using higher-level, reusable domain-specific components to ease building and verifying complex software systems. MBD can be applied to many degrees, e.g., from generating configuration files used during initialization to generating the software for nearly-complete systems. Although studies have shown that MBD can reduce software development costs [13], [14], numerous challenges remain [15] that can wipe out savings if not addressed. For example, it is not uncommon to see code size and execution times increase when code is generated from models automatically.

A common use of MBD in aerospace systems is for control system development where control theory forms the mathematical underpinning for higher level system building blocks. A well-planned integration of control system MBD requires software design considerations to be factored into models generating software. In a more traditional approach, control engineers create analysis models without concern for software design issues that are then restructured by software engineers to form an efficient software design. Current practices emphasize interdisciplinary design teams where control engineers develop models meeting design constraints necessary for efficient software execution in collaboration with software engineers. Besides avoiding two separate design activities, this also has the advantage of having a single model that can be used for initial control law validation, target code generation, and test case generation.

Verifying software to the stringent standards of RTCA/DO-178B is increasingly difficult as the volume of software increases and code gets generated automatically from models. Challenges of verifying complex software systems are being met with increased tool automation. Automated test case generation, once dubbed as worthless because "it can only confirm the code does what it does", is making a comeback with MBD. Since models are often considered to depict low level requirements, tools that generate test cases from models to achieve structural coverage, such as Modified Condition/Decision Coverage (MCDC) [16], [17], are becoming more widely accepted. A case can be made that such auto-generated test cases help verify the software implementation against the models, especially if the models are independently validated through simulation.

Modeling packages such as MATLAB/Simulink and SCADE also include tools that perform property proofs using formal methods which could supplement or replace certain forms of testing [18]. Likewise, hand-written code can be shown to be free of certain anomalies that lead to system crash or memory corruption using tools such as SPARK [19].

Electronic distribution of loadable software and data (EDSD) contributes to a reduction of system weight from onboard storage media [20], [21]. Avionics suppliers deliver safety-assured software [22] to LRUs via different entities, and ground systems download avionics data such as software configuration as shown in Fig. 1. When updates and data for safety-critical software (e.g., flight control computer or DO-178B Level A) or business-critical software (e.g., cabin lights or DO-178B Level D) are distributed over insecure channels, security of the distribution becomes pivotal [23], [24].

Digital signatures offer a solution to protect the end-to-end integrity and authenticity of safety-assured loadable software and data, even in the presence of attackers [25]–[28]. For instance, software incompatibility across versions

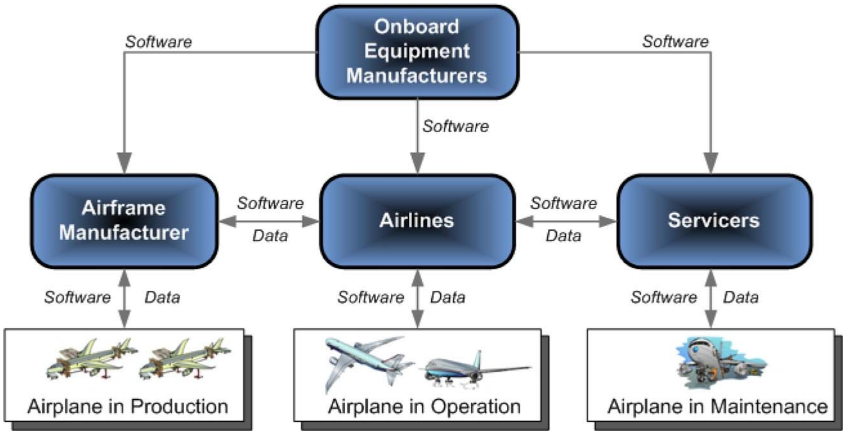

Fig. 1. ***Airplane software and onboard data distribution.***

and aircraft models is a vulnerability. One mitigation approach includes metadata with versions and timestamps to reject outdated software, and intended destinations to reject diverted software updates [28].

Formal methods, such as model checking, can establish with high confidence the attack immunity of EDSD systems. Formally assessing end-to-end security however is complex, time-consuming and expensive, requiring high assurance of a large number of connected components [27]. A promising solution is to transport software from source to destination via interaction of instances of a single reusable certified core component communicating over insecure links [29].

One engineer working on a digital fly-by-wire airplane design in the 1990's expressed a software-centric view of the relationship between the aircraft and the software, paraphrased here as: "The airplane is nothing more than a peripheral to the software." Although that is perhaps an extreme perspective, the role of software in modern aircraft has certainly grown from augmenting the human pilot to transforming the pilot into an operator of a complex network of interconnected computer systems that manage virtually every aspect of flight. To meet this complexity, new tools and techniques are emerging and regulatory guidelines are being updated to leverage them. DO-178C [30], which is expected to supersede DO-178B in late 2010, for example, will address aspects of software developed using MBD. Automated tools—including formal methods—will permeate every aspect of software development, from requirements capture to integration, and the verification of those tools will be increasingly relevant to the ultimate safety of the software-rich system. Accordingly, DO-178C addresses tool qualification in greater detail than its predecessor.

B. Military Flight Controls

In the United States, MIL-STD-882D governs the design of safety critical defense systems, including flight controls. MIL-STD-882D is somewhat less restrictive than commercial aircraft safety standards, but it still requires rigorous development processes and system verification.

Flight control systems have evolved from totally mechanical to hydro-mechanical systems to analog and/or digital stability/control augmentation systems. For instance, F/A-18A/B achieved first flight in 1978 with a full fly-by-wire (FBW) digital control system. This system although primarily digital retained both a backup analog Direct Electrical Link (DEL) mode for ailerons and rudders and mechanical backup control of the differential stabilators. The aircraft was significantly enlarged for the E/F version in 1995. By this time, the high reliability of the A/B/C/D flight control system combined with the additional instability of the airframe that prevented open loop vehicle control in either analog DEL or mechanical modes, justified the removal of both the analog DEL mode and the mechanical backup from earlier models.

Fly-By-Wire flight control systems are designed to prohibit single-point failures (where a single failure can lead to catastrophic failure) or sometimes two-point failures. This necessitates pervasive hardware redundancy. Each redundant path is referred to as a channel. The input portion of the path includes sensors, wiring and analog-to-digital conversion. The output portion of the path includes digital-to-analog conversion, wiring, and actuators. Each of these sub-paths typically has two, three, or four parallel and independent channels, depending on the criticality of the parameter. The inputs from one input channel come into one or more processing channels and are transmitted from the receiving channel to all processing channels. This allows any failures in inputs to be detected in all processing channels and each processing channel to work on the best inputs available. Various input signal management algorithms are used to determine the best value to use for a particular input. By comparing the redundant inputs from the pilot control stick, for example, hardware failures can be identified, isolated and eliminated in common signal selection logic. Control law processing then proceeds in multiple channels. Channels are typically synchronized so that inputs are sampled at near simultaneous times in order to keep cross channel tolerances very tight in the input signal management algorithms and prevent divergent outputs from being commanded by channels due to input sampling and frame completion time differences.

Rigorous design standards and reviews ensure that independence and redundancy is maintained through the complete path from inputs through processing to outputs. Any failure mode which is the same in all channels (e.g., a software bug) compromises the independence between channels and represents a potential single point failure. Redundant hardware is obviously essential for each channel, but software redundancy is sometimes applied as well. Challenging and costly "n-version" programming approaches entail different development teams producing independent software implementations from a single set of detailed algorithms and requirements. One variant of n-version programming is the NASA space shuttle which uses a quad-channel Primary Avionics Software System running identical software, and a Backup Flight System which has separately-developed software that is used only if all four channels of the primary system fail [31]. In practice, advanced fault management approaches have sometimes proven so difficult to properly design and integrate that they become a leading source of system instability. Cases exist where lack of trust in backup systems has led to either replacement of them with simpler solutions or reluctance to employ them when needed.

Pervasive built-in-test (BIT) capabilities before, during, and after flight can often entail 50% of the completed software. Every wire, every sensor channel, and every actuator channel must be testable to assess system health. Digitally-controlled analog signal multiplexers route multiple analog signals to shared analog-digital converters. When an anomaly is detected, readings from related signals are taken to isolate the problem. Failed signals may have their paths shut down to maintain safe operation.

Flight control systems require strict adherence to deterministic hard real time design tenets to meet performance requirements. Control law processing is inherently periodic. Timers generate interrupts at the highest rate of processing. These interrupts drive sensor inputs which then trigger input, processing, and output phases of each cycle. Increasingly unstable aircraft require higher rate control loops for controlled flight, but remain in the tens or hundreds of Hertz for fighter aircraft. In contrast to this needed determinism, microprocessor designs tend to be increasingly non-deterministic, with multi-level caches and speculative execution for example. In this context, the degree of determinism necessary becomes an important design consideration. What affects aircraft operation is determinism in sampling inputs and sending actuator commands. Processing jitter between these two times can be tolerated to increase processor throughput.

Recent trends in military aircraft have expanded the role for traditional flight control systems to control many other aircraft subsystems. These newer subsystems were appropriately renamed Vehicle Management Systems (VMS) and are used extensively in unmanned aircraft such as the X-45A Unmanned Combat Air Vehicle. To achieve autonomous flight, the VMS in X-45A includes air data, navigation, fuel, engine, power distribution, brakes, landing gear, environmental control, weapons bay doors and communications [32]. The X-45A VMS system also includes a contingency management system capable of autonomously performing flight phase dependent actions in the event of onboard failures that are normally handled procedurally by a pilot [33].

C. Avionics Mission Computing Systems

Avionics mission computing systems provide the capabilities directly used by aircrew to achieve their objectives. They reside on manned fighters, bombers, tankers, trainers and reconnaissance and cargo platforms. Typical capabilities include managing cockpit displays, monitoring aircraft health, determining navigational states, fusing sensor data for situation awareness, delivering weapons and cargo payloads, and aiding in formation flying. These systems combine high levels of user interaction with hard and soft real-time behaviors. They integrate, display, and react to information from many subsystems on and off the aircraft such as sensors, stores management, and data link subsystems.

Military aircraft are maintained and upgraded for decades. The B-52 entered service in 1955 and in 2000, it began a "Avionics Midlife Improvement," implying operation for another 45 years! Requirements for upgrades result from yearly U.S. Congressional appropriations. They vary in size and complexity as military needs and budgets fluctuate. This may include changing processing hardware, adding new sensor or weapon capabilities, or interoperating with new off board systems.

Aircraft size, power, and weight constraints limit processing resources. This domain, however, requires significant processing for accurate display of high speed aircraft, for example. In addition, hard real-time processing (i.e., determinism) is required to support weapon delivery. Ordnance release at a highly predictable time after the aircrew command requires low jitter to ensure accuracy. Complicating matters, many requirements are only active intermittently, for mission computing is highly modal, providing different functionality based on crew selected displays. There is also significant aperiodic functionality due to crew interaction with the system.

A traditional approach to meeting this real-time requirement is to fit inputs-processing-output sequences within one of a set of harmonically scheduled frames (i.e., periodic time slices). These are managed by a cyclic executive that uses a static call tree for each frame. This approach has been extended to incorporate variable execution priority and variably threaded execution environments to accommodate a wide range of aircraft physical architectures [34].

Capturing the necessary data to do schedulability analysis is daunting due to complex function dependencies and extensive branching. Rate monotonic analysis (RMA) without mode awareness data leads to highly pessimistic results, unacceptable where utilization rates of 80–90% are frequently desired. Therefore, basic theoretic principles are used (e.g., higher rate threads preempting lower rate threads) but empirical techniques are emphasized for detailed design.

"Mission criticality" identifies that portion of functionality essential to mission success. It is closely related to availability. While not critical to preserving life like safety critical systems, mission criticality still requires fault tolerant and fault mitigating behaviors to meet basic capabilities. Resource limitations make full hardware redundancy infeasible. Instead, a processor failure causes software reconfiguration on the remaining processors to preserve mission critical functionality while terminating less critical processing.

Prior to the 1990's, tightly coupled software and hardware resulted in costly software redesigns upon hardware upgrade. Oftentimes, by the time software was fielded, the hardware was already obsolete. With increased CPU performance, object-oriented strategies that isolate change and provide extensibility became feasible. Usage must be tempered by performance constraints, however. Relaxed instead of strict layering is used to optimize timing, for example [35]. The Boeing Bold Stroke product line multi-year development plan anticipated hardware upgrades by isolating hardware and operating system dependencies [36]. This allowed the system to meet performance requirements by incorporating Moore's Law-induced processor upgrades just prior to fielding without requiring significant software changes.

Many complex mission computing systems use lean error reporting rather than full exception handling to pinpoint the location of errors from the field. Resource constraints limit the amount of diagnostics that can be stored. In some cases, the fixed number of threads is leveraged to capture stack information for a thread when an error occurs. That data is transferred off the aircraft via removable media to perform analysis back in the lab.

Affordability goals have made software reuse a significant driver. The Bold Stroke product line was created to enable reuse across F/A-18, AV-8B, and F-15E aircraft, and later applied to T-45 and X-45. A common architecture was developed and common code was tailored to meet aircraft-specific requirements [37]. The architecture includes flexible event-based execution ordering, proxies for location independence, and other strategies [38]. The greatest reuse was achieved where requirements aligned. Because these aircraft had evolved independently, more reuse was possible in generic areas like infrastructure, and kinematics. When trainers with requirements similar to their fighter counterparts were added to the product line increased levels of reuse were possible.

Component-oriented architectures like Bold Stroke increase the flexibility of software composition, but increase configuration complexity. The integrator must address each variability point and meet capability and performance requirements via configuration code that instantiates components and connects them to other components. As systems grow to thousands of components and tens of thousands of connections, the configuration becomes daunting [39]. Hand-generated configuration code is error prone and more difficult to debug.

As another application of model-based techniques, model-based integration approaches address these configuration problems by using domain specific models to represent component deployments. Models are analyzed for unsatisfied connections, circular dependencies, and inefficiencies. Configuration code is then generated from the model [40]. Greater than 10× savings in time to detect and correct configuration errors has been measured using this approach in representative-scale experiments [41].

D. Intelligence, Surveillance, Reconnaissance (ISR) Systems

A diverse collection of systems fall under the umbrella of Intelligence, Surveillance, and Reconnaissance (ISR), including those that monitor airspaces, seas, land masses, and even outer space. Such monitoring activities are often the responsibility of air traffic control, shipboard defense, and fire detection systems just to name a few. Beyond serving solely in monitor roles, many ISR systems also attempt to predict a variety of end states within their realms of responsibility such as possible loss of separation between aircraft, projectile impact points, and potential enemy tactics to be defended against. Many ISR systems include some form of a "Common Operational Picture" (COP) to operators to foster shared situational awareness. For instance, in an airborne ISR system, a display centered on the ISR aircraft and showing all other sensed aircraft might be displayed to all operators on-board.

Regardless of specific responsibility, successful deployments of ISR systems—including the airborne systems that are the focus of this article—share many of the same challenges, approaches, and future directions. Two areas of particular relevance are information assurance and information integration.

Technological advances have contributed to new sensors and capabilities creating circumstances where military ISR systems have classified aspects to them in addition to unclassified ones. For instance, a single aircraft might carry multiple electro-optical cameras and radars, some of which have classified capabilities and some of which do not. A resulting complication of a system with multiple classification levels might require that they be partitioned in a way so that users without required clearances have no means of accessing unauthorized capabilities or information.

Both hardware and software separation techniques are used to meet security needs and isolate sensitive data and capabilities. A hardware separation approach for supporting access control relies on electronics to physically isolate any software executing at different classification levels.

One example of a hardware separation approach involves partitioning data and services into multiple security enclaves based on classification. Users are only allowed to log in to one enclave at a time based on their clearance level. Information can usually be moved in its entirety from enclaves with lower classification levels to higher ones. Moving information from higher classification levels to lower ones requires, if allowable at all, sanitization of classified data into a form that is appropriate for the classification level of the destination enclave. The movement of such data is supported by devices known as a Cross Domain Guards (CDG) and are typically configured with a set of rules governing which data is and is not allowed to flow between enclaves. For example, a rule might be configured to reduce the accuracy of sensor information available in a more sensitive enclave so it can be used in a lower classification enclave to avoid compromise of information related to sensor capabilities.

A software partitioning solution for supporting access control, however, is one where the operating system accommodates required separation from not only an access control aspect, but also ensures that the side affects of any failures are isolated to the relevant partition. Some software separation approaches such as ARINC 653 can be used to provide isolation between security domains as well as separating safety critical from non-safety critical software.

One such approach for supporting access control in a mixed classification environment is known as Multilevel Security (MLS). In general, this approach relies on the information processing system and underlying operating system to perform access control on all data and services based on the user's login credentials. All data and services are tagged with a classification level that is compared to user's clearance level before access is granted.

Software separation approaches provide a number of advantages over hardware separation that include deployment flexibility, reduced hardware footprint (by eliminating hardware-based cross domain guards), and eliminating dependency on special purpose computing hardware. The advantages of software separation, however, rely on continued evolution of computing kernels that support it as well as improved strategies for providing sufficient evidence to assure certification agents that required separation has been realized.

Beyond concerns related to authorization and access control, safeguarding sensitive information in ISR systems is also a challenge and becomes even more difficult when those systems are vehicles, for example, that are vulnerable to overrun by adversaries. A simple yet effective approach for rendering sensitive information unusable in the event of hostile overrun is to use an inline media encryptor (IME).

The IME uses a key to encrypt all data at rest (i.e., nonvolatile storage) that must also be used to access the data. In the event of overrun, the key is destroyed. In addition to provide a very fast means to render data inaccessible, it also makes it possible to recover the data by restoring the key if the danger of overrun ends.

ISR systems often rely on a diverse collection of sensors to supply the data used to provide a graphical view of the corresponding situational context. The task of integrating the inputs is not only a challenge from the perspective of diverse capability, but also accommodating technology insertions that are possible as the result of technological advancements. In addition, when a new ISR system is deployed, it is often the case that it must not only be interoperable with pre-existing sensors, but also be sufficiently flexible to accommodate new sensors as they become available as the result of technology advancements.

In the absence of standards that define the interface signatures of services as well as the data associated with the services, ISR systems often establish their own internal representation of, for example, sensor reports and use adaptors at system boundaries to transform non-compatible formats to a form expected by the ISR system. The approach largely eliminates any changes required to information integration software beyond characterizing a new source via configuration data. While the adaptor approach has proven itself as a successful tactic in many systems, having to write new software to adapt to new data sources is a constraint that limits "on the fly" interoperability.

Although distributed services made available by systems using Service Oriented Architectures (SOA) have the ability to provide an ISR system with diverse information, these services often suffer from the same lack of standards on interfaces and data formats. Such mismatches seriously limit interoperability regardless of the underlying architecture.

Data standardization efforts are important for improving interoperability. The Command and Control Information Exchange Data Model (C2IEDM) and its successor, JC3IEDM (Joint Consultation, Command and Control Information Exchange Data Model), are standardization efforts whose primary objective is to establish a common battle space language for joint forces. These two efforts are managed by the Multilateral Interoperability Programme (MIP).

Beyond standardizing only data, software interfaces used for accessing this data must undergo standardization as well. Interface standardization is occurring in isolated pockets such as in Communities of Interest, and within the simulation community. Some hope that interface standardization will be solved as part of the net-centric movement, and still others suggest that interface standardization is not an issue because software application should access standardized data directly from a database using a language such as structured query language (SQL). Overcoming these challenges will be necessary to achieve the interoperability goals of net-centric operations.

Beyond issues of integration and interoperability, stringent survivability and availability requirements sometimes found in ISR systems often require rapid recovery in the event of software or hardware failures. A common tactic for enabling survivability and availability involves periodically saving selected application state (i.e., checkpoint) so that it can be accessed and restored as part of COP recovery efforts. The difficulties of capturing a consistent snapshot of diverse and distributed application state from a temporal sense are further complicated in the face of an increasing number of sensors feeding advanced ISR systems.

The tactics used for capturing a consistent snapshot of situation data typically involve a software entity that has responsibility for orchestrating application persistence of local state to ensure there is a minimum of temporal inconsistency between elements of data being used to restore the COP in the event of failure recovery.

E. Satellite Systems

Spacecraft flight control is another domain in which unique challenges have driven and continue to drive technology development. During the 1960's, 70's and most of the 80's, prior to the advent of advanced satellite flight computers, spacecraft flight control logic was mostly executed on the ground because of the technical challenges of constrained space, weight and power, along with extreme environmental conditions that made on-board processing impractical. About the only processing that did occur on-board was command decoding and telemetry encoding with extremely limited general purpose computing capabilities. However, the nature of the spacecraft's mission demanded closed-loop control and greater autonomy. Therefore, as on-board computer technology advanced, more and more of the flight control software was implemented on the spacecraft. By 2000, flight control software autonomously managed the attitude of the spacecraft while also managing power and thermal control subsystems. Today, the introduction of very high performance microprocessors and programmable logic enables the introduction of sophisticated software architectures and designs that expand the capabilities of the spacecraft software control software significantly beyond what was envisioned four decades ago.

Several challenges within the spacecraft flight control software domain require special attention. Foremost among these challenges is the need for extremely high availability and reliability despite the fact that the spacecraft is only remotely accessible and performs in an extremely harsh environment. This also means that the flight control software must sometimes perform its mission autonomously and deal with unforeseen hardware anomalies.

Once it becomes operational, the flight control software is expected to execute flawlessly for the duration of the spacecraft system. That duration can be from a few months to fifteen or more years of operation. For instance, the MARISAT-F2 satellite was recently decommissioned after 32 years of service. These reliability expectations are a result of the critical missions that spacecraft undertake and their relatively high cost. A flight control application is expected to be autonomous in case interaction with ground controllers is infeasible or unexpectedly interrupted. Under more extreme conditions an anomaly condition may require the flight control software to safeguard the spacecraft until operators can re-establish control and resolve the anomaly. This safeguarding of the spacecraft, such that its normal operations are discontinued in order to assure that critical spacecraft systems are kept powered and thermally protected is referred to as safe-hold.

All of these challenges need to be met while accommodating typical embedded real-time software quality attributes. These challenges and others drive technology in the software architecture, design, testing, and tooling. Perhaps the most unique aspect of spacecraft flight control software however, is the rigorous qualification effort employed.

The ultra reliability requirement of spacecraft flight control software demands a rigorous qualification program. At the lowest levels, extensive and automated branch and logic testing is performed. The automated aspect of the testing helps implement automated regression testing later on. Once the low level "white box" testing is completed, the integrated software modules are "black box" tested at a system level by an independent verification and validation team. This final qualification effort is performed on a hardware-in-the-loop test system in order to verify integrated software and hardware operation.

An extraordinary test environment is used to support this rigorous qualification program. As previously described, the flight control software is branch and logic tested prior to being released and submitted for integration. The integrated product is then tested in a high fidelity simulation environment which re-uses the dynamics model, sensor models and actuator models, developed by system analysts as part of the activities to develop the flight control software requirements and algorithms. The emphasis on verification and testing at the lowest levels and testing under flight-like conditions cannot be overemphasized [42].

There are three instantiations of the simulation environment used for software test. The first is a pure software system in which the spacecraft flight control software image is executed on a simulation of the on-board microprocessor(s) with a simulation of the processor I/O and the system analytical models. This system is used for early integration testing, but is robust enough to use for most testing activities, with the exception of qualification testing.

In a second instantiation of the simulation environment shown in Fig. 2, the hardware emulator is replaced with a single board computer that is almost identical to the flight system on-board computer, except that the data input and output is via circuitry that emulates the flight hardware equivalent so that the I/O looks identical to the operational system. The data is passed to and from analytical models within the simulation computer. This system is used to create the qualification tests, pre-qualify the flight software, and support re-qualification testing.

A third instantiation of the simulation environment incorporates a flight computer and is used to qualify the flight software as well as the interfacing input and output circuitry. This third simulation environment, like the other two, also re-uses the dynamics model, sensor models and actuator models, originally developed by system analysts.

The basic historical pattern of moving ground-based processing to the satellite platform for increased spacecraft autonomy will continue. Ground-based processing which moves to orbiting spacecraft will tend to focus on payload support. This will allow for shorter delays in getting the right information to the right place. Closely related to this will be the need to accommodate ground-based communication network routing so that a satellite can function as a router in the sky. There is also likely to be a trend toward satellites that work in concert with other satellites and therefore software technologies associated with cooperative behavior will start to appear on-board. In the area of simulation and test, there is likely to be a need to support overall constellations and networks as well as a need for higher fidelity closed loop simulations involving sensors and systems.

F. Space Systems

The example presented here for space systems is the International Space Station (ISS). An example of the complex challenges presented to the ISS software development community is the phased design, development and construction of the forty-three incremental spacecraft leading up to the Assembly Complete International Space Station. Its design has been extended from a pair of computers on the first US module, interfacing with the Russian-provided FGB (Functionalui Germaticheskii Block), to the system described here. The ISS is a permanently manned, tele-operated system. The first time all components come together is on-orbit.

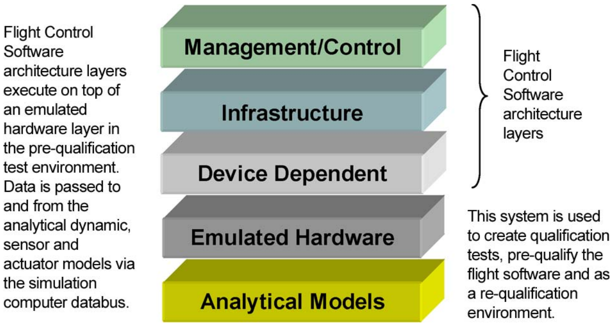
Flight Control Software architecture layers execute on top of an emulated hardware layer in the pre-qualification test environment. Data is passed to and from the analytical dynamic, sensor and actuator models via the simulation computer databus.
Management/Control
Infrastructure
Device Dependent
Emulated Hardware
Analytical Models
Flight Control Software architecture layers
This system is used to create qualification tests, pre-qualify the flight software and as a re-qualification environment.

Fig. 2. ***Spacecraft flight control software layers in simulation environment.***

A highly condensed representation of the ISS Command and Data Handling Architecture is presented in Fig. 3 [43]. Each rectangle represents a Multiplexer DeMultiplexer (MDM) or equivalent processor from international partners. Stacked symbols represent redundancy. Lines represent functional connectivity, not individual MIL-STD-1553 serial data buses.

The ISS avionics design consists of hierarchical tiers of MIL-STD-1553 data buses which are used for control and monitoring of the spacecraft systems. There are 104 1553 data buses in the US portion of the ISS, with 45 MDMs arranged in redundant pairs (except for the Command and Control Computers which include three machines).

The MDM is a 386 SX processor designed in the mid-1980's, which comes in several chassis sizes and is available with a standard 12 Mhz processor or an enhanced 16 Mhz processor. The use of Error Detection and Correction (EDAC) circuitry increases availability by protecting computer memory from single event radiation upsets. Techniques such as watchdog timers and memory scrubbing are used to maintain the integrity of the software loads, or detect and recover from their corruption. The Crew Interface Processor is a commercial laptop.

These many buses and processors are distributed across a very large space platform. The completed ISS is longer than a Boeing 767, and wider than it is long. The flight software supporting the first two flight elements totaled about 96 500 source lines of code (SLOCs). The running total of flight software delivered for the US elements is in excess of 1.8 million unique SLOCS. Over three million lines of simulation code were written to support the incremental test and certification of the software.

There are over 250 000 signal primitives in this system grouped into many words, groups, messages and packets. A signal primitive is defined as a measurement such as pressure, temperature, position, or a command parameter (e.g., valve position= > "Open"). It is maintained in the centralized Mission Build Facility (MBF). The MBF database produces all the artifacts required to reconfigure the MCC for each subsequent mission, e.g., command lists, telemetry formats.

The entire ISS could not be assembled and functionally exercised on the ground due to constraints imposed by gravity and the availability of flight hardware. The development schedule has currently spanned more than a decade and continues through 2011.

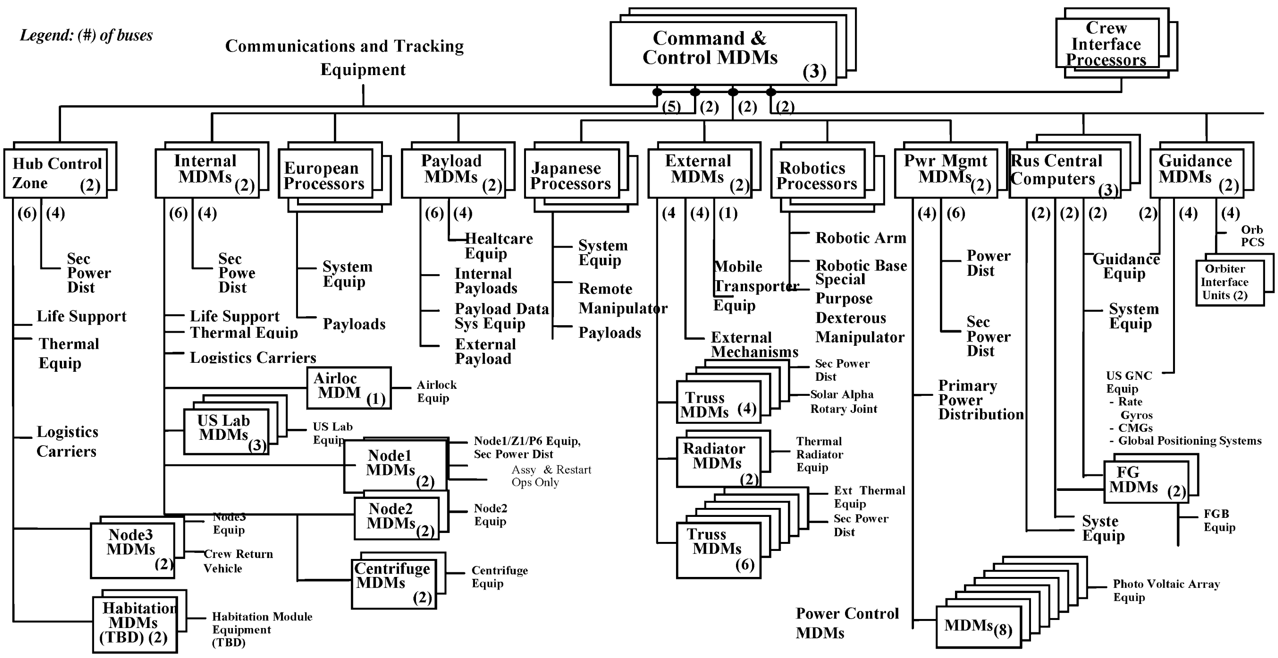

Fig. 3. ***International space station command and data handling summary architecture.***

Without a dedicated "Iron Bird" to use as a test/integration facility, simulations were used to stand in for flight hardware and software which was unavailable due to any of the following reasons: its development was moved later to accommodate funding profiles, the hardware had already been launched, or the hardware was geographically located elsewhere supporting some other development/integration activity.

The ISS program is geographically distributed over several development sites within the U.S. Canoga Park and Huntington Beach in CA, Huntsville, AL and Houston, TX were major development and integration centers for the software. Final integration and Launch Processing was accomplished at Kennedy Space Center (KSC) in FL. International portions of the ISS were developed in Canada (the robotic arm) Russia (the Russian Space Agency Service Module, and Functional Energy Block), Japan (NASDA's Japanese Experimental Module JEM), and in Italy and Germany for the European Space Agency (ESA) module.

The ISS program adopted a driving principal for the software architecture: development autonomy requires accountability. Increased risk due to development autonomy was mitigated by emphasis on integration early in the program. This evolved to periodic integration sessions in the lab. International partners were included in these integration activities.

The ISS is an example of the primary mission not being flight itself. The program's emphasis is on support of the scientific payload community. A planned upgrade will make it easier for the Payload Operations Control Center to manage data processing and communications resources the payloads require and provide better service to payloads.

G. Unmanned Aircraft

Unmanned Air Systems (UASs) are increasingly used in both civilian and military applications. UASs are used for a variety of purposes from reconnaissance to communications relay to weapons delivery. Systems are typically composed of one or more Unmanned Air Vehicles (UAVs) and one or more ground stations, often called Mission Ground Stations (MGSs). Ground stations are used to control and task vehicles. Vehicle control can range from a remotely piloted vehicle (e.g., a cockpit with stick and throttle on the ground) to completely autonomous vehicles. Fully autonomous vehicles can taxi, take off, perform their mission, return and land with minimal supervision from the ground station. Examples of UAV tasking include taking an image of a particular ground location, flying to a certain location, or following a particular route.

UAVs vary in size from small vehicles that can be carried in a backpack and deployed by an individual to gather reconnaissance (e.g., over the next hill) to large UAVs that are comparable in size to modern manned fighter aircraft. These larger vehicles are used for a variety of civilian and military missions. Ground stations may be composed of several elements. Typically there is an element for mission control and one for launch and recovery. Ground stations include computers that perform pre-mission planning and mission execution as well as communication systems.

Larger UAVs share many common elements with modern manned fighter aircraft including mission avionics, flight control, and guidance and navigations systems. Many of the earlier descriptions regarding manned aircraft therefore apply to unmanned systems as well. This section focuses on unique elements for unmanned systems.

Communication and navigation systems on unmanned systems take on increased levels of criticality compared to manned systems. Communication systems are critical in some phases of UAV flight. During flight to a mission area, for example, loss of communications for periods of time can be more easily tolerated but on final approach to a runway, or when operating alongside manned vehicles, or when flying in commercial air space robust communications systems are essential.

Navigation systems are essential to unmanned systems. Accurate position data is necessary to follow a route plan, stay in safe transit corridors and stay away from no fly areas. No pilot is available on-board to override system inaccuracies.

Similar to flight control systems, the most common approach to meeting this quality attribute is redundancy. Redundant communication systems (including antennas, radio links and on board processing to understand and act on commands from a ground station) are a common method of solving communication system criticality issues. Additional approaches include instructing the UAV to perform a series of actions in response to a radio link loss situation. Navigation reliability may be increased with brute force redundancy of the primary navigation sensor [e.g., Global Positioning System (GPS)] but may also be solved by using another system (e.g., an inertial sensor) or a combination of systems to determine position. For example, prior to GPS, aircrew often used inertial guidance systems for primary navigation and then during a mission the aircrew would perform a navigation update by flying over a known location and updating the system with the known position. The same technique could be done on a UAV using synthetic aperture radar or an optical sensor in conjunction with on board software to recognize discrete locations.

Unmanned systems include increasing levels of autonomy as technology advances. Motivation for autonomy varies, including enhancing vehicle safety and control, increasing mission effectiveness, or reducing workload on ground operators to allow one operator to control multiple vehicles. One current area of emphasis is the area of task allocation and on board routing. Robust autonomy in this area allows the vehicle to autonomously make or suggest changes to its mission (under supervision by the operator). A level of autonomy setting (possibly by function or component) would be set by the operator or mission plan and then followed by the UAV.

A concrete example of this would be for a UAV sensor to notice some change in the world state (e.g., a nearby aircraft or a new threat) and based on the level of autonomy alter the vehicle's flight path (or suggest alterations to the ground operator) to evade a threat or miss an approaching vehicle.

Solutions to autonomy include defining and implementing fine grained selection of level of autonomy (e.g., by function) coupled with rigorous testing and evaluation by high fidelity simulation systems. Experience shows that ground station operators are initially apprehensive about using higher levels of autonomy but after gaining experience they gain confidence in the autonomy and use it more.

With ever-increasing numbers of UAVs and those UAVs taking on increasingly complex missions UAV autonomy needs to be improved and enhanced to allow them to reduce operator workload and to minimize communication link bandwidth. A number of former and current U.S. Navy, U.S. Air Force Research Laboratory (AFRL) and DARPA programs pursue related research; including the DARPA Mixed-Initiative Control of Automata Teams (MICA) [44] and the U.S. Office of Naval Research (ONR) Intelligent Autonomy initiatives.

H. Weapon Systems

At the heart of a modern weapons system is a smart weapon: a weapon containing avionics that steer it to a prescribed impact point. The smart weapon is initialized with navigation and targeting information, and launched by the weapon launching system (e.g., aircraft, helicopter).

A typical smart weapon is in production for many years. During that time, some of the originally-specified avionics may go out of production, or may be supplanted by more capable avionics. Boeing may warrant the weapon to be free from defects for a prescribed number of years, and can expect to diagnose and repair weapons that have been returned by the customer. Boeing verifies correct weapon behavior before a new or updated weapon-type is used by the aircrew. This is accomplished by testing a sample of weapons at a government test range. Here, and only here, the weapon is outfitted with a telemetry kit that radios relevant flight software computational information to a ground station for real-time and post-flight analysis. Through the lifetime of the weapon-type, its capability is enhanced by occasional updates to its flight software. The updated software must be compatible with and loaded onto all vintages of already-manufactured weapons by the aircrew sometime before weapon use.

Given the relatively large number of units produced and their small size, affordability is a key objective for weapon systems. Weapon systems include extremely limited computing resources which arise both as a consequence of reducing weapon costs given the relatively large number of units produced, and as a result of continually adding capabilities to the flight software of smart weapons—some of which might have been manufactured a decade or more ago.

The weapons problem space demands extremely high first-time-quality flight software. Cost and schedule concerns prohibit multiple weapon flight tests as a means of perfecting the flight software. During a weapon flight test, the weapon cannot be halted or otherwise directly interacted with for purposes of diagnostic analysis/debugging. At the end of a flight test, the weapon avionics are necessarily destroyed and so are unavailable for further diagnostic analysis/debugging. Additionally, when a weapon is deployed with the aircrew, there can be no provisions for debugging software anomalies.

The weapons problem space also demands highly reliable hardware, and software that accurately identifies failed hardware. Due to weapons' extreme operating environments, intermittent and non-repeatable problems sometimes occur, and warranty efforts are benefited when these problems are expeditiously identified and resolved.

For a weapon, the radioed telemetry stream containing thousands of parameters is the most important artifact of a flight test. The challenge is to have a process and toolset that allow the telemetry stream to be changed quickly and accurately to support evolving test objectives, and that maximize the insight gained from all this collected data.

The evolving avionics hardware during many years of weapon production drives the requirement for the flight software to be compatible with multiple versions of various avionics. The challenge is to accomplish this software compatibility with a single version of the flight software, and with the minimum amount of software development expense and fielding complexities.

The traditional paradigm of cloning the existing software and the software support staff for each new, similar weapon project excessively strains the overall software support staff and leads to cost and schedule inefficiencies. To lower software development costs and reduce time-to-market for new but similar weapon-types, the software product line paradigm is being adopted [45] as earlier done for Bold Stroke. This paradigm is a good fit where there is similarity in the weapons' requirements and avionics. This approach also provides a useful and symbiotic framework for accommodating occasional avionics upgrades within a weapon. Per this paradigm, the trans-project software development team develops core assets (e.g., fin actuation and navigation) that can be used with a minimal amount of redevelopment and test across multiple weapon types.

Future directions of weapons systems are driven by a combination of new aircrew requirements and advances in avionics. To minimize the number of different weapon-types that must be procured, maintained, and carried on an aircraft, aircrews are increasingly asking for multi-role weapons. For instance, a single weapon that can be used in either an air-to-air or air-to-ground mode. This is made feasible with the smaller size and lower cost of electronics, and the availability of ever-higher computing resources—which both enable and require increasingly sophisticated on-weapon software solutions. To facilitate the prosecution of more-challenging targets, aircrews seek networking capabilities for weapons to take advantage of off-board sensors for providing precise real-time target position updates.

IV. CONCLUSION

Achieving safe flight is only the starting point for the functionality in software intensive aerospace systems. Significant ground-based elements support air vehicle operation and entail their own challenges. The complexity of these software intensive systems often becomes the pacing item for initial air vehicle development, and the centerpiece of upgrades for decades. As such, proper application of current techniques in addition to technology advances associated with both development processes and products are essential to feasibly develop and extend future aerospace systems. ■

Acknowledgment

As a survey of software intensive systems across a wide range of Boeing aerospace systems, this paper leverages accomplishments from large segments of the Boeing systems development community. In addition, the authors acknowledge Donald B. Akers and Richard V. Robinson who contributed directly to this paper.

REFERENCES

[1] *IEEE Standard for a Software Quality Metrics Methodology—Description*, IEEE Std. 1061-1998.

[2] D. C. Schmidt, D. Levine, and S. Mungee, "The design of the TAO real-time object request broker," *Computer Communications Special Issue on Building Quality of Service Into Distributed Systems*, vol. 21, no. 4, Apr. 1998, Elsevier Science.

[3] W. Roll, "Towards model-based and CCM-based applications for real time systems," in *International Symposium on Object-Oriented Real-Time Computing Proceedings*, 2003.

[4] M. Agrawal and L. Graba, "Distributed middleware requirements for disparate avionics and control software," in *24th Digital Avionics Systems Conference*, Oct. 2005.

[5] K. Kemp, *Flight of the Titans.* U.K.: Virgin Books, 2007.

[6] U.S. Centennial of Flight Commission. [Online]. Available: http://www.centennialofflight.gov/essay/Commercial_Aviation/Dereg/Tran8.htm

[7] Energy Information Administration, U.S. Department of Energy. [Online]. Available: http://tonto.eia.doe.gov/dnav/pet/hist/rjetnyhd.htm

[8] J. W. Ramsey, "Integrated modular avionics: Less is more," *Avionics Magazine*, Feb. 1, 2007.

[9] M. J. Morgan, "Integrated modular avionics for next generation commercial airplanes," Aerospace and Electronic Systems Magazine, IEEE, vol. 6, no. 8, pp. 9–12, Aug. 1991.

[10] C. Adams, "Product focus: ARINC 653 and RTOS," *Avionics Magazine*, Jul. 1, 2004.

[11] Certification Authorities Software Team, *Addressing Cache in Airborne Systems and Equipment*, Jun. 2003, CAST Position Paper 20.

[12] N. Gautam and O. P. Yadav, *Model Based Development and Auto Testing: A Robust Approach for Reliable Automotive Software Development*, Apr. 2006, SAE International Document 2006-01-1420.

[13] N. Tudor, M. Adams, P. Clayton, and C. O'Halloran, "Auto-coding/auto-proving flight control software," in *Digital Avionics Systems Conference*, Oct. 24–28, 2004.

[14] M. D. Schulte, "Model-based integration of reusable component-based avionics systems—A case study," in *9th International Symposium on Object-Oriented Real-Time distributed Computing (ISORC)*, Seattle, Washington, May 16–20, 2005.

[15] J. C. Knight, "Future trends of software technology and applications model-based development," in *Computer Software and Applications Conference*, Sep. 2006, vol. 1, p. 18.

[16] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol. 9, no. 5, pp. 193–200, Sep. 1994.

[17] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, "A Practical Tutorial on Modified Condition/Decision Coverage," NASA Technical Report NASA/TM-2001-210876.

[18] G. Berry, "Synchronous design of embedded systems: The Esterel/Scade approach," in *Formal Methods for Industrial Critical Systems: 12th International Workshop, FMICS 2007, Berlin, Germany, July 1–2, 2007, Revised Selected Papers*, 2008, Springer.

[19] J. Barnes, *High Integrity Software*, Addison-Wesley Publishing Co., 2003.

[20] *Boeing Frontiers*, vol. 4, no. 8. [Online]. Available: www.boeing.com/news/frontiers/archive/2005/december/ts_sf10.html

[21] G. Bird, M. Christensen, D. Lutz, and P. Scandura, "Use of integrated vehicle health management in the field of commercial aviation," in *Proceedings of NASA ISHEM Forum*, 2005.

[22] Radio Technical Commission for Aeronautics, *Software Considerations in Airborne Systems and Equipment Certification (RTCA/DO-178B)*, 1992.

[23] Federal Aviation Administration, "14 CFR part 25, special conditions: Boeing model 787-8 airplane; systems and data networks security isolation or protection from unauthorized passenger domain systems access, [docket no. NM364 special conditions no. 250701SC]," *Federal Register*, vol. 72, no. 71, Apr. 13, 2007. [Online]. Available: http://edocket.access.gpo.gov/2007/pdf/E7-7065.pdf

[24] Federal Aviation Administration, "14 CFR part 25, special conditions: Boeing model 787-8 airplane; systems and data networks security protection of airplane systems and data networks from unauthorized external access, [docket no. NM365 special conditions no. 250702SC]," *Federal Register*, vol. 72, no. 72, Apr. 16, 2007. [Online]. Available: http://edocket.access.gpo.gov/2007/pdf/07-1838.pdf

[25] Aeronautical Radio Inc. (ARINC), *Electronic Distribution of Software (ARINC 666)*, 2006.

[26] J. Pawlicki, J. Touzeau, and C. Royalty, *Data and Communication Security Standards in Practice*, 2006. [Online]. Available: http://www.ataebiz.org/forum/2006presentations/StandardsInPracticeAll.pdf, 2006.

[27] R. Robinson, M. Li, S. Lintelman, K. Sampigethaya, R. Poovendran, D. von Oheimb, J. Busser, and J. Cuellar, "Electronic distribution of airplane software and the impact of information security on airplane safety," in *Proceedings of the International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, 2007.

[28] K. Sampigethaya, R. Poovendran, and L. Bushnell, "Secure operation, control, and maintenance of future e-enabled airplane," in *Proceedings of the IEEE*, vol. 96, no. 12, Dec. 2008.

[29] M. Maidl, D. von Oheimb, P. Hartmann, and R. Robinson, "Formal security analysis of electronic software distribution systems," in *Proceedings of the International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, 2008.

[30] B. Carey, "Editor's Note: Software Standard," *Avionics Magazine*, Jun. 1, 2008.

[31] "Space Shuttle," *Wikipedia, The Free Encyclopedia*, Jan. 29, 2009, 21:05 UTC. 30 Jan. 2009. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Space_Shuttle&oldid=267264985

[32] K. W. Wise, "First flight of the X-45A Unmanned Combat Air Vehicle (UCAV)," in *AIAA Atmospheric Flight Mechanics Conference and Exhibit*, Austin, Texas, Aug. 11–14, 2003.

[33] R. W. Davidson, "Flight control design and test of the Joint Unmanned Combat Air System (J-UCAS) X-45A," in *AIAA 3rd "Unmanned Unlimited" Technical Conference, Workshop and Exhibit*, Chicago, Illinois, Sep. 20–23, 2004.

[34] D. C. Sharp, "Object-oriented real-time computing for reusable avionics software," in *International Symposium on Object-Oriented Real-Time Computing Proceedings*, 2001.

[35] F. Buschmann, R. Meunier, H. Rohnert, and P. Sommerlad, *Pattern-Oriented Software Architecture: A System of Patterns.* Wiley & Sons, 1996, p. 31 ff.

[36] D. C. Sharp, "Reducing avionics software cost through component based product line development," in *Software Technology Conference Proceedings*, Apr. 1998.

[37] L. M. Northrop and P. C. Clements, *A Framework for Software Product Line Practice, Version 5.0*: Carnegie Mellon University Software Engineering Institute. [Online]. Available: http://www.sei.cmu.edu/plp/framework.html

[38] B. S. Doerr and D. C. Sharp, "Freeing product line architectures from execution dependencies," in *Software Technology Conference*, May 1999.

[39] M. Effinger, C. Miller, W. Roll, D. Sharp, and D. Stuart, "Challenges and visions for model based integration of avionics systems," in *Digital Avionics Systems Conference Proceedings, 2001.*

[40] G. Karsai, S. Neema, B. Abbott, and D. Sharp, "A modeling language and its supporting tools for avionics systems," in *The 21st Digital Avionics Systems Conference, 2002. Proceedings*, 2002, vol. 1, pp. 6A3-1–6A3-13.

[41] M. D. Schulte, "Model-based integration of reusable component-based avionics systems—A case study," in *International Symposium on Object-Oriented Real-Time Computing Proceedings*, 2005.

[42] B. Tosney and S. Pavlica, "A successful strategy for satellite development and testing," in *Aerospace Corporation Crosslinks*. Fall, 2005.

[43] D. E. Cooke, M. Barry, M. Lowry, and C. Green, "NASA's exploration agenda and capability engineering," in *Computer*, vol. 39, no. 1, pp. 63–73, Jan. 2006.

[44] D. Corman, T. Herm, J. Paunicka, T. Sheperd, and T. Emrich, *Mixed Initiative Control of Automa-Teams, Open Experimentation Platform.* [Online]. Available: http://www.stormingmedia.us/34/3445/A344524.html

[45] Northrop, op cit.

ABOUT THE AUTHORS

David C. Sharp (Member, IEEE) received the B.S. degree in electrical engineering in 1984 from the University of Missouri-Rolla, USA, and the M.S. degree in electrical engineering from Stanford University in 1987.

He is a Senior Technical Fellow at Boeing, where he is currently Chief Software Architect for the US Army's Brigade Combat Team Modernization (formerly Future Combat Systems) program and was previously Lead Architect for the Boeing Bold Stroke software product line.

Alex E. Bell received a B.S. degree in systems engineering in 1982 from the University of California at San Diego.

He is an Associate Technical Fellow at The Boeing Company where he works as software architect on a number of airborne surveillance programs.

Jeffrey J. Gold earned the B.S. degree in physics and computer science in 1980 from the State University of New York at Albany and the M.S. degree in system engineering from the University of Southern California in 1994.

He is a Technical Fellow of the Boeing Company and has worked at the company's satellite division in southern California for over 25 years.

Ken W. Gibbar received the B.S. degree in electrical engineering in 1984 from the University of Missouri, Columbia, USA.

He is an Associate Technical Fellow at Boeing, and has 25 years experience at Boeing developing flight control/vehicle management systems and software for a wide range of manned and unmanned air vehicles and smart weapons.

Dennis W. Gvillo received the B.S. degree in mechanical engineering in 1984, and the M.S. degree in computer science in 1987, both from the University of Illinois, Urbana, Illinois, USA.

He is an Associate Technical Fellow at Boeing, developing flight software for a range of unmanned autonomous air vehicles and smart weapons.

Vann M. Knight earned a B.S. in electrical engineering from Washington University in St. Louis, Missouri, USA in 1984.

He is a Technical Fellow at The Boeing Company and has over 20 years experience in real-time embedded systems software development and software architecture primarily in military avionics on manned and unmanned systems.

Kevin P. Murphy earned a B.S. in Aerospace Engineering from the University of Texas (Austin) in 1982.

He is an Associate Technical Fellow at The Boeing Company, where he has worked for 27 years in the area of Human Space Flight at Johnson Space Center, almost entirely in the area of embedded systems.

Wendy C. Roll earned a B.S. in mathematics at The University of Tulsa, Tulsa, Oklahoma in 1987.

She is an Associate Technical Fellow at The Boeing Company and has over twenty years of experience in distributed real-time embedded systems software development and software architecture on various military avionics and land-based systems.

Radhakrishna G. Sampigethaya (Member, IEEE) received the Ph.D. degree in electrical engineering from the University of Washington, Seattle, in 2007.

He is an Advanced Computing Technologist with Boeing Research & Technology, working on the security evaluation of e-enabled aircraft network applications such as loadable software and data distribution.

Viswa Santhanam (Member, IEEE) received a B.Tech. in Electrical Engineering from the Indian Institute of Technology, Kanpur in 1971 and a Ph.D. in Computer and Information Science from the Ohio State University in 1975.

He served on the Computer Science Faculty at Wichita State University from 1975 to 1987. He joined Boeing in 1987 where he currently holds the position of Technical Fellow. At Boeing, he has worked on a number of embedded software applications including the 777 Primary Flight Controls, the 767 Aerial Refueling System, and currently the 787 Flight Control System.

Steven P. Weismuller received the M.S. and B.S. degrees in computer science from the University of Houston in 1989, 1988 respectively.

He has worked on a number of NASA and defense systems, including serving as Chief Software Engineer for the US Army's Future Combat Systems program.